tment ay.”
acebook

Appl
icati
on
Sec

Thiel

iOS

Developers

urity

The

Definitive

Guide *for*

David Thiel

Hackers

and

Foreword by Alex Stamos

iOS ApplicAtiOn Security

the Definitive Guide for Hackers and Developers

by David Thiel

San Francisco

Printed in USA

First printing

20 19 18 17 16 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-601-X ISBN-13: 978-1-59327-601-0

Publisher: William Pollock Production Editor: Alison Law Cover Illustration:
Garry Booth Interior Design: Octopod Studios Developmental Editor: Jennifer
Griffith-Delgado Technical Reviewer: Alban Diquet Copyeditor: Kim Wimpsett
Compositor: Alison Law Proofreader: James Fraleigh

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc. 245 8th Street, San Francisco, CA 94103 phone:
415.863.9900; info@nostarch.com www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Names: Thiel, David, 1980- author. Title: iOS application security : the definitive guide for hackers and
developers / by David Thiel. Description: San Francisco : No Starch Press, [2016] | Includes index. Identifiers:
LCCN 2015035297| ISBN 9781593276010 | ISBN 159327601X Subjects: LCSH: Mobile computing--Security
measures. | iPhone
(Smartphone)--Mobile apps--Security measures. | iPad (Computer)--Security measures. | iOS (Electronic resource) | Application
software--Development. | Objective-C (Computer program language) Classification: LCC QA76.9.A25 T474 2016 | DDC
004--dc23 LC record available at http://lccn.loc.gov/2015035297

To whomever I happen to be dating right now.

And to my parents, for attempting to restrict my computer access as a child.

Also cats. They're pretty great.

About the Author

David Thiel has nearly 20 years of computer security experience. Thiel's research and book

Mobile Application Security (McGraw-Hill) helped launch the field of iOS application security, and he has pre- sented his work at security conferences like Black Hat and DEF CON. An application security consultant for years at iSEC Partners, Thiel now works for the Internet.org Connectivity Lab.

About the technical reviewer

Alban Diquet is a software engineer and security researcher who special- izes in security protocols, data privacy, and mobile security, with a focus on iOS. Diquet has released several open source security tools, such as SSLyze, iOS SSL Kill Switch, and TrustKit. Diquet has also presented at various security conferences, including Black Hat, Hack in the Box, and Ruxcon.

BRIEF CONTENTS

CONTENTS IN DETAIL

Foreword

Prior to the digital age, people did not typically carry a cache of sensitive personal information with them as they went about their day. Now it is the person who is not carrying a cell phone, with all that it contains, who is the exception. . . .

Modern cell phones are not just another technological convenience. With all they contain and all they may reveal, they hold for many Americans "the privacies of life". . . . The fact that technology now allows an individual to carry such information in his hand does not make the information any less worthy of the protection for which the Founders fought.

— Chief Justice John Roberts, Riley v. California (2014)

Few would argue that the smartphone has been, by far, the most impactful technological advance of the 21st century. Since the release of the iPhone in 2007, the number of active smartphones has skyrocketed. As I write this at the end of 2015, there are nearly 3.4 billion in use; that's one for just about half the human population (somewhere over 7.3 billion). Globally, phones have easily eclipsed all other types of computers used to access the Internet, and an entire book could be filled with examples of how near-ubiquitous access is shaping human civilization. Mobile is changing the world, and has enriched countless lives by bringing widespread access to educational resources, entertainment, and unprecedented economic oppor- tunities. In some parts of the world, mobile connectivity and social network- ing has even led to the downfall of autocratic regimes and the realignment of societies.

Even the septuagenarians on the US Supreme Court have recognized the power of modern mobile computing, setting new legal precedents with judgements, like Riley v. California quoted above, that recognize that a smartphone is more than just a device—it is a portal into the private aspects of everyone's lives.

Like all technological revolutions, the mobile revolution has its down- sides. Our ability to connect with the far side of the world does nothing to improve the way we communicate with those in front of our faces, and mobile has done nothing to eliminate the world's long-established economic disparities. At the same time, as with enterprise computing, personal com- puting, and networking revolutions, smartphones have introduced new kinds of potential security flaws, and introduced or reinvented all kinds of secu- rity and safety issues.

While the proto-smartphones released prior to 2007 brought us several important technological innovations, it was the subsequent publishing of rich SDKs and the opening of centralized app stores that turned the new mobile computers into platforms for third-party innovation. They also created a whole new generation of developers who now need to adapt the security lessons of the past to a new, uncertain threat landscape.

In the ten years I have known David Thiel, I have constantly been impressed by his desire to examine, disassemble, break, and understand the latest technologies and apply his knowledge to improving the security of others. David was one of the first people to recognize the fascinating secu- rity challenges and awesome potential of the iPhone, and since the first days of what was then the iPhone OS SDK, he has studied the ways app developers could stumble and expose their users to risk, or rise above the limitations of the platform to build privacy- and safety-enhancing applications.

This book contains the most thorough and thoughtful treatment of iOS security that you can find today. Any iOS developer who cares about their customers should use it to guide their product, architecture, and engineer- ing decisions and to learn from the mistakes that David has spent his career finding and fixing.

The smartphone revolution has tremendous potential, but only if we do the utmost to protect the safety, trust, and privacy of the people holding these devices, who want to enrich their lives through our inventions.

Alex Stamos Chief Security Officer, Facebook

Acknowledgments

Thanks to Jennifer Griffith-Delgado, Alison Law, Bill Pollock, and the rest of the No Starch team, as well as Tom Daniels for his major contributions to Chapter 9, and Alban Diquet and Chris Palmer for their excellent review and feedback.

INTRODUCTION

Much has been written regarding iOS’s security model, jailbreaking, finding code execution vulnerabilities in the base OS, and other security-related characteris- tics. Other work has focused on examining iOS from a forensic perspective, including how to extract data from physical devices or backups as part of criminal investigations. That information is all useful, but this book aims to fill the biggest gaps in the iOS literature: applications.

Little public attention has been given to actually writing secure applica- tions for iOS or for performing security evaluations of iOS applications. As a consequence, embarrassing security flaws in iOS applications have allowed for exposure of sensitive data, circumvention of authentication mechanisms, and abuse of user privacy (both intentional and accidental). People are using iOS applications for more and more crucial tasks and entrusting them with a lot of sensitive information, and iOS application security needs to mature in response.

As such, my goal is for this book is to be as close as possible to the canon- ical work on the secure development of iOS applications in particular. iOS is a rapidly moving target, of course, but I’ve tried to make things as accu- rate as possible and give you the tools to inspect and adapt to future API changes.

Different versions of iOS also have different flaws. Since Apple has “end- of-lifed” certain devices that developers may still want their applications to run on (like the iPad 1), this book covers flaws present in iOS versions 5.*x* to 9.0 (the latest at the time of writing) and, where applicable, discusses risks and mitigations specific to each version.

Who This Book Is For

First, this is a book about security. If you're a developer or security specialist looking for a guide to the common ways iOS applications fail at protecting their users (and the options available to you or a client for patching those holes), you're in the right place.

You'll get the most out of this book if you have at least a little experience with iOS development or a passing familiarity with how iOS applications work under the hood. But even without that knowledge, as long as you're an experienced programmer or penetration tester who's not afraid to dig in to Apple's documentation as needed, you should be fine. I give a whirlwind tour of Objective-C and its most commonly used API, Cocoa Touch, in Chap- ter 2, so if you need some high-level basics or a refresher on the language, start there.

What's in This Book

I've been performing a wide variety of iOS application security reviews and penetration tests since about 2008, and I've collected a lot of knowledge on the pitfalls and mistakes real-world developers encounter when writing iOS applications. This book boils down that knowledge to appeal both to iOS developers looking to learn the practice of secure development and to security specialists wanting to learn how to spot problems in iOS security.

How This Book Is Structured

In **Part I: iOS Fundamentals**, you'll dig in to the background of iOS, its security history, and its basic application structure.

• **Chapter 1: The iOS Security Model** briefly examines the iOS security model to give you an idea of the platform's fundamental security protec- tions and what they can and cannot provide.

• **Chapter 2: Objective-C for the Lazy** explains how Objective-C differs from other programming languages and gives a quick overview of its ter- minology and design patterns. For seasoned Objective-C programmers,

this may not be new information, but it should be valuable to beginners and others dabbling in iOS for the first time.

• **Chapter 3: iOS Application Anatomy** outlines how iOS applications are structured and bundled and investigates the local storage mechanisms that can leak sensitive information.

In **Part II: Security Testing**, you'll see how to set up your security testing environment, for use either in development or in penetration testing. I'll also share some tips for setting up your Xcode projects to get the most out of the available security mechanisms.

• **Chapter 4: Building Your Test Platform** gives you all the information that you need to get started with tools and configurations to help you audit and test iOS applications. This includes information on using the Simulator, configuring proxies, bypassing TLS validation, and analyzing application behavior.

• **Chapter 5: Debugging with lldb and Friends** goes deeper into monitor- ing application behavior and bending it to your will using lldb and Xcode's built-in tools. This will help you analyze more complex prob- lems in your code, as well as give you a test harness to do things like fault injection.

- **Chapter 6: Black-Box Testing** delves into the tools and techniques that you'll need to successfully analyze applications that you don't have source code for. This includes basic reverse engineering, binary modification, copying programs around, and debugging on the device with a remote instance of lldb.

In **Part III: Security Quirks of the Cocoa API**, you'll look at common security pitfalls in the Cocoa Touch API.

- **Chapter 7: iOS Networking** discusses how networking and Transport Layer Security work in iOS, including information on authentication, certificate pinning, and mistakes in TLS connection handling.
- **Chapter 8: Interprocess Communication** covers interprocess communi- cation mechanisms, including URL schemes and the newer Universal Links mechanism.
- **Chapter 9: iOS-Targeted Web Apps** covers how web applications are integrated with iOS native apps, including working with web views or using JavaScript/Cocoa bridges such as Cordova.
- **Chapter 10: Data Leakage** discusses the myriad ways that sensitive data can unintentionally leak onto local storage, to other applications, or over the network.
- **Chapter 11: Legacy Issues and Baggage from C** gives an overview of C flaws that persist in iOS applications: stack and heap corruption, format string flaws, use-after-free, and some Objective-C variants of these classic flaws.

- **Chapter 12: Injection Attacks** covers attacks such as SQL injection, cross- site scripting, XML injection, and predicate injection, as they relate to iOS applications.

Finally, **Part IV: Keeping Data Safe** covers issues relating to privacy and encryption.

- **Chapter 13: Encryption and Authentication** looks at encryption best practices, including how to properly use the Keychain, the Data Protection API, and other cryptographic primitives provided by the CommonCrypto framework.
- **Chapter 14: Mobile Privacy Concerns** ends the book with a discussion of user privacy, including what collecting more data than needed can mean for both application creators and users.

By the end of this book, you should be well equipped to grab an appli- cation, with or without source code, and quickly pinpoint security bugs. You should also be able to write safe and secure applications for use in the wider world.

