ABSTRACT

DESHOTELS, LUKE ALEC. Automated Evaluation of Access Control in the iPhone Operating System. (Under the direction of William Enck).

A decade long arms race between Apple and jailbreakers has forged iPhone OS (iOS) into a hardened, but complex operating system. As a modern operating system, iOS treats applications as security principals, which allows for fine-grained access control policies that prevent applications from having the same authority as the device owner. However, iOS and other modern operating systems use inter-dependent access control mechanisms (e.g., Unix permissions and sandboxing), and the policies on iOS may be closed-source, proprietary, or decentralized. This practice makes it difficult to model the actual privileges of an iOS process which is subject to various undocumented access control policies. Confusion regarding these privileges hides flaws in misconfigured iOS access control policies that attackers can exploit in order to invade user privacy or damage the system.

This dissertation demonstrates that the access control mechanisms in iOS are inter-dependent and their analysis should consider the composite protection system as a whole. We analyze three access control systems in iOS, the Apple Sandbox, Unix Permissions, and Inter-Process Communication (IPC). First, we present the SandScout framework which converts iOS Apple Sandbox policies from their compiled proprietary state into queriable Prolog facts. SandScout's utility is demonstrated by detecting seven types of novel vulnerabilities allowing third party apps to steal private data or damage the system, which resulted in six Common Vulnerability and Exposure (CVE) acknowledgements from Apple. Second, we introduce iOracle, a logical framework for extracting multiple access control policies (i.e., the Apple Sandbox and Unix Permissions) as well as runtime context from iOS and unifying this policy and contextual data into a composite model. iOracle is evaluated through detection of previously known exploits as well as detection of unknown policy flaws. Finally, we propose Kobold, a framework for enumerating and evaluating IPC remote methods accessible by third party applications. Access to these remote methods is regulated by the Apple Sandbox and static capabilities called entitlements. Therefore, Kobold also automates the collection and extraction of third party application entitlements. During this analysis Kobold discovers third party applications with undocumented, semi-private entitlements. Kobold identifies over one hundred accessible methods and detects previously unknown access control flaws and daemon crashes.

© Copyright 2018 by Luke Alec Deshotels

All Rights Reserved

Automated Evaluation of Access Control in the iPhone Operating System

by
Luke Alec Deshotels

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Bradley Reaves	Alexandros Kapravelos
Matthias Stallmann	William Enck Chair of Advisory Committee

DEDICATION

This dissertation is dedicated to my parents for their unwavering love and support.

BIOGRAPHY

Luke received his Bachelor's and Master's from the University of Louisiana at Lafayette where he studied computer vision and malware analysis. He started the PhD program at NCSU in order to study Android malware with Dr. Xuxian Jiang. However, through a serendipitous series of events involving significant help from Dr. Purush Iyer, ultrasonic sound-waves, and an extra-credit report on OS verification, Luke started studying the iOS sandbox with Dr. William Enck. It's a long story that won't be told here, but Luke has no regrets with the way things turned out.

ACKNOWLEDGEMENTS

I could have never made it this far on my own. In my darkest moments of paper rejections, technical setbacks, and bureaucratic nightmares, I was never alone. Along my journey there was always help in the forms of loving support, writing advice, and technical contributions. In addition to my committee members, Bradley Reaves, Alexandros Kapravelos, and Matthias Stallmann, I must thank my advisor, co-authors, labmates, and friends who have given so much and asked nothing in return.

First, I must thank my advisor, William Enck. I have been Will's teaching assistant, student, and apprentice. As Will's teaching assistant, he demonstrated his fairness and professionalism as we handled issues of academic integrity together. As his student, I learned exactly the information I would need to win and thrive in computer security internships. As an apprentice, I learned to write, design experiments, analyze results, present research, and collaborate with other universities. Will has been generous with his knowledge, time, and introductions, and at every stage of my academic career, Will has acted as my role model.

Second, I owe much to my co-authors, especially Razvan Deaconescu who has given so much more time and brilliance than I deserved. Razvan Deaconescu reversed the Apple sandbox, and made many other technical contributions. It was Razvan that found our first vulnerability and inspired the team to keep pursuing iOS access control. Mihai Chiroiu hosted me at Politehnica University of Bucharest and helped us keep an eye on the bigger picture beyond just iOS. Costin Carabas was instrumental to discovering the correct syntax to invoke NSXPC methods on iOS, and he contributed in many other ways to iOracle and Kobold. Iulia Manda contributed to our knowledge of mach-ports and sandbox extensions and was kind enough to find me during my brief visit to Romania. Lucas-Davi contributed his expertise of iOS and writing skill to SandScout. Ahmad-Reza Sadeghi hosted me at TU Darmstadt and provided significant insight and style to our papers. It was Ahmad that put our team together and introduced me to Razvan, Mihai, and Lucas. Finally, Ninghui-Li contributed his expertise of access control analysis to iOracle.

Third, I would like to acknowledge my labmates for their support and advice throughout my time in the WSPR lab. Micah Bushouse has been my greatest critic, but he has also been a true friend and a source of wisdom and encouragement. Isaac Polinsky has been a friend and one of the most entertaining people in the lab. Ben Andow started his PhD program in the same year as me, and has given me many insights into low level sytem's research. Ben and I often joked that our research topics should have been reversed. Sigmund "Al" Gorski shared his relevant knowledge of Android access control analysis, and he was a great roommate during our Qualcomm internship. Adwait Nadkarni was a great verbal sparring partner and lead graduate student. I look forward to working at Samsung with Akash Verma, Michael Grace, and Kunal Patel. Jason Gionta inspired me to ask many questions. I enjoyed exercising and discussing shared hobbies with Russell Meredith. Sarah

Elder provided wisdom and news from outside the lab. Sanghyun went to great lengths to make me feel at home during my trip to Korea. Kyle Martin gave me insights into fuzzing, crashes, and memory exploitation. I would like to thank the other current and past members of the WSPR lab for their company, advice, and hard work.

Finally, I am honored to list many friends that helped me throughout my PhD. Rusty Gibson contributed his time and expertise to a very fun side project on steganography in video games. Sean Mealin, Peipei Wang, Sheldon Abrams, Barry Peddicord, and Aurora Peddicord provided advice as students themselves, and they were excellent gaming partners. Feifei Wang was very kind to me and introduced me to many more friends. David Fiala helped recruit me to NCSU and encouraged me to persist through hard times. Most of all, I would like to thank Xiaoting "Sisi" Fu, for her patience and love throughout the ups and downs of my final year as a PhD student. Lastly, I thank Noodle, the cat, for listening to explanations of iOS access control policies when I needed an audience.

TABLE OF CONTENTS

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

Chapter 1 INTRODUCTION **1**
1.1 Thesis Statement 2
1.2 Contributions 5
1.3 Thesis Organization 6

Chapter 2 Background **9**
2.1 Introduction 9
2.2 iOS Basics 11
2.3 Entitlements 12
2.3.1 Policy 12
2.3.2 Enforcement 13
2.4 Sandboxing 14
2.4.1 Policy 14
2.4.2 Enforcement 16
2.5 Privacy Settings 17
2.5.1 Policy 19
2.5.2 Enforcement 20
2.6 Conclusions 21

Chapter 3 Related Work **22**
3.1 Access Control Policy Design 22
3.2 Access Control Policy Evaluation 23
3.3 iOS Security 24
3.3.1 Binary Analysis of iOS Apps 24
3.3.2 Exploits for Third Party Apps 25
3.3.3 In-Line Reference Monitors for Third Party Apps 26

Chapter 4 SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles **28**
4.1 Introduction 28
4.2 Background 30
4.2.1 iOS Security Mechanisms 30
4.2.2 Sandbox Profile Language (SBPL) 31
4.3 Overview 33
4.4 Design 35
4.4.1 Decompiling Sandbox Profiles 36
4.4.2 Modeling Sandbox Profiles in Prolog 37
4.4.3 Policy Analysis 39
4.4.4 Attack Testing Application 41
4.5 Results 42