Conventions This Book Follows

Because Objective-C is a rather verbose language with many extremely long class and method names, I've wrapped lines in source code listings to maxi- mize clarity. This may not reflect the way you'd actually want to format your code. In some cases, the results are unavoidably ugly—if wrapping makes the code seem less clear, try pasting it into Xcode and allowing Xcode to reformat it.

As I will detail in Chapter 2, I favor the traditional Objective-C infix notation instead of dot notation. I also put curly braces on the same line as method declarations for similar reasons: I'm old.

Objective-C class and method names will appear in `monospaced` font. C functions will appear in `monospaced` font as well. For brevity and cleanli- ness, the path */Users/<your username>/Library/Developer/CoreSimulator/* will be referred to as *$SIMPATH*.

A Note on Swift

There's been much interest in the relatively new Swift language, but you'll find I don't cover it in this book. There are a few reasons why.

First, I have yet to actually come across a production application written in Swift. Objective-C is still far and away the most popular language for iOS applications, and we'll be dealing with code written in it for many years to come.

Second, Swift just has fewer problems. Since it's not based on C, it's easier to write safer code, and it doesn't introduce any new security flaws (as far as anyone knows).

Third, because Swift uses the same APIs as Objective-C, the security pitfalls in the Cocoa Touch API that you may run into will be basically the

same in either language. The things you learn in this book will almost all apply to both Objective-C and Swift.

Also, Swift doesn't use infix notation and square brackets, which makes me sad and confused.

Mobile Security Promises and Threats

When I first started working with mobile applications, I honestly questioned the need for a separate mobile application security category. I considered mobile applications to be the same as desktop applications when it came to bugs: stack and heap overflows, format string bugs, use-after-free, and other code execution issues. While these are still possible in iOS, the security focus for mobile devices has expanded to include privacy, data theft, and malicious interprocess communication.

As you read about the iOS security specifics I cover in this book, keep in mind that users expect apps to avoid doing certain things that will put their security at risk. Even if an app avoids overtly risky behaviors, there are still several threats to consider as you fortify that app's defenses. This section discusses both security promises an app makes to its users and the types of attacks that can force an app to break them.

What Mobile Apps Shouldn't Be Able to Do

Learning from the design mistakes of earlier desktop operating systems, the major mobile operating systems were designed with application segregation in mind. This is different from desktop applications, where any application a user runs more or less has access to all that user's data, if not control of the entire machine.

As a result of increased focus on segregation and general improve- ments in the mobile OS arena, user expectations have expanded. In gen- eral, mobile applications (including yours) should be unable to do a few key things.

Cause Another Application to Misbehave

Applications shouldn't be able to crash or meddle with other applications. In the bad old days, not only could other applications generally read, mod- ify, or destroy data, they could take down the entire OS with that data. As time went on, desktop process segregation improved but primarily with the goal of increasing stability, rather than addressing security or privacy concerns.

Mobile operating systems improve upon this, but total process segre- gation is not possible while fulfilling users' interoperability needs. The boundary between applications will always be somewhat porous. It's up to developers to ensure that their applications don't misbehave and to take all prudent measures to safeguard data and prevent interference from malicious applications.

Deny Service to a User Given that iOS has historically been used primarily on phones, it's crucial that an application not be able to do something that would prevent the user from making an emergency call. In many places, this is a legal requirement, and it's the reason for protective measures that keep attackers (and users) from tampering with the underlying OS.

Steal a User's Data An application should not be able to read data from other applications or the base OS and deliver it to a third party. It should also not be able to access sensitive user data without the permission of the user. The OS should keep applications from reading data directly from other application's data stores, but preventing theft via other channels requires developers to pay attention to what IPC mechanisms an application sends or receives data on.

Cost the User Unexpected Money Apps shouldn't be able to incur charges without the user's approval. Much of the mobile malware that has been found in the wild has used the ability to send SMS messages to subscribe the user to third-party services, which pass charges through to the user's phone provider. Purchases made within the application should be clear to the user and require explicit approval.

Classifying Mobile Security Threats in This Book

To help understand mobile device security threats and their mitigations, it's also useful to keep a few attack types in mind. This keeps our analysis of threats realistic and helps to analyze the true impact of various attacks and their defenses.

Forensic Attacks Forensic attackers come into possession of a device or its backups, intending to extract its secrets. Most often, this involves examination of the physical storage on the device. Because phone or tablet theft is relatively easy and common compared to stealing other computing devices, much more atten- tion is placed on forensics.

Forensic attacks can be performed by either an opportunistic attacker or a skilled attacker targeting a specific individual. For opportunistic attackers, extracting information can be as simple as stealing a phone without any PIN protection; this allows them to steal images, notes, and any other data nor- mally accessible on the phone. It can also assist an attacker in compromising services that use two-factor authentication in conjunction with a phone- based token or SMS.

A skilled forensic attacker could be a rogue employee, corporation, government, law enforcement official, or perhaps really motivated extor- tionist. This kind of attacker knows the techniques to perform a temporary jailbreak, crack simple PINs, and examine data throughout the device's file- system, including system-level and application-level data. This can provide an attacker with not just data presented through the UI but the underlying cache information, which can include screenshots, keystrokes, sensitive information cached in web requests, and so forth.

I'll cover much of the data of interest to forensic attackers in Chapter 10, as well as some further protective measures in Chapter 13.

Code Execution Attacks Remote code execution attacks involve compromising the device or its data by execution of code on the device, without having physical possession of the device. This can happen via many different channels: the network, QR codes or NFC, parsing of maliciously crafted files, or even hostile hardware peripherals. Note that after gaining code execution on a device, many of the forensic attacks used to expose user secrets are now possible. There are a few basic subtypes of code execution attacks that frequently result from lower-level programming flaws, which I'll discuss in Chapter 11.

Web-Based Attacks Web-based remote code execution attacks primarily use maliciously crafted HTML and JavaScript to mislead the user or steal data. A remote attacker either operates a malicious website, has taken over a legitimate website, or simply posts maliciously crafted content to a public forum.

These attacks can be used to steal data from local data stores such as HTML5 database storage or localStorage, alter or steal data stored in SQLite databases, read session cookies, or plant a fake login form to steal a user's credentials. I'll talk more about web application–related issues in Chapter 9 and Chapter 12.

Network-Based Attacks Network-based code execution attacks attempt to gain control over an application or the entire system by injecting executable code of some type over the network. This can be either modification of network traffic com- ing into the device or exploitation of a system service or the kernel with a code execution exploit. If the exploit targets a process with a high degree of privilege, the attacker can gain access not only to the data of a specific application but to data all over the device's storage. They can also monitor the device's activity and plant backdoors that will allow later access. I'll talk specifically about network-related APIs in Chapter 7.

Attacks That Rely on Physical Proximity Physical code execution attacks tend to be exploits that target devices using communications such as NFC or the USB interface. These types of attacks have been used for jailbreaking in the past but can also be used to compro- mise the device using brief physical interaction. Many of these attacks are on the OS itself, but I'll discuss some issues relating to physical proximity in Chapter 14.

Some Notes for iOS Security Testers

It's my strong belief that penetration tests should be performed with source code if at all possible. While this is not representative of the position of most external attackers, it does maximize the ability to find important bugs within a limited time frame. Real-world attackers have as much time as they care to spend on analyzing your application, and Objective-C lends well to reverse engineering. They'll figure it out, given the time. However, most penetration tests are limited by time and money, so simulating a real-world attacker should not usually be the goal.

I cover both white-box (that is, source-assisted) and black-box method- ologies in this book, but the focus will be on source-assisted penetration tests because this finds more bugs faster and helps with learning the standard Cocoa library. Many techniques I describe in this book lend well to either approach.

All that said, iOS developers come from many different disciplines, and each person's skill set affects the types of security issues that slip into an app unnoticed. Whether you're testing someone else's application or trying to poke holes in your own, keep in mind a few different development backgrounds as you test.

Some iOS developers come from a C or C++ background, and since we all tend to use what we know, you'll find their codebases often use C/C++ APIs rather than Cocoa equivalents. If you know an application under test was created by former C/C++ programmers, you may find Chapter 11 to be useful reading because it discusses issues commonly found in straight C/C++ code.

For some new programmers, Objective-C is actually their first program- ming language. They often haven't learned that many vanilla C APIs, so ideally, you'll find fewer of those issues. There's also the rare wizened NeXTStep programmer who's made the move to OS X or iOS, with a library of collected wisdom regarding NeXTStep/Cocoa APIs but less mobile experience. If either sounds like you or your client, you'll find the chapters in Part III most helpful.

Programmers with Java backgrounds might try to force Java design patterns onto an application, endlessly abstracting functionality. Web developers who have been drafted into writing a mobile application, on the other hand, may try to wrap as much code as possible into a web app, writing minimal applications that rely on WebKit to view application content. Check out Chapter 9 for some WebKit-related pitfalls.

Developers with the last few skill sets I mentioned are less likely to use low-level APIs, which can prevent classic C flaws. They are, however, unlikely to spot mistakes when using those low-level APIs, so you'll want to pay close attention if they use them.

Of course, none of these backgrounds is necessarily better suited to secure development than the others—both high-level and low-level APIs can be abused. But when you know how existing skills can affect the writing of iOS applications, you're a step closer to finding and solving security issues.

My own background is that of a penetration tester, which I consider akin to being an art critic: I *can* write code, but the vast majority of my time is spent looking at other people's code and telling them what's wrong with it. And like in the art world, the majority of that code is rather crap. Unlike the art world, however, code problems can often be fixed with a patch. My hope is that at the end of this book, you'll be able to spot bad iOS code and know how to start plugging the holes.

PART I

1

THE IOS SECURITY MODEL

Let’s give credit where credit is due: Apple has been pretty successful in keeping malicious software out of the App Store (as far as I know). But the application review process can be a frustrating black box for devel- opers. The process used by Apple’s reviewers is not publicly documented, and sometimes it’s simply not clear what functionality is and isn’t permitted. Apple gives some decent guidelines,[1] but apps have been rejected based on criteria that apply to accepted appli- cations as well.

Of course, what qualifies as malicious is defined by Apple, not by users. Apple uses the App Store as a way to control what functionality is available on the iOS platform, meaning the only way to obtain certain functionality is to jailbreak the device or subvert the App Store review process. An example of this is the Handy Light application, which masqueraded as a flashlight application but contained a hidden mode to enable device tethering.[2]

1. *https://developer.apple.com/appstore/resources/approval/guidelines.html* 2.

http://www.macworld.com/article/1152835/iphone_flashlight_tethering.html

The app review process on its own will never catch all sophisticated (or trivial) malicious applications, so other mechanisms are needed to effectively keep bad applications from affecting the wider OS environment. In this chapter, you'll learn about the architecture of iOS's security mechanisms; in later chapters, you'll dig in to how to take advantage of these mechanisms properly in your own programs.

Let's take a quick look at the fundamental security components iOS implements to prevent exploits and protect data. I'll dive deeper into the actual mechanics of most of these in later sections, but I'll start by giving a broad overview of the impetus behind them and their utility.

Secure Boot

When you power on an iOS device, it reads its initial instructions from the read-only Boot ROM, which bootstraps the system. The Boot ROM, which also contains the public key of Apple's certificate authority, then verifies that the low-level bootloader (LLB) has been signed by Apple and launches it. The LLB performs a few basic tasks and then verifies the second-stage boot- loader, iBoot. When iBoot launches, the device can either go into recovery mode or boot the kernel. After iBoot verifies the kernel is also signed by Apple, the boot process begins in earnest: drivers are loaded, devices are probed, and system daemons start.

The purpose of this chain of trust is to ensure that all components of the system are written, signed, and distributed by Apple—not by third parties, which could include malicious attackers and authors of software intended to run on jailbroken devices. The chain is also used to bootstrap the signature checking of individual applications; all applications must be directly or indirectly signed by Apple.

Attacking this chain of trust is how jailbreaking works. Jailbreak authors need to find a bug somewhere in this chain to disable the verification of the components further down the chain. Exploits of the Boot ROM are the most desirable because this is the one component Apple can't change in a software update.

Limiting Access with the App Sandbox

Apple's sandbox, historically referred to as Seatbelt, is a *mandatory access control (MAC)* mechanism based on FreeBSD's TrustedBSD framework, pri- marily driven by Robert Watson. It uses a Lisp-like configuration language to describe what resources a program can or cannot access, including files, OS services, network and memory resources, and so on.

MAC is different from traditional access control mechanisms such as discretionary access control (DAC) in that it disallows *subjects*, such as user processes, from manipulating the access controls on *objects* (files, sockets,

and so on). DAC, in its simplest, most common form, is controlled on a UNIX system with *user*, *group*, and *other* permissions, all of which can be granted read, write, or execute permissions.[3] In a DAC system, users can change permissions if they have ownership of an object. For example, if you own a file, you can set it to be world-readable or world-writable, which obviously subverts access controls.

While MAC is a broad term, in sandbox-land it means that applications are shunted into a virtual container that consists of detailed rules specify- ing which system resources a subject is allowed to access, such as network resources, file read and writes, the ability to fork processes, and so on.[4] On OS X you can control some of how your application is sandboxed, but on iOS all third-party applications are run with a single restrictive policy.

In terms of file access, processes are generally confined to their own application bundle directory; they can read and write only the files stored there. The standard policy is slightly porous, however. For example, in some versions of iOS, photos in */private/var/mobile/Media/Photos/* can be directly accessed by third-party applications, despite being outside the application's bundle directory, which allows programs to surreptitiously access photos without asking for user permission. The only protection against applications abusing this type of privilege is Apple's application review process.

This approach differs from that used by Android, which implements a more traditional DAC model, where applications are given their own user ID and a directory owned by that ID. Permissions are managed strictly via traditional UNIX file permissions. While both approaches are workable, MAC generally provides more flexibility. For instance, in addition to app directory segregation, MAC policies can be used to restrict network access or limit what actions system daemons can take.

Data Protection and Full-Disk Encryption

iOS led the way in offering mobile devices with filesystem encryption, for which some credit is due. iOS offers full-disk encryption and additionally provides developers with the Data Protection API to further protect their files. These two related mechanisms make it possible to wipe remote devices and protect user data in the event of device theft or compromise.

Historically, full-disk encryption is made to solve one problem: data at rest being stolen by an attacker. In the laptop or desktop world, this would involve either removing the hard drive from a machine and mounting it on a separate machine or booting into an OS that could read the files off the drive. Filesystem encryption does *not* protect against data being stolen off of a running device. If an application is able to read a file from the disk,

3. This description is, of course, slightly simplified; there are also sticky bits, setuid bits, and so forth. Since iOS doesn't use DAC as its primary access control mechanism, though, I won't get into those topics in this book. 4. You can find a good summary of the default iOS sandbox policies at *https://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf*

filesystem encryption provides no benefit because the kernel transparently decrypts files for any process that tries to read them. In other words, file- system encryption works at a lower level than the calls typically used to read files. An attacker who can authenticate to the system can read any available files unimpeded. iOS devices are generally designed to be running at all times, and their internal storage is not easily removable. If an attacker wanted to read sen- sitive data from a device without authenticating, they would have to com- pletely disassemble the device and hook up the flash storage to a custom interface to read storage directly. There are several far easier methods for obtaining data from the device—including code execution exploits, jailbreaking, and so on—so no one would ever actually go to all that trouble. But that doesn't mean iOS's full filesystem encryption is completely useless. It's necessary to correctly implement two other critical security fea- tures: secure file deletion and remote device wipe. Traditional methods of securely erasing files don't apply to iOS devices, which use solid-state drives (SSDs). The wear-reduction mechanisms used by this hardware remove all guarantees that overwriting a file actually overwrites the previous physical location of the file. The solution to this problem is to ensure that files are encrypted with safely stored keys so that in the event that data destruction is requested, keys can be discarded. The encryption key hierarchy used in iOS is layered. Entire classes of data or even the whole filesystem can be destroyed by throwing away a single encryption key.

The Encryption Key Hierarchy

Filesystem encryption keys for stored data on iOS are hierarchical, with keys encrypting other keys, so that Apple has granular control if and when data is available. The basic hierarchy is shown in Figure 1-1.

Device Key ❶ User Passcode

❷

❸ Filesystem Key Class Key

❹File Key ❺

Figure 1-1: The simplified iOS encryption key hierarchy

The *File Key* is an individual key generated per file and stored in the file's metadata. The *Class Key* is a dedicated key for a particular Data Protection class so that files classified with different protection levels use separate cryptographic keys. In older versions of iOS, the default protec- tion class was `NSFileProtectionNone`; from version 5 onward, the default pro- tection class is `NSFileProtectionCompleteUntilFirstUserAuthentication`, which is further described in Chapter 13. The *Filesystem Key* is a global encryption key used to encrypt the file's security-related metadata after the metadata is encrypted by the Class Key.

The *Device Key* , also known as the UID key, is unique for each device and accessible only by the hardware AES engine, not by the OS itself. This is the master key of the system, as it were, which encrypts the Filesystem Key and the Class Keys. The *User Passcode* , if enabled, is combined with the Device Key when encrypting Class Keys.

When a passcode is set, this key hierarchy also allows developers to spec- ify how they want their locally stored data to be protected, including whether it can be accessed while the device is locked, whether data gets backed up to other devices, and so on. You'll learn more about how to use encryption and file protection features to protect files from device thieves in Chapter 13, where I cover the Data Protection API in greater depth.

The Keychain API

For small pieces of secret information, iOS offers a dedicated Keychain API. This allows developers to store information such as passwords, encryption keys, and sensitive user data in a secure location not accessible to other applications. Calls to the Keychain API are mediated through the `securityd` daemon, which extracts the data from a SQLite data store. The programmer can specify under what circumstances keys should be readable by applica- tions, similar to the Data Protection API.

The Data Protection API

The Data Protection API leverages filesystem encryption, the Keychain, and the user's passcode to provide an additional layer of protection to files at the developer's discretion. This limits the circumstances under which processes on the system can read such files. This API is most commonly used to make data inaccessible when a device is locked.

The degree of data protection in effect depends heavily on the version of iOS the device is running because the default Data Protection classes have changed over time. In newly created iOS application projects, Data Protec- tion is enabled by default for all application data until the user unlocks the device for the first time after boot. Data Protection is enabled in project settings, as shown in Figure 1-2.

Figure 1-2: Adding a data protection entitlement to a project

Native Code Exploit Mitigations: ASLR, XN, and Friends

iOS implements two standard mechanisms to help prevent code execution attacks: *address space layout randomization (ASLR)* and the *XN bit* (which stands for *eXecute Never*). ASLR randomizes the memory location of the program executable, program data, heap, and stack on every execution of the pro- gram; because shared libraries need to stay put to be shared by multiple processes, the addresses of shared libraries are randomized every time the OS boots instead of every program invocation. This makes the specific memory addresses of functions and libraries hard to predict, preventing attacks such as a return-to-libc attack, which relies on knowing the memory addresses of basic libc functions. I'll talk more about these types of attacks and how they work in Chapter 11.

The XN bit, generally known on non-ARM platforms as the NX (No- eXecute) bit, allows the OS to mark segments of memory as nonexecutable, which is enforced by the CPU. In iOS, this bit is applied to a program's stack and heap by default. This means in the event that an attacker is able to insert malicious code onto the stack or heap, they won't be able to redirect the program to execute their attack code. Figure 1-3 shows the segments of process memory and their XN status.

A program can have memory that is both writable and executable only if it's signed with Apple's own code-signing entitlement; this is primarily used for the JavaScript just-in-time (JIT) compiler included as part of Mobile Safari. The regular WebViews that you can use in your own programs don't have access to the same functionality; this is to help prevent code execution attacks. An unfortunate effect of Apple's policy is that it effectively bans third-party JITs, notably preventing Chrome from performing as well as Safari on iOS. Chrome has to use the built-in WebViews.

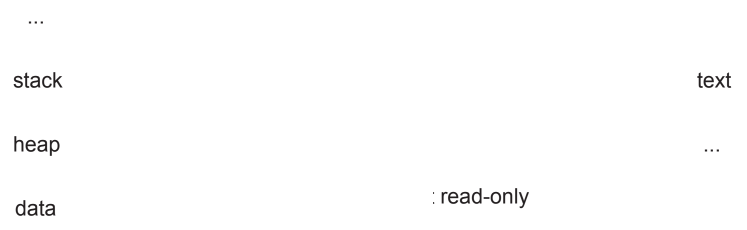

Figure 1-3: Basic memory segments of a process

Jailbreak Detection

Fundamentally, *jailbreaking* is any procedure that disables iOS's code- signing mechanisms, allowing a device to run applications other than those approved directly by Apple. Jailbreaking allows you to take advantage of some useful development and testing tools, as well as utilities that would never pass App Store muster.[5] The ability to jailbreak is critical to testing applications in a black-box fashion; I'll dig in to black-box testing further in Chapter 6.

Contrary to popular belief, jailbreaking doesn't necessarily disable the iOS sandbox. It just allows you to install applications outside of the sandbox. Applications installed in the home directory of the *mobile* user (that is, ones installed via the App Store) are still subject to sandbox restrictions. Third- party iOS applications that need higher levels of privilege on jailbroken devices are installed in the */Applications* folder, alongside the stock Apple applications.

The history of jailbreak detection is long and comical. This procedure is intended to detect whether the device is at heightened risk for compromise because of the less trustworthy nature of unsigned third-party programs. To be fair, there isn't a shortage of malware and misbehaving programs in third- party application repositories, but in general, jailbreak detection isn't worth your time because it won't stop a determined attacker.