4.5.1 Prolog Query Results . . . 42
4.5.2 Verified Attacks . . . 44
4.6 Limitations . . . 48
4.7 Related Work . . . 49
4.8 Conclusions . . . 50
4.9 Acknowledgments . . . 51

Chapter 5 iOracle: Automated Evaluation of Access Control Policies in iOS . . . 53
5.1 Introduction . . . 53
5.2 Background . . . 55
5.3 Overview . . . 56
5.4 iOracle . . . 59
5.4.1 Policy and Context Extraction . . . 59
5.4.2 Knowledge Base Construction . . . 62
5.5 Case Study: iOS Jailbreaks . . . 66
5.5.1 Understanding iOS Jailbreaks . . . 66
5.5.2 Evaluating iOracle . . . 71
5.6 Previously Unknown policy flaws . . . 72
5.6.1 Self-Granted Capabilities . . . 72
5.6.2 Capability Redirection . . . 73
5.6.3 Write Implies Read . . . 74
5.6.4 Keystroke Exfiltration . . . 74
5.6.5 Chown Redirection . . . 75
5.7 Comparison of iOS Versions . . . 76
5.7.1 Access Control Complexity . . . 76
5.7.2 Detecting Responses to Jailbreaks . . . 76
5.8 Other Policy flaws . . . 78
5.8.1 Denial of Service . . . 78
5.8.2 Address Book Privacy Setting Bypass . . . 79
5.8.3 Symlink Restriction Bypass . . . 79
5.9 Limitations . . . 80
5.10 Related Work . . . 80
5.11 Conclusions . . . 82
5.12 Acknowledgments . . . 82

Chapter 6 A House of Many Doors: Evaluating Access Control for Remote NSXPC Methods on iOS . . . 83
6.1 Introduction . . . 83
6.2 Background . . . 85
6.2.1 Mach IPC . . . 86
6.2.2 IPC Access Control . . . 87
6.3 Overview . . . 89
6.4 Kobold . . . 92
6.4.1 Identify Third Party Entitlements . . . 92
6.4.2 Enumerate Accessible NSXPC Services . . . 93

6.4.3 Evaluate Security Sensitivity of NSXPC Services 95
6.5 Entitlement Survey Results 97
6.5.1 Public Entitlements 97
6.5.2 Semi-Private Entitlements 98
6.6 Empirical Study 100
6.7 Findings 103
6.7.1 Confused Deputy Attacks 103
6.7.2 Daemon Crashes 104
6.8 Limitations 105
6.9 Related Work 106
6.10 Conclusion 108
6.11 Acknowledgements 108

Chapter 7 DIRECTIONS FOR IOS ACCESS CONTROL ANALYSIS 112
7.1 Dissertation Summary 112
7.2 Trends 114
7.3 Future Work 116
7.4 Concluding Remarks 117

BIBLIOGRAPHY 118

LIST OF TABLES

Table 2.1 tccd managed privacy settings for iOS 8. 19
Table 2.2 TCC database experiments. 21

Table 4.1 Logic for Match/Unmatch Edges* 37
Table 4.2 Attack Verification Functions 42
Table 4.3 Query Results for iOS 9.0.2 43

Table 5.1 Policy and Runtime Context Prolog Facts 61
Table 5.2 Triage of Likely Attack Vectors and Confused Deputies Based on Known Jailbreak Gadgets 69
Table 5.3 Measuring the Increasing Complexity of iOS Access Control 76

Table 6.1 Public Entitlements 98
Table 6.2 Semi-Private Entitlements 99
Table 6.3 Per Method Entitlement Requirements Based on Error Messages 100
Table 6.4 Entitlement Requirements for Port Access Through Sandbox 101
Table 6.5 Methods by Number of Arguments 102
Table 6.6 Common Data Types in Extracted NSXPC Methods 109
Table 6.7 Methods per Mach Port. Inconsistent Entitlement Requirements Highlighted. 110
Table 6.8 Confused Deputy Attacks 110
Table 6.9 Daemon Crashes 111