For a brief period, Apple had an official jailbreak detection API, but this was pulled rather quickly from subsequent releases of iOS. In the absence of this API, developers have implemented a number of tricks to try detect jail- breaking themselves. The most popular techniques for attempting jailbreak detection go along these lines:

5. It seems, however, that most jailbreak users are motivated by the ability to perform the digital equivalent of putting spinning hubcaps on your car.

• Spawn a new process, such as using `fork()`, `vfork()`, `popen()`, and so on. This is something explicitly prevented by the sandbox. Of course, on jailbroken devices the sandbox is still enabled, making this strategy fairly pointless. It will fail for any App Store application regardless of whether the device is jailbroken.

• Read a file path outside of the sandbox. Developers commonly try to access the binary for `ssh`, `bash`, the *Cydia.app* directory, the path to the `apt` repository that Cydia uses, and so forth. These checks are painfully easy to get around, and tools such as Xcon[6] help end users bypass these checks automatically.

• Ensure that the method name with your jailbreak detection logic is something blatantly obvious, such as `isJailbroken`, allowing attackers to easily spot and disable your jailbreak checks.

There are some more obscure techniques as well. For example, Apple's iBooks application attempts to run unsigned code distributed with the app bundle.[7] Savvy developers will also attempt to use `_dyld_image_count()` and `_dyld_get_image_name()` to check the total number of loaded dynamic libraries (*dylibs*) and their names,[8] and use `_dyld_get_image_header()` to inspect their locations in memory.[9] Getting around these checks generally requires patching an application's binary directly.

As you may have noticed, I don't think much of jailbreak detection in general. Like binary obfuscation techniques and DRM, jailbreak detec- tion techniques typically serve only to make you look foolish when they're bypassed (and believe me, I've seen some foolish obfuscation techniques). Proponents often argue that performing cursory jailbreak detection slows down pirates or attackers. But your adversary's hobby is cracking applica- tions, and they have weeks of time on hand in which to do so—slowing them down by a few hours isn't really worthwhile. The longest it has taken me to develop a jailbreak detection bypass is about a day with an obfuscated binary and a battery of tests—and I'm an amateur at such things.

How Effective Is App Store Review?

When developing an application or assessing the threats that an app faces, it's important to evaluate the risk of a rogue application ending up on end users' devices. Any malicious third-party applications that make it onto devices are able to interact with applications via IPC mechanisms, as well as steal personal information. The primary defense against these applications is Apple's App Store review process.

6. *http://theiphonewiki.com/wiki/XCon* 7. *http://www.cultofmac.com/82097/ibooks-1-2-1-tries-to-run-jailbreak-code-to-detect-jailbroken-iphones/* 8. *http://theiphonewiki.com/wiki/Bypassing_Jailbreak_Detection* 9. *http://stackoverflow.com/questions/4165138/detect-udid-spoofing-on-the-iphone-at-runtime/*

Apple doesn't publicly disclose the techniques it uses to test applications for possible acceptance to the App Store, but it's clear that both binary anal- ysis and dynamic testing are performed. This process has kept most blatant malware out of the App Store, at the cost of also barring any applications that Apple doesn't like the look of (including many types of communication apps, pornography, anything fun, and so on).

But despite Apple's efforts, it has been demonstrated that a moderately sophisticated attacker can get applications through App Store review while retaining the ability to download new code dynamically.

There are a few different ways an attacker can approach this.

Bridging from WebKit There are several approaches to accessing native iOS APIs via JavaScript, such as a user's location or use of media services, using a WebKit-based bridge. PhoneGap is a prominent example of such a package.[10] While these bridges can provide useful functionality and flexibility, using them also means that much application logic ends up in JavaScript and isn't necessarily shipped with the application to begin with. For example, a developer might implement a generic file-opening function that's accessible via JavaScript and avoid using it for anything evil during the review process. But later, that developer can alter the JavaScript served to the device and attempt to read data from areas on the device that they shouldn't be accessing.

I'll discuss the implementation of and some of the issues with JavaScript/ native code bridges in Chapter 9.

Dynamic Patching Normally, applications are prevented from running any native code that hasn't been cryptographically signed by Apple-issued keys. If a bug or mis- feature in Apple's signature-checking logic is found, it can potentially allow for the downloading and execution of native code. A notable example of this in the wild was Charlie Miller's exploitation of a feature that allowed programs to allocate memory regions without NX protection (that is, mem- ory regions that are readable, writable, and executable) and that do not require code to be signed.[11] This mechanism was put in place by Apple to allow Safari's JIT compiler to function,[12] but a bug in the implementation let third-party applications perform the same trick.

10. *http://phonegap.com/* 11. *http://arstechnica.com/apple/2011/11/safari-charlie-discovers-security-flaw-in-ios-gets-booted -from-dev-program/* 12. *http://reverse.put.as/wp-content/uploads/2011/06/syscan11_breaking_ios_code_signing.pdf*

This meant that native code could be downloaded and executed with- out needing to be signed at all. Miller demonstrated this by submitting an application, called *InstaStock*, to the App Store which purported to be a stock ticker checking program. At the time of app review, the app did nothing malicious or unusual; however, after the review process was complete, Miller was able to instruct the program to download new, unsigned code and exe- cute that code without problem. This issue is now resolved, but it does give you an idea of the things that can slip through the cracks of review.

Intentionally Vulnerable Code An interesting approach to bypassing App Store review is to intentionally make your app vulnerable to remote exploits. *Jekyll*[13] was a proof-of-concept application developed at Georgia Tech that intentionally introduced a buffer overflow in the core application. Malicious code was included in the app itself so that the code would be signed but was never called by the appli- cation. After approval, the researchers were able to use a buffer overflow exploit to change the control flow of the application to include malicious code, allowing it to use private Apple frameworks to interact with Bluetooth, SMS, and more.

Embedded Interpreters While Apple's policy on this practice has shifted over the years, many prod- ucts (primarily games) use an embedded Lua interpreter to perform much of the internal logic. Malicious behavior using an embedded interpreter has not yet been reported in the wild, but a crafty application using a similar interpreter could download code dynamically and execute it from memory, though not during the review process, of course. This would add new and malicious (or helpful, if you're so inclined) functionality.

Closing Thoughts

Ultimately, what protections does application review provide? Well, it does weed out less sophisticated malware. But you can assume with some cer- tainty that malicious applications will indeed slip through from time to time. Keep that in mind and code your applications defensively; you definitely *cannot* assume other applications on the OS are benign.

13. *http://www.cc.gatech.edu/~klu38/publications/security13.pdf*

2

OBJECTIVE-C FOR THE LAZY

Objective-C has been met with both derision and adulation during its illustrious career. Brought to popularity by NeXTStep and inspired by the design of Smalltalk, Objective-C is a superset of C. Its most notable characteristics are the use of infix notation and absurdly long class names. People tend to either love it or hate it. People who hate it are wrong.

In this chapter, I'll go over the basics of Objective-C, assuming that you're already familiar with programming in some language or another. Know, however, that Cocoa and Objective-C are constantly changing. I can't cover all of their finer details adequately in a single chapter, but I do include some hints here to help nondevelopers get their bearings when examining Objective-C code. If you're starting from very little programming knowledge, you may wish to check out a book like Knaster, Malik, and Dalrymple's *Learn Objective-C on the Mac: For OS X and iOS* (Apress, 2012) before you dig in.

As much as I'd like to stick with the most modern coding patterns of Objective-C, if you're auditing existing code, you may come across plenty of crusty, reused code from the early days of iOS. So just in case, I'll go over both historical Objective-C constructs and the newly sanctioned versions.

Key iOS Programming Terminology

There are a few terms you'll want to be familiar with to understand where Apple's various APIs come from. *Cocoa* is the general term for the frame- works and APIs that are used in Objective-C GUI programming. *Cocoa Touch* is a superset of Cocoa, with some added mobile-related APIs such as dealing with gestures and mobile GUI elements. *Foundation* classes are Objective-C classes that make up much of what we call the Cocoa API. *Core Foundation* is a lower-level C-based library upon which many Foundation classes are based, usually prefixed with `CF` instead of `NS`.

Passing Messages

The first key to grokking Objective-C is understanding that the language is designed around the concept of *message passing*, rather than *calling*. It's use- ful (for me, anyway) to think of Objective-C as a language where objects sit around shouting at each other in a crowded room, rather than a language where hierarchical directors give orders to their subordinates. This analogy especially makes sense in the context of delegates, which I'll get to shortly.

At its most basic, sending Objective-C messages looks like this:

```
[Object doThisThingWithValue:myValue];
```

That's like saying, "Hey there, `Object`! Please do this thing using a value of `myValue`." When passing in multiple parameters, the nature of the first one is conventionally indicated by the message name. Any subsequent parame- ters must be both defined as part of the class and specifically named when called, as in this example:

```
if (pantsColor == @"Black") {

    [NSHouseCat sleepOnPerson:person
withRegion:[person lap] andShedding:YES
             retries:INT_MAX]; }
```

In this simplified simulation of catnapping under certain conditions, `sleepOnPerson` specifies a place to sleep (`person`), and `withRegion` specifies the region of the person to sleep on by sending `person` a message returning that person's `lap`. The `andShedding` parameter accepts a Boolean, and `retries` speci- fies the number of times this action will be attempted—in this case, up to the maximum value of an integer on a platform, which will vary depending on whether you have a 64-bit cat.

If you've been writing Objective-C for a while, you may notice that the formatting of this code looks different than what you're used to. That's because this is an arcane method of formatting Objective-C code, known

as "the correct way," with vertically aligned colons between argument names and values. This keeps the pairings between parameter names and values visually obvious.

Dissecting an Objective-C Program

The two main parts of an Objective-C program are the *interface* and the *implementation*, stored in *.h* and *.m* files, respectively. (These are roughly analogous in purpose to *.h* and *.cpp* files in C++.) The former defines all of the classes and methods, while the latter defines the actual meat and logic of your program.

Declaring an Interface

Interfaces contain three main components: instance variables (or *ivars*), class methods, and instance methods. Listing 2-1 is the classic (that is, depre- cated) Objective-C 1.0 way to declare your interfaces.

```
@interface Classname : NSParentClass {  NSSomeType aThing;
int anotherThing; }  + (type)classMethod:(vartype)myVariable;
   - (type)instanceMethod:(vartype)myVariable;
      @end
```

Listing 2-1: Declaring an interface, archaic version

Inside the main `@interface` block at , instance variables are declared with a class (like `NSSomeType`) or a type (like `int`), followed by their name. In Objective-C, a `+` denotes the declaration of a class method , while a `-` indi- cates an instance method . As with C, the return type of a method is speci- fied in parentheses at the beginning of the definition.

Of course, the modern way of declaring interfaces in Objective-C is a little different. Listing 2-2 shows an example.

```
@interface Kitty : NSObject {

  @private NSString *name; @private NSURL *homepage;

  @public NSString *color; }@property NSString *name;

     @property NSURL *homepage;  @property(readonly)

                                  NSString *color;
```

```
+ (type)classMethod:(vartype)myVariable; -
         (type)instanceMethod:(vartype)myVariable;
```

Listing 2-2: Declaring an interface, modern version

This new class, called `Kitty`, inherits from `NSObject` . `Kitty` has three instance variables of different accessibility types, and three properties are declared to match those instance variables. Notice that `color` is declared `readonly` ; that's because a `Kitty` object's color should never change. This means when the property is synthesized, only a getter method will be cre- ated, instead of both a getter and a setter. `Kitty` also has a pair of methods: one class method and one instance method.

You may have noticed that the example interface declaration used the `@private` and `@public` keywords when declaring instance variables. Similar to other languages, these keywords define whether ivars will be accessible from within only the class that declared it (`@private`), accessible from within the declaring class and any subclasses (`@protected`), or accessible by any class (`@public`). The default behavior of ivars is `@protected`.

NOTE *Newcomers to the language often want to know whether there is an equivalent to private methods. Strictly speaking, there isn't a concept of private methods in Objective- C. However, you can have the functional equivalent by declaring your methods only in the* `@implementation` *block instead of declaring them in both the* `@interface` *and the* `@implementation`*.*

Inside an Implementation File

Just like *.c* or *.cpp* files, Objective-C implementation files contain the meat of an Objective-C application. By convention, Objective-C files use *.m* files, while Objective-C++ files (which mix C++ and Objective-C code) are stored in *.mm* files. Listing 2-3 breaks down the implementation file for the `Kitty` interface in Listing 2-2.

```
@implementation Kitty  @synthesize name;
       @synthesize color; @synthesize
       homepage;

       + (type)classMethod:(vartype)myVariable {

// method logic }- (type)instanceMethod:(vartype)myVariable {
    // method logic }@end
```

```
Kitty *myKitty = [[Kitty alloc] init];

 [myKitty setName:@"Ken"];  myKitty.homepage = [[NSURL alloc]
initWithString:@"http://me.ow"];
```

Listing 2-3: A sample implementation

The @synthesize statements at create the setter and getter methods for the properties. Later, these getter and setter methods can be used either with Objective-C's traditional infix notation , where methods of the format *propertyName* and *setPropertyName* (like name and setName, respectively) get and set values, or with dot notation , where properties like homepage are set or read using the *.property* format, as they might be in other languages.

NOTE *Be careful with dot notation, or just don't use it. Dot notation makes it hard to know whether you're dealing with an object or a C struct, and you can actually call* any *method with it—not only getters and setters. Dot notation is also just visually inconsis- tent. Long story short, in this book I'll avoid dot notation in the name of consistency and ideological purity. But despite my best efforts, you'll likely encounter it in the real world anyway.*

Technically, you don't need to synthesize properties that are declared in the interface file with @property, like name, color, and homepage in Listing 2-3; the compiler in recent versions of Xcode synthesizes these properties on its own. But you may want to manually declare them anyway for clarity or when you want to change the name of the instance variable to differentiate it from the property name. Here's how manually synthesizing a property works:

```
@synthesize name = thisCatName;
```

Here, the property name is backed by the instance variable thisCatName because it was manually synthesized. However, the default behavior with automatic property synthesis is analogous to this:

```
@synthesize name = _name;
```

This default behavior prevents developers from accidentally meddling with the instance variables directly, instead of using setters and getters, which can cause confusion. For example, if you set an ivar directly, you'll be bypassing any logic in your setter/getter methods. Automatic synthesis is probably the best way to do things, but you'll be seeing manual synthesis in code for a long time to come, so it's best to be aware of it.

Specifying Callbacks with Blocks

One thing that's becoming increasingly popular in Objective-C code is the use of *blocks*, which are often used in Cocoa as a way to specify a callback. For example, here's how you'd use the `dataTaskWithRequest` method of the `NSURLSessionDataTask` class:

```
      NSURLSession *session = [NSURLSession sessionWithConfiguration:configuration
delegate:self delegateQueue:nil];

      NSURLSessionDataTask *task = [session dataTaskWithRequest:request
                                 completionHandler:  ^(NSData *data,
   NSURLResponse *response, NSError *error) {
 NSLog(@"Error: %@ %@", error, [error userInfo]); }];
```

The `^` at is declaring a block that will be executed once the request is complete. Note that no name is specified for this function because it won't be called from anywhere other than this bit of code. A block declaration just needs to specify the parameters that the closure will take. From there, the rest of the block is just like a normal function. You can use blocks for tons of other things as well, but to start with, it's probably sufficient to have a basic understanding of what they are: things that begin with `^` and do stuff.

How Objective-C Manages Memory

Unlike some other languages, Objective-C does not have any garbage collec- tion. Historically, Objective-C has used a *reference counting model*, using the `retain` and `release` directives to indicate when an object needs to be freed, to avoid memory leaks. When you `retain` an object, you increase the *reference count*—that is, the number of things that want that object to be available to them. When a piece of code no longer needs an object, it sends it a `release` method. When the reference count reaches zero, the object is deallocated, as in this example:

```
   NSFish *fish = [[NSFish alloc] init]; NSString
  *fishName = [fish name];   [fish release];
```

Assume that before this code runs, the reference count is 0. After , the reference count is 1. At , the `release` method is called to say that the `fish` object is no longer needed (the application just needs the `fish` object's `name` property), and when `fish` is released, the reference count should be 0 again.

The `[[Classname alloc] init]` can also be shortened to `[Classname new]`, but the `new` method isn't favored by the Objective-C community because it's less explicit and is inconsistent with methods of object creation other than `init`. For example, you can initialize `NSString` objects with `[[NSString alloc] initWithString:@"My string"]` , but there's no equivalent `new` syntax, so your code would end up having a mix of both methods. Not everyone is averse to `new`, and it's really a matter of taste, so you’re likely to see it both ways. But in this book, I’ll favor the traditional approach.

Regardless of which allocation syntax you prefer, the problem with a manual retain/release is that it introduced the possibility of errors: program- mers could accidentally release objects that had already been deallocated (causing a crash) or forget to release objects (causing a memory leak). Apple attempted to simplify the situation with automatic reference counting.

Automatic Reference Counting

Automatic reference counting (ARC) is the modern method of Objective-C memory management. It removes the need for manually tracking reference counts by automatically incrementing and decrementing the retain count where appropriate.[1] Essentially, it inserts `retain` and `release` methods for you. ARC introduces a few new concepts, listed here:

• *Weak* and *strong* references assist in preventing cyclical references (referred to as *strong reference cycles*), where a parent object and child object both have ownership over each other and never get deallocated.

• Object ownership between Core Foundation objects and Cocoa objects can be bridged. Bridging tells the compiler that Core Foundation objects that are cast to Cocoa objects are to be managed by ARC, by using the `__bridge` family of keywords.

• `@autoreleasepool` replaces the previously used `NSAutoReleasePool` mechanism.

In modern Cocoa applications with ARC, the details of memory man- agement are unlikely to come into play in a security context. Previously exploitable conditions such as double-releases are no longer a problem, and memory-management-related crashes are rare. It’s still worth noting that there are other ways to cause memory management problems because `CFRetain` and `CFRelease` still exist for Core Foundation objects and C `malloc` and `free` can still be used. I’ll discuss potential memory management issues using these lower-level APIs in Chapter 11.

1. *http://developer.apple.com/library/mac/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html*

Delegates and Protocols

Remember how objects “shout at each other in a crowded room” to pass messages? *Delegation* is a feature that illustrates Objective-C’s message- passing architecture particularly well. Delegates are objects that can receive messages sent during program execution and respond with instructions that influence the program’s behavior.

To be a delegate, an object must implement some or all meth- ods defined by a *delegate protocol*, which is an agreed-upon method of communication between a delegator and a delegate. You can declare your own protocols, but most commonly you’ll be using established protocols in the core APIs.

The delegates you’ll write will typically respond to one of three funda- mental message types: *should*, *will*, and *did*. Invoke these messages whenever an event is about to happen and then let your delegates direct your program to the correct course of action.

Should Messages Objects pass *should* messages to request input from any available delegates on whether letting an event happen is a good idea. Think of this as the final call for objections. For example, when a `shouldSaveApplicationState` message is invoked, if you've implemented a delegate to handle this message, your delegate can perform some logic and say something like, "No, actually, we shouldn't save the application state because the user has checked a checkbox saying not to." These messages generally expect a Boolean as a response.

Will Messages A *will* message gives you the chance to perform some action before an event occurs—and, sometimes, to put the brakes on before it does. This message type is more like saying, "Hey guys! Just an FYI, but I'm going to go do this thing, unless you need to do something else first. I'm pretty committed to the idea, but if it's a total deal-breaker, let me know and I can stop." An example would be the `applicationWillTerminate` message.

Did Messages A *did* message indicates that something has been decided for sure and an event is going to happen whether you like it or not. It also indicates that if any delegates want to do some stuff as a result, they should go right ahead. An example would be `applicationDidEnterBackground`. In this case, did isn't really an indication that the application *has* entered the background, but it's a reflection of the decision being definitively made.

Declaring and Conforming to Protocols To declare that your class conforms to a protocol, specify that protocol in your `@interface` declaration within angle brackets. To see this in action, look at Listing 2-4, which shows an example `@interface` declaration that uses the NSCoding protocol. This protocol simply specifies that a class implements two methods used to encode or decode data: `encodeWithCoder` to encode data and `initWithCoder` to decode data.

```
@interface Kitty : NSObject <NSCoding> {

        @private NSString *name; @private NSURL

            *homepage; @public NSString *color;

                         }@implementation Kitty

- (id)initWithCoder:(NSCoder *)decoder {

     self = [super init]; if (!self) {

return nil; }[self setName:[decoder decodeObjectForKey:@"name"]]; [self
     setHomepage:[decoder decodeObjectForKey:@"homepage"]]; [self
     setColor:[decoder decodeObjectForKey:@"color"]];
return self; }

     - (void)encodeWithCoder:(NSCoder *)encoder {
        [encoder encodeObject:[self name] forKey:@"name"]; [encoder encodeObject:[self
        author] forKey:@"homepage"]; [encoder encodeObject:[self pageCount]
        forKey:@"color"]; }
```

Listing 2-4: Declaring and implementing conformance to the NSCoding protocol

The declaration at specifies that the `Kitty` class will be conforming to the NSCoding protocol.[2] When a class declares a protocol, however, it must also conform to it, which is why `Kitty` implements the required `initWithCoder` and `encodeWithCoder` methods. These particular methods are used to serialize and deserialize objects.

2. *https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSCoding_Protocol/Reference/Reference.html*

If none of the built-in message protocols do what you need, then you can also define your own protocols. Check out the declaration of the NSCod- ing protocol in Apple's Framework header files (Listing 2-5) to see what a protocol definition looks like.