LIST OF FIGURES

Figure 2.1 Overview of iOS access control according to access revocability and resource being managed. 10
Figure 2.2 Sandbox Architecture 17
Figure 2.3 Privacy Settings Policy Architecture 18
Figure 2.4 Requesting access to a file-based resource through TCC 20

Figure 4.1 Overview of SandScout. 31
Figure 4.2 Converting Graph to SBPL. 38
Figure 4.3 SBPL Context Free Grammar. 52

Figure 5.1 iOracle Overview 57
Figure 5.2 Simplified Hierarchy of Prolog Rules 63
Figure 5.3 Caption for LOF 66
Figure 5.4 Name Resolution Based Jailbreak Steps 78
Figure 5.5 Capability Based Jailbreak Steps 78

Figure 6.1 Three Stages of NSXPC Access Control: 1) Sandbox Access to Port; 2) Entitlement Checks for Port; 3) Entitlement checks for Remote Method 88
Figure 6.2 Kobold Overview 90
Figure 6.3 NSXPC Method Invocation Quantitative Results 101

CHAPTER

1 INTRODUCTION

The smartphone market is currently dominated by iOS (iPhone Operating System) and Android powered devices. Due to the closed source nature of iOS, the vast majority of mobile operating system security research has been focused on Android. While the academic community focused on Android, hackers frustrated with security restrictions in iOS sought exploits that could elevate their privileges. Since Apple's restrictions were referred to as a jail, exploits that bypassed them were called jailbreaks. From iOS 1 and iOS 11, a decade long arms race between "jailbreakers" and Apple would harden iOS into one of the world's most advanced operating systems. However, as the security features of iOS became more complex, they also became more difficult to validate.

iOS is a mobile operating system based on macOS, the operating system for Apple's Mac desktops. However, unlike desktop operating systems, mobile operating systems often assume the device has one human user that does not require root access, the device owner. By withholding administrative rights from the device owner and treating applications as security principals, iOS access control mechanisms mitigate the damage of operator errors and malicious programs. Instead of defining access control policies that protect human users from one another, the policies protect the user from untrusted applications and protect applications from each other. For example, with a single non-root user on iOS, Unix permissions can be used to prevent the user or third party applications from modifying system files. iOS applications are further isolated by other mechanisms including a sandbox that restricts system calls, capability systems, and access control lists. These mechanisms exist on modern versions of macOS, but the assumption of one non-root human user on an iOS

device allows Apple to apply significantly more restrictive policies.

For the first version of iOS, Apple assumed developers would only write web applications. However, Apple quickly changed this policy and allowed for the creation of native third party applications that could be published on an App Store. As of January 2017, the Apple App Store contained over 2 million applications, and as of 2016 developers earned over 20 billion dollars in App Store revenue. With so many applications available on the App Store there is little need for iOS users to seek applications from any other source. This allows Apple to use the App Store as central point of control over third party software allowing Apple to set requirements for entry to the app store or remove any applications found to be malicious.

In the time from the first iPhone to August 2017, Apple sold over one billion iPhones. Many users of these devices trust them with financial credentials, personal photographs, location data, health records, and more sensitive information. Such information is of great value to advertisers, potential blackmailers, and surveillance states. This wealth of data on has led attackers to create weaponized jailbreaks that can bypass multiple layers of security mechanisms to gain elevated privileges, stealth, and persistance [Loo]. Simpler attacks allow malicious applications to bypass only the restrictions protecting user privacy [Wan13; Han13b]. Fortunately the access control mechanisms implemented in iOS can mitigate these attacks, if they are configured correctly.