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder; -

(id)initWithCoder:(NSCoder *)aDecoder;

@end
```

Listing 2-5: The declaration of the NSCoding protocol, from Frameworks/NSCoding.h

Notice that the NSCoding definition contains two methods that any class conforming to this protocol must implement: `encodeWithCoder` and `initWithCoder`. When you define a protocol, you must specify those methods yourself.

The Dangers of Categories

Objective-C's *category* mechanism allows you to implement new methods on existing classes at runtime, without having to recompile those classes. Categories can add or replace methods in the affected class, and they can appear anywhere in the codebase. It's an easy way to quickly change the behavior of a class without having to reimplement it.

Unfortunately, using categories is also an easy way to make egregious security mistakes. Because they can affect your classes from anywhere within the codebase—even if they appear only in third-party code—critical func- tionality, such as TLS endpoint validation, can be completely overridden by a random third-party library or a careless developer. I've seen this happen in important iOS products before: after carefully verifying that TLS/SSL works correctly in their application, developers include a third-party library that overrides that behavior, messing up their own properly designed code.

You can usually spot categories by noting `@implementation` directives that purport to implement classes already present in Cocoa Touch. If a developer was actually creating a category there, then the name of the category would follow the `@implementation` directive in parentheses (see List- ing 2-6).

```
@implementation NSURL (CategoryName)

- (BOOL) isPurple; {
   if ([self isColor:@"purple"])
      return YES;
```

```
elsereturn NO; }@end
```

Listing 2-6: Implementing a category method

You can also use categories to override *existing* class methods, which is a potentially useful but particularly dangerous approach. This can cause secu- rity mechanisms to be disabled (such as the aforementioned TLS validation) and can also result in unpredictable behavior. Quoth Apple:

> If the name of a method declared in a category is the same as a method in the original class, or a method in another category on the same class (or even a superclass), the behavior is undefined as to which method implementation is used at runtime.

In other words, multiple categories can define or overwrite the same method, but only one will "win" and be called. Note that some Framework methods may themselves be implemented via a category—if you attempt to override them, your category *might* be called, but it might not.

A category may also accidentally override the functionality of subclasses, even when you only meant for it to add a new method. For example, if you were to define an `isPurple` method on `NSObject`, all subclasses of `NSObject` (which is to say, all Cocoa objects) would inherit this method. Any other class that defined a method with the same name might or might not have its method implementation clobbered. So, yes, categories are handy, but use them sparingly; they can cause serious confusion as well as security side effects.

Method Swizzling

Method swizzling is a mechanism by which you can replace the implemen- tation of a class or instance method that you don't own (that is, a method provided by the Cocoa API itself). Method swizzling can be functionally similar to categories or subclassing, but it gives you some extra power and flexibility by actually swapping the implementation of a method with a totally new implementation, rather than extending it. Developers typically use this technique to augment functionality of a method that's used by many different subclasses so they don't have to duplicate code.

The code in Listing 2-7 uses method swizzling to add a logging state- ment to any call of `setHidden`. This will affect any subclass of `UIView`, including `UITextView`, `UITextField`, and so forth.

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

@implementation UIView(Loghiding)

- (BOOL)swizzled_setHidden {
    NSLog(@"We're calling setHidden now!");

BOOL result = [self swizzled_setHidden];

return result; }

+ (void)load {
    Method original_setHidden; Method
    swizzled_setHidden;

    original_setHidden = class_getInstanceMethod(self, @selector(setHidden));
    swizzled_setHidden = class_getInstanceMethod(self, @selector(swizzled_
setHidden)); method_exchangeImplementations(original_setHidden, swizzled_setHidden); }@end
```

Listing 2-7: Exchanging the implementation of an existing method and a replacement methodAt , a wrapper

method is defined that simply spits out an `SLog` that the `setHidden` method is being called. But at , the `swizzle_SetHidden` method appears to be calling itself. That's because it's considered a best practice to call the original method after performing any added functionality, to prevent unpredictable behavior like failing to return the type of value the caller would expect. When you call `swizzled_setHidden` from within itself, it actually calls the *original* method because the original method and the replacement method have already been swapped.

The actual swapping is done in the `load` class method , which is called by the Objective-C runtime when loading the class for the first time. After the references to the original and swizzled methods are obtained, the `method_exchangeImplementations` method is called at , which, as the name implies, swaps the original implementation for the swizzled one.

There are a few different strategies for implementing method swizzling, but most of them carry some risk since you're mucking around with core functionality.

If you or a loved one want to implement method swizzling, you may want to consider using a fairly well-tested wrapper package, such as JRSwizzle.[3] Apple may reject applications that appear to use method swizzling in a dangerous way.

Closing Thoughts

Overall, Objective-C and the Cocoa API are nicely high-level and prevent a number of classic security issues in C. While there are still several ways to mess up memory management and object manipulation, most of these methods result in a denial of service at worst in modern code. If you're a developer, rely on Cocoa as much as possible, rather than patching in C or C++ code.

Objective-C does, however, contain some mechanisms, such as cate- gories or swizzling, that can cause unexpected behavior, and these mech- anisms can affect your codebase widely. Be sure to investigate these tech- niques when you see them during an app assessment because they can potentially cause some serious security misbehavior.

3. *https://github.com/rentzsch/jrswizzle/*

3

IOS APPLICATION ANATOMY

To understand some of the problems iOS applications face, it's useful to get an idea of how different types of data are stored and manipulated within an appli- cation's private directory, where all of its configura- tion, assets, binaries, and documents are stored. This is where you can discover all manner of information leakage, as well as dig in to the guts of the program that you're examining.

The quickest way find out what data your application stores locally on an iOS device is to check out *~Library/Developer/CoreSimulator/Devices*. Starting with Xcode 6, each combination of device type and OS ver- sion you've ever deployed into the Simulator application is assigned a UUID. Your particular application's data will be stored in two places under this directory. Your application binary and assets, including *.nib* user interface files and graphic files included with the application, are in *<device ID>/data/Containers/Bundle/Application/<app bundle id>*. The more dynamic data that your application stores is in ~*<device ID>/ data/Containers/Data/Application/<app bundle id>*. Systemwide data such as global configurations will be stored in the remainder of the *<device ID>* directory.

Exploring this directory structure, which is sketched out in simplified form in Figure 3-1, also reveals which types of data are handled by OS ser- vices rather than your application.

<app bundle ID>AppName.appen.lproj

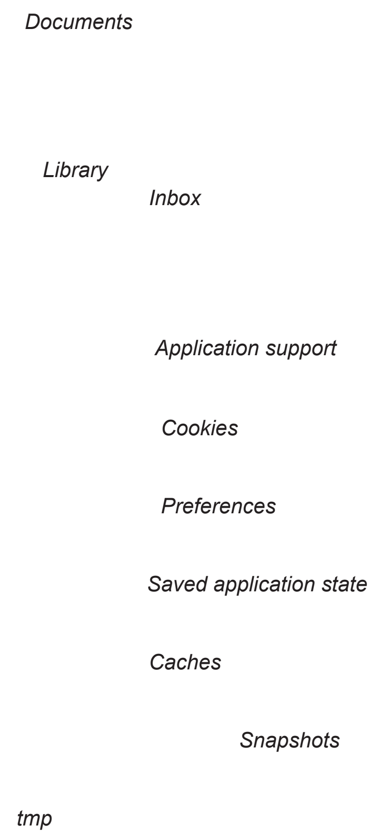

Figure 3-1: Layout of an application directory

If you're on a jailbroken device, you can use SSH to connect to the device and explore the directory structure; I'll talk about jailbreaking and connecting to test devices in Chapter 6. Whether or not your device is jailbroken, you can use a tool such as iExplorer[1] to examine the directory structure of your installed applications, as shown in Figure 3-2.

In the rest of this chapter, I'll cover some of the common directories and data stores used by iOS applications, as well as how to interact with them programmatically and what data can leak from them.

1. *http://www.macroplant.com/iexplorer/*

Figure 3-2: Examining an application bundle with iExplorer

Dealing with plist Files

Before you start examining the directory structure, you need to know how to read some of the stuff you'll find there. iOS stores app configura- tion data inside *property list (plist)* files, which hold this information in Core Foundation data types such as `CFArray`, `CFString`, and so forth. From a security standpoint, you want to examine plists for things that shouldn't be stored in plaintext, such as credentials, and then potentially manipulate them to change the application's behavior. For instance, you could enable a paid feature that's disabled.

There are two types of plist formats: binary and XML. As you can see in the following example, the XML format is easily readable by humans.

```
<?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> <plist version="1.0"> <dict>
  <key>CFBundleDevelopmentRegion</key>
  <string>en</string> <key>CFBundleExecutable</key>
  <string>Test</string> <key>CFBundleIdentifier</key>
  <string>com.dthiel.Test</string>
  <key>CFBundleInfoDictionaryVersion</key>
```

```
  <string>6.0</string> <key>CFBundleName</key>
  <string>Test</string>
  <key>CFBundlePackageType</key>
  <string>APPL</string>
  <key>CFBundleShortVersionString</key>
  <string>1.0</string> <key>CFBundleSignature</key>
  <string>????</string>
  <key>CFBundleSupportedPlatforms</key> <array>
<string>iPhoneSimulator</string> </array> --snip--
```

This is simply a dictionary containing hierarchical keys and values, which provides information about the app—the platforms it can run on, the code signature, and so forth (the signature is not present here because the app is deployed in the Simulator application).

But when examining files from the command line or working with plists programmatically, you'll frequently encounter plists in binary format, which is not particularly human readable (or writable). You can convert these plists to XML using the `plutil(1)` command.

```
$ plutil -convert xml1 Info.plist -o - $ plutil -convert xml1
Info.plist -o Info-xml.plist $ plutil -convert binary1
Info-xml.plist -o Info-bin.plist
```

The first command converts a binary plist to XML and outputs it to stdout, where you can pipe it to `less(1)` or similar commands. You can also output directly to a file with `-o` *`filename`*, as in the second command. In the third command, the `binary1` conversion type turns an XML-formatted plist to binary; but since the formats are interchangeable, you shouldn't really need to do this.

To make reading and editing plists more seamless, you can also config- ure your text editor to automatically convert plist files so that if you need to read or write to one, you can do so smoothly from a familiar environment. For example, if you happen to use Vim, you might add a configuration like this to your *.vimrc* file:

```
" Some quick bindings to edit binary plists command -bar PlistXML :set binary | :1,$!plutil
          -convert xml1 /dev/stdin -o - command -bar Plistbin :1,$!plutil -convert binary1
          /dev/stdin -o -

          fun ReadPlist()
             if getline("'[") =~ "^bplist"
                :PlistXML
```

Figure 3-3: Viewing a plist within Xcode

See the man pages `plist(5)` and `plutil(1)` for more information about viewing and editing plists. If you’re working on a jailbroken device, you can use the `plutil` command included with Erica Sadun’s Erica Utilities[2] (available in Cydia) to work with plists locally.

2. Erica Utilities has a number of other useful tools for working with jailbroken devices; you can check out the list at *http://ericasadun.com/ftp/EricaUtilities/*.

```
set filetype=xml endif endfunction

       augroup misc

au BufWinEnter *.plist, call ReadPlist() augroup end
```

This configuration will use the `:PlistXML` command to automatically convert any binary plist that you edit to XML format, allowing you to make changes in a human-readable format. Before actually writing those changes to the file, the configuration will convert the file to binary again using the `:Plistbin` command. Note that the file will still be successfully consumed by applications regardless of whether it is in binary or XML format.