In order for an operating system to provide finer grained access control and accomodate new features, multiple complex access control policies can be defined. However, as these policies increase in complexity and interdependence they become more difficult to manually evaluate. Even with a comprehensive model of each policy, the interactions between policies could produce unexpected results (e.g., one policy overriding another).

As we will show in Chapter 5 the complexity of iOS access control policies is increasing significantly over time. iOS also implments multiple access control mechanisms that have undocumented interactions with each other. These challenges and the closed source nature of iOS have led to a dearth of research in iOS security relative to an abundance of Android security research. However, the prevalence of jailbreaks exploiting multiple misconfigurations in iOS access control policies calls for better methods of evaluating these policies.

1.1 Thesis Statement

Before the works in this dissertation, existing tools and studies were not sufficient to model iOS access control policies. We attribute this inadequacy to the difficulty of extracting relevant policy and contextual data from iOS and the largely undocumented semantics of these policies. Prior work has modeled complex access control policies such as those of SELinux [1]. However, no prior work has applied these modeling techniques to the access control policies of iOS and they generally focus

[1] https://github.com/SELinuxProject

on modeling access control mechanisms in isolation. While researchers had taken steps to reverse engineer the iOS sandbox, their tools could only provide a partial decompilation of sandboxes from iOS 7 and later. Other tools extracted security relevant data but could not integrate this data toward a comprehensive model.

To the best of our knowledge iOS uses the following access control mechanims: 1) Unix permissions; 2) POSIX ACLs; 3) Apple Sandbox / Seatbelt; 4) adhoc access control lists stored in files; and 5) mutable and immutable capabilities. Many of these mechanisms offer negative policies such as capabilities that remove privileges instead of providing them. This expressability can lead to conflicts within and between policies, and the system's resolution of these conflicts seems to be based on adhoc patches built around features as opposed to a centralized metapolicy. For example, Unix permissions and sandboxing must agree to allow a process to access a file, but capabilities can override membership requirements in adhoc ACL files.

The goal of this dissertation is to model multiple iOS access control policies into a unified and queriable model, by converting extracted policy data and context into Prolog facts and representing policy semantics as Prolog rules, such that analysts can automatically evaluate security qualities of the system as a whole.

This goal is accomplished in three works. First, SandScout models iOS sandbox policies such that vulnerabilities in the policy used for third party applications can be found with simple queries. However, this is not sufficient to model jailbreaks or system processes with unknown context. Second, iOracle integrates Unix permissions, sandbox policies, and runtime context to provide a better model for detecting jailbreaks. iOracle lacks the ability to model access control for Inter-Process Communication (IPC) services, which are relevant to modeling jailbreaks but largely use adhoc capability checks instead of centralized policies. Third, we present Kobold, a framework for enumerating and evaluating IPC remote methods accessible to third party applications.

The scope of this dissertation is limited to access control policies in iOS. iOS implements an application verification process and several code security mechanisms including code signing, address space layout randomization (ASLR), and data execution prevention (DEP). However, researchers and malware authors have demonstrated strategies to bypass these mechanisms including return-oriented programming (ROP), export symbol redirection, and confused deputy attacks. Therefore, access control is required to mitigate the damage caused by malicious third party applications or compromised system applications. Unfortunately for analysts, Apple chose not implement iOS access control in a single centralized policy. Instead a combination of existing technologies, centralized proprietary mechanisms, and adhoc policies built directly into new features. There is a concern that access control for third party applications could be defeated by applications that request access to all privileges without justification. However, Apple's app vetting process serves well as a barrier to such greedy apps. While the works in this dissertation focus on iOS, the idea of modeling policies for inter-dependent access control policies could be applied to other modern operating systems

such as Android and macOS.

This dissertation addresses three research questions (RQ) regarding iOS access control.