You can view plists of either format within Xcode, as in Figure 3-3. The advantage of using Xcode is that you'll have some additional help and drop- down menus that show you what potential values you might be able to use for the various keys. It's good to know how to work with plists from the command line, though, because this lets you directly interact with them via SSH sessions to jailbroken devices.

Device Directories

Starting with iOS 8, Simulator platforms such as iPhone, iPad, and their variations are stored in directories named with unique identifiers. These identifiers correspond with the type of device you choose when launching the Simulator from Xcode, in combination with the requested OS version. Each of these directories has a plist file that describes the device. Here's an example:

```
<?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>UDID</key> <string>DF15DA82-1B06-422F-860D-84DCB6165D3C</string>
<key>deviceType</key>
<string>com.apple.CoreSimulator.SimDeviceType.iPad-2</string> <key>name</key>
<string>iPad 2</string> <key>runtime</key>
<string>com.apple.CoreSimulator.SimRuntime.iOS-8-0</string> <key>state</key>
<integer>3</integer> </dict> </plist>
```

In this plist file, it's not immediately obvious which directory is for which device. To figure that out, either you can look at the *.default_created.plist* file in the *Devices* directory, or you can just grep all of the *device.plist* files, as shown in Listing 3-1.

```
$ cd /Users/me/Library/Developer/CoreSimulator/Devices && ls
    26E45178-F483-4CDD-A619-9C0780293DD4
    78CAAF2B-4C54-4519-A888-0DB84A883723
    A2CD467D-E110-4E38-A4D9-5C082618604A
    AD45A031-2412-4E83-9613-8944F8BFCE42
    676931A8-FDA5-4BDC-85CC-FB9E1B5368B6
    989328FA-57FA-430C-A71E-BE0ACF278786
    AA9B1492-ADFE-4375-98F1-7DB53FF1EC44
    DF15DA82-1B06-422F-860D-84DCB6165D3C


$ for dir in `ls|grep -v default` doecho $dir grep -C1 name $dir/device.plist |tail

    -1|sed -e 's/<\/*string>//g' done
```

```
26E45178-F483-4CDD-A619-9C0780293DD4
  iPhone 5s 676931A8-FDA5-4BDC-85CC-FB9E1B5368B6
   iPhone 5 78CAAF2B-4C54-4519-A888-0DB84A883723
   iPad Air 989328FA-57FA-430C-A71E-BE0ACF278786
  iPhone 4s A2CD467D-E110-4E38-A4D9-5C082618604A
 iPad Retina AA9B1492-ADFE-4375-98F1-7DB53FF1EC44
                 Resizable iPad
       AD45A031-2412-4E83-9613-8944F8BFCE42
                Resizable iPhone
       DF15DA82-1B06-422F-860D-84DCB6165D3C
      iPad 2
```

Listing 3-1: Grepping to determine which identifier maps to which model of iOS device

After entering the appropriate directory for the device you've been testing your application on, you'll see a *data* directory that contains all of the Simulator files, including those specific to your application. Your application data is split into three main directories under *data/Containers*: *Bundle*, *Data*, and *Shared*.

The Bundle Directory

The *Bundle* directory contains an *Applications* directory, which in turn con- tains a directory for each of the applications stored on the device, repre- sented by that application's bundle ID. In each application's directory, the *.app* folder is where the application's core binary is stored, along with image assets, localization information, and the *Info.plist* file that contains the core configuration information for your application. *Info.plist* includes the bundle identifier and main executable, along with information about your application's UI and which device capabilities an application requires to be able to run.

On the filesystem, these plists are stored in either XML or binary format, with the latter being the default. You can retrieve the informa- tion in *Info.plist* programmatically by referencing dictionary attributes of `[NSBundle mainBundle]`;[3] this is commonly used for loading styling or localiza- tion information.

3. *https://developer.apple.com/library/Mac/documentation/Cocoa/Reference/Foundation/Classes/NSBundle_Class/Reference/Reference.html*

One thing that will potentially be of interest in the *Info.plist* file is the `UIRequiredDeviceCapabilities` entry, which looks something like this:

```
<key>UIRequiredDeviceCapabilities</key> <dict>
```

```
<key>armv7</key> <true/> <key>location-services</key> <true/> <key>sms</key> <true/>
</dict>
```

The `UIRequiredDeviceCapabilities` entry describes which system resources an app requires. While not an enforcement mechanism, this can give you some clues as to what type of activities the application will engage in.

The Data Directory

The primary area of interest in the *Data* directory is the *Applications* sub- directory. The *Data/Applications* directory contains the rest of the data an application uses to run: preferences, caches, cookies, and so on. This is also the primary location you'll want to inspect for most types of data leakage. Now, let's go over the various subdirectories and the types of data that they may end up holding.[4]

The Documents and Inbox Directories

The *Documents* directory is intended to store your nontransient application data, such as user-created content or local information allowing the app to run in offline mode. If `UIFileSharingEnabled` is set in your application's *Info.plist* file, files here will be accessible via iTunes.

Data files that other applications ask your app to open are stored in your application's *Documents/Inbox* directory. These will be invoked by the calling application using the `UIDocumentInteractionController` class.[5]

You can only read or delete files stored in the *Inbox* directory. These files come from another application that can't write to your app directory, so they're put there by a higher-privileged system process. You may want to consider deleting these files periodically or giving the user the option to delete them because it will not be apparent to the user what documents are stored here and whether they contain sensitive information.

4. Note that not all directories that can exist in this directory tree will exist for every application; some are created on the fly only when certain APIs are used by the app. 5. *http://developer.apple.com/library/ios/#documentation/FileManagement/Conceptual/ DocumentInteraction_TopicsForIOS*

If you're writing an application with the goal of ensuring sensitive infor- mation doesn't remain on disk, copy documents out of the *Inbox* directory to a separate location where you can apply Data Protection and then remove those files from the *Inbox* directory.

It's also worth remembering that under certain circumstances, any file your application asks to open may persist on the disk *forever*. If you attempt to open a file type that your program isn't a handler for, then that file will be passed off to a third-party app, and who knows when the other app will delete it? It may get stored indefinitely. In other words, the cleanup of any file that you ask another app to open is beyond your control, even if you simply preview the contents using the Quick Look API. If having *Inbox* files kick around for a long time is problematic, consider giving your application the ability to view such data on its own (rather than relying on a helper) and then make sure to dispose of the files properly.

The Library Directory The *Library* directory contains the majority of your application's files, includ- ing data cached by the application or by particular networking constructs. It will be backed up via iTunes and to iCloud, with the exception of the *Caches* directory.

The Application Support Directory The *Application Support* directory is not for storing files created or received by the user but rather for storing additional data files that will be used by your application. Examples would be additional purchased downloadable content, configuration files, high scores, and so on—as the name implies, things that support the running and operation of the application. Either these files can be deployed when the application is first installed or they can be downloaded or created by your application later.

By default, iTunes backs up the data in this directory to your com- puter and to iCloud. However, if you have privacy or security concerns about this data being stored in Apple's cloud environment, you can explic- itly disallow this by setting the `NSURLIsExcludedFromBackupKey` attribute on newly created files. I'll discuss preventing data from syncing to iCloud further in Chapter 10.

Note that Apple requires that applications back up only user data to iCloud (including documents they've created, configuration files, and so forth), never application data. Applications that allow application con- tent, such as downloadable app content, to be backed up to iCloud can be rejected from the App Store.

The Caches and Snapshots Directories The *Caches* directory is similar in function to a web browser's cache: it's intended for data that your application will keep around for performance reasons but not for data that is crucial for the application to function. As such, this directory won't be backed up by iTunes.

While Apple states that your application is responsible for managing the *Caches* directory, the OS does actually manipulate the directory's con- tents and that of its subfolder, *Snapshots*. Always consider the contents of the *Caches* directory to be transient, and expect it to disappear between program launches. iOS will cull these cache directories automatically if the system starts running low on space, though it won't do this while the application is running.

The *Caches* directory also sometimes stores web cache content in a subdirectory such as *Caches/com.mycompany.myapp*. This is one place where sensitive data can leak because iOS can cache information delivered over HTTPS for quite a long time. If the developer hasn't made special effort to prevent data from being cached or to expire cached data quickly, you can often find some goodies in here.

Finally, when an application is put into the background, the OS also automatically stores screenshots of the application in the *Snapshots* sub- directory, potentially leaving sensitive information on local storage. This is done for one reason: so that the OS can use the current screen state to create the "whooshing" animation that happens when you bring an appli- cation to the foreground. Unfortunately, a side effect I frequently see in iOS applications is that the disk stores images of people's Social Security numbers, user details, and so on. I'll discuss mitigation strategies for this (and many other caching problems) in Chapter 10.

The Cookies Directory The *Cookies* directory stores cookies set by the URL loading system. When you make an `NSURLRequest`, any cookies will be set according to either the default system cookie policy or one that you've specified. Unlike on OS X, cookies on iOS are not shared between applications; each application will have its own cookie store in this directory.

The Preferences Directory iOS stores application preferences in the *Preferences* directory, but it doesn't allow applications to write directly to the files there. Instead, files in this directory are created, read, and manipulated by either the `NSUserDefaults` or `CFPreferences` API.

These APIs store application preference files in plaintext; therefore, you most definitely should *not* use them to store sensitive user information or credentials. When examining an application to see what information it's storing locally, be sure to examine the plist files in the *Preferences* directory. You'll sometimes find usernames and passwords, API access keys, or security controls that are not meant to be changed by users.

The Saved Application State Directory Users expect apps to remember what they enter into text fields, which set- tings they've enabled, and so on. If a user switches to another application and then restores the original application at a later time, the application may have actually been killed by the operating system during the interval. To make it so that the UI remains consistent between program launches, recent versions of iOS store object state information in the *Saved Application State* directory by the State Preservation API.[6] Developers can tag specific parts of their UI to be included in State Preservation.

If you're not careful about what you store as part of the application state, this is one place you can wind up with data leaks. I'll discuss how to avoid those in depth in Chapter 10.

The tmp Directory As you might surmise, *tmp* is where you store transient files. Like the *Caches* directory, the files contained in this directory may be automatically removed by the OS while your application isn't running. The usage of this directory is fairly similar to that of the *Caches* directory; the difference is that *Caches* is meant to be used for files that might need to be retrieved again or re-created. For example, if you download certain application data from a remote server and want to keep it around for performance reasons, you'd store that in *Caches* and redownload it if it disappears. On the other hand, *tmp* is for strictly temporary files generated by the application—in other words, files that you won't miss if they're deleted before you can revisit them. Also, like the *Caches* directory, *tmp* is not backed up to iTunes or iCloud.

The Shared Directory

The *Shared* directory is a bit of a special case. It's for applications that share a particular app group (introduced in iOS 8 to support extensions), such as those that modify the behavior of the Today screen or keyboard. Apple requires all extensions to have a container application, which receives its own app ID. The *Shared* directory is the way that the extension and its containing app share data. For example, apps can access databases of shared user defaults by specifying a suite name during initialization of `NSUserDefaults`, like this:

```
[[NSUserDefaults alloc] initWithSuiteName:@"com.myorg.mysharedstuff"];
```

While the *Shared* directory isn't commonly used at the time of writing, it's prudent to check this directory when looking for any sensitive informa- tion potentially stored in preferences or other private data.

6. *http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf* (page 69)

Closing Thoughts

With a basic understanding of the iOS security model, the Cocoa API, and how iOS applications are laid out, you're now ready to move on to the fun stuff: tearing apart applications and finding their flaws. In Part II, I'll show you how to build your testing platform, debug and profile applications, and deal with testing third-party apps for which source code is available.

PART II

SECURITY TESTING

4

BUILDING YOUR TEST PLATFORM

In this chapter, I'll outline the tools you need to review your code and test your iOS applications, and I'll show you how to build a robust and useful test plat- form. That test platform will include a properly set up Xcode instance, an interactive network proxy, reverse engineering tools, and tools to bypass iOS platform security checks.

I'll also cover the settings you need to change in Xcode projects to make bugs easier to identify and fix.
You'll then learn to leverage Xcode's static analyzer and compiler options to produce well-protected

binaries and perform more in-depth bug detection.

Taking Off the Training Wheels

A number of behaviors in a default OS X install prevent you from really digging in to the system internals. To get your OS to stop hiding the things you need, enter the following commands at a Terminal prompt:

```
$ defaults write com.apple.Finder AppleShowAllFiles TRUE $ defaults write
com.apple.Finder ShowPathbar -bool true $ defaults write com.apple.Finder
_FXShowPosixPathInTitle -bool true
$ defaults write NSGlobalDomain AppleShowAllExtensions -bool true $
chflags nohidden ~/Library/
```

These settings let you see all the files in the Finder, even ones that are hidden from view because they have a dot in front of their name. In addi- tion, these changes will display more path information and file extensions, and most importantly, they allow you to see your user-specific *Library*, which is where the iOS Simulator will store all of its data.

The `chflags` command removes a level of obfuscation that Apple has put on directories that it considers too complicated for you, such as */tmp* or */usr*. I'm using the command here to show the contents of the iOS Simulator directories without having to use the command line every time.

One other thing: consider adding *$SIMPATH* to the Finder's sidebar for easy access. It's convenient to use *$SIMPATH* to examine the iOS Simula- tor's filesystem, but you can't get to it in the Finder by default. To make this change, browse to the following directory in the Terminal:

```
$ cd ~/Library/Application\ Support $ open .
```

Then, in the Finder window that opens, drag the iPhone Simulator directory to the sidebar. Once you're riding without training wheels, it's time to choose your testing device.

Suggested Testing Devices

My favorite test device is the Wi-Fi only iPad because it's inexpensive and easy to jailbreak, which allows for testing iPad, iPhone, and iPod Touch applications. Its lack of cellular-based networking isn't much of a hindrance, given that you'll want to intercept network traffic most of the time anyway. But this configuration does have some minor limitations. Most signif- icantly, the iPad doesn't have GPS or SMS, and it obviously doesn't make phone calls. So it's not a bad idea to have an actual iPhone of some kind available.

I prefer to have at least two iPads handy for iOS testing: one jailbro- ken and one stock. The stock device allows for testing in a legitimate, real- istic end-user environment, and it has all platform security mechanisms still intact. It can also register properly for push notifications, which has proven problematic for jailbroken devices in the past. The jailbroken device allows you to more closely inspect the filesystem layout and more detailed workings of iOS; it also facilitates black-box testing that wouldn't be feasible using a stock device alone.

Testing with a Device vs. Using a Simulator

Unlike some other mobile operating systems, iOS development uses a *simula- tor* rather than an emulator. This means there's no full emulation of the iOS device because that would require a virtualized ARM environment. Instead, the simulators that Apple distributes with Xcode are compiled for the x64 architecture, and they run natively on your development machine, which makes the process significantly faster and easier. (Try to boot the Android emulator inside a virtual machine, and you'll appreciate this feature.)

On the flip side, some things simply don't work the same way in the iOS Simulator as they do on the device. The differences are as follows:

Case-sensitivity Unless you've intentionally changed this behavior, OS X systems operate with case-insensitive HFS+ filesystems, while iOS uses the case-sensitive variant. This should rarely be relevant to security but can cause interoperability issues when modifying programs. **Libraries** In some cases, iOS Simulator binaries link to OS X frame- works that may behave differently than those on iOS. This can result in slightly different behavior. **Memory and performance** Since applications run natively in the iOS Simulator, they'll be taking full advantage of your development machine's resources. When gauging the impact of things such as PBKDF2 rounds (see Chapter 13), you'll want to compensate for this or test on a real device. **Camera** As of now, the iOS Simulator does not use your development machine's camera. This is rarely a huge issue, but some applications do contain functionality such as "Take a picture of my check stub or receipt," where the handling of this photo data can be crucial. **SMS and cellular** You can't test interaction with phone calls or SMS integration with the iOS Simulator, though you can technically simulate some aspects, such as toggling the "in-call" status bar.

Unlike in older versions of iOS, modern versions of the iOS Simulator do in fact simulate the Keychain API, meaning you can manage your own certificate and store and manipulate credentials. You can find the files behind this functionality in *$SIMPATH/Library/Keychains*.

Network and Proxy Setup

Most of the time, the first step in testing any iOS application is to run it through a proxy so you can examine and potentially modify traffic going from the device to its remote endpoint. Most iOS security testers I know use BurpSuite[1] for this purpose.

1. *http://www.portswigger.net*

Bypassing TLS Validation There's one major catch to running an app under test through a proxy: iOS resolutely refuses to continue TLS/SSL connections when it cannot authenticate the server's certificate, as well it should. This is, of course, the correct behavior, but your proxy-based testing will screech to a halt rather quickly if iOS can't authenticate your proxy's certificate.

For BurpSuite specifically, you can obtain a CA certificate simply by configuring your device or iOS Simulator to use Burp as a proxy and then browsing to *http://burp/cert/* in Mobile Safari. This should work either on a real device or in the iOS Simulator. You can also install CA certificates onto a physical device by either emailing them to yourself or navigating to them on a web server.

For the iOS Simulator, a more general approach that works with almost any web proxy is to add the fingerprint of your proxy software's CA certifi- cate directly into the iOS Simulator trust store. The trust store is a SQLite database, making it slightly more cumbersome to edit than typical certifi- cate bundles. I recommend writing a script to automate this task. If you want to see an example to get you started, Gotham Digital Science has already created a Python script that does the job. You'll find the script here: *https://github.com/GDSSecurity/Add-Trusted-Certificate-to-iOS-Simulator/*.

To use this script, you need to obtain the CA certificate you want to install into the trust store. First configure Firefox[2] to use your local proxy (127.0.0.1, port 8080 for Burp). Then attempt to visit any SSL site; you should get a familiar certificate warning. Navigate to **Add Exception** → **View** → **Details** and click the **PortSwigger CA** entry, as shown in Figure 4-1.

Click **Export** and follow the prompts. Once you've saved the CA certifi- cate, open *Terminal.app* and run the Python script to add the certificate to the store as follows:

```
$ python ./add_ca_to_iossim.py ~/Downloads/PortSwiggerCA.pem
```

Unfortunately, at the time of writing, there isn't a native way to config- ure the iOS Simulator to go through an HTTP proxy without also routing the rest of your system through the proxy. Therefore, you'll need to config- ure the proxy in your host system's Preferences, as shown in Figure 4-2.

If you're using the machine for both testing and other work activities, you might consider specifically configuring other applications to go through a separate proxy, using something like FoxyProxy[3] for your browser.

2. I generally consider Chrome a more secure daily browser, but the self-contained nature of Firefox does let you tweak proxy settings more conveniently. 3. *http://getfoxyproxy.org*

Figure 4-1: Selecting the PortSwigger CA for export

Figure 4-2: Configuring the host system to connect via Burp

Figure 4-3: Setting up invisible proxying through the local stunnel endpoint

4. *http://www.stunnel.org/*

Bypassing SSL with stunnel One method of bypassing SSL endpoint verification is to set up a termina- tion point locally and then direct your application to use that instead. You can often accomplish this without recompiling the application, simply by modifying a plist file containing the endpoint URL.

This setup is particularly useful if you want to observe traffic easily in plaintext (for example, with

Wireshark), but the Internet-accessible endpoint is available only over HTTPS. First, download and install stun- nel,[4] which will act as a broker between the HTTPS endpoint and your local machine. If installed via Homebrew, stunnel's configuration file will be in */usr/local/etc/stunnel/stunnel.conf-sample*. Move or copy this file to */usr/local/etc/stunnel/stunnel.conf* and edit it to reflect the following:

```
; SSL client mode client = yes

          ; service-level configuration [https] accept
          = 127.0.0.1:80 connect = 10.10.1.50:443
          TIMEOUTclose = 0
```

This simply sets up stunnel in client mode, instructing it to accept con- nections on your loopback interface on port 80, while forwarding them to the remote endpoint over SSL. After editing this file, set up Burp so that it uses your loopback listener as a proxy, making sure to select the **Support invisible proxying** option, as shown in Figure 4-3. Figure 4-4 shows the result- ing setup.

Figure 4-4: Final Burp/stunnel setup

Certificate Management on a Device

To install a certificate on a physical iOS device, simply email the certificate to an account associated with the device or put it on a public web server and navigate to it using Mobile Safari. You can then import it into the device's trust store, as shown in Figure 4-5. You can also configure your device to go through a network proxy (that is, Burp) hosted on another machine. Simply install the CA certificate (as described earlier) of the proxy onto the device and configure your proxy to listen on a network-accessible IP address, as in Figure 4-6.

Figure 4-5: The certificate import prompt

Figure 4-6: Configuring Burp to use a nonlocalhost IP address

Figure 4-7: Configuring the device to use a test proxy on an internal network

Note that when configuring a device to use a proxy, only connections initiated by `NSURLConnection` or `NSURLSession` will obey the proxy settings; other connections such as `NSStream` and `CFStream` (which I'll discuss further in Chap- ter 7) will not. And of course, since this is an HTTP proxy, it works only for HTTP traffic. If you have an application using `CFStream`, you can edit the codebase with the following code snippet to route stream traffic through the same proxy as the host OS:

```
CFDictionaryRef systemProxySettings = CFNetworkCopySystemProxySettings();
```

```
CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, systemProxySettings
    );

CFWriteStreamSetProperty(writeStream, kCFStreamPropertyHTTPProxy,
    systemProxySettings);
```

Proxy Setup on a Device Once you've configured your certificate authorities and set up the proxy, go to **Settings** → **Network** → **Wi-Fi** and click the arrow to the right of your currently selected wireless network. You can enter the proxy address and port from this screen (see Figure 4-7).