- *RQ1: What flaws in sandbox policies can third-party iOS applications exploit?* Sandscout led us to identify seven types of vulnerability present in the sandbox policy for third party apps.
- *RQ2: What flaws in iOS filesystem access control policies can jailbreak attacks exploit?* iOracle detected flaws exploited by existing jailbreaks as well previously undisclosed vulnerabilities in iOS access control policies that can lead to privilege escalation attacks usable in jailbreaks.
- *RQ3: Which third party accessible IPC remote methods provide security sensitive functionality?* Kobold enumerates the set of IPC remote methods implemented using the NSXPC interface that are accessible to third party applications. Since access to IPC methods is regulated by both the sandbox and decentralized entitlement requirements, Kobold is designed to consider both policies. Kobold then dynamically tests these remote methods to detect daemon crashes and functionality that allows third party applications to invade user privacy or damage the system.

Many of the access control policies in iOS can override decisions made by other policies. These interactions among policies could lead to false assumptions when evaluating a policy in isolation. Therefore, to evaluate the overall security quality of access control policies on iOS, it is not only necessary to model each policy, but also necessary to model the interactions of these policies and the runtime context of the system, which leads us to the following thesis:

Platforms such as iOS with multiple access control mechanisms contain interdependent policy flaws that require analysis of the composite protection system.

To model the interactions of access control policies we first need to extract and model the policies individually, and we began with Apple's sandbox mechanism. In modeling and evaluating the sandbox policies we found vulnerabilities in the container sandbox policy allowing third party applications to invade user privacy and damage the system. Next, we needed to model other policies and runtime context as well as build a framework to model these mechanisms along with the sandbox policy data. With iOracle we studied previous jailbreaks and used the framework to triage and help detect the attacks exploited by jailbreaks. We also found previously undiscovered

vulnerabilities in access control policies. iOracle and SandScout limit their scope to the file system largely because many IPC mechanisms are regulated through adhoc checks for entitlements as opposed to the sandbox or Unix permissions. IPC is important for jailbreaks and exploits in general and can influence file system security if a service performs file manipulation. In order to expand our scope to IPC, we must perform an empirical analysis of entitlements and the IPC services they can provide access to.

The scope of this dissertation includes iOS access control policies and their influence on file system operations and inter-process communication. We restrict our scope to the file system and IPC for three reasons. First, unless the content of a file or IPC reply message is encrypted, we can perform manual analysis of the data to determine if it contains private or high integrity information and whether access to the file or reply message constitutes a security violation. Second, we can confirm that a subject has access to files and IPC without significant manual reverse engineering. Third, many of the jailbreaks we have studied must gain access to certain system files or IPC connections as part of their privilege escalation exploits.

Notable topics outside of our scope include software vulnerabilities, network access, driver access, and secure enclave. We assume that Apple assigns access control policies to those applications that Apple expects to become compromised. We also assume that if the application's behavior is maliciously modified the access control policy is expected to mitigate harm and prevent privilege escalation. Therefore, we consider the details of attacks that cause modifications in an application's behavior to be out of scope. Instead, we focus on flaws in access control policies that fail to mitigate damage caused by such a compromised application. Networks, drivers, shared memory, and other resources are also relevant to access control. However, without access to source code or official specifications, it is intractable for us to determine the purpose of each of these resources and whether access to these resources is justified or safe.

1.2 Contributions

SandScout (CCS 2016): The SandScout work provides three contributions. First, we provide the first methods to automatically produce human readable SBPL policies. Prior work was unable to produce SBPL policies for human review or automated analysis. A component of SandScout called SandBlaster extracts and decompiles all sandbox profiles in firmwares for iOS 7, 8, and 9[2]. Second, we formally model SBPL policies using Prolog. We create an SBPL to Prolog compiler based on a context free grammar we have defined for SBPL. Third, we perform the first systematic evaluation of the container sandbox profile for recent versions of iOS and discover vulnerabilities. We develop Prolog queries representing security requirements. When these queries were applied to the iOS 9.0.2

[2]Extended for iOracle and Kobold to process iOS 10 and 11 sandbox profiles

container sandbox profile, we discovered seven classes of security vulnerabilities.

iOracle (AsiaCCS 2018): iOracle also makes three contributions. First, we present the iOracle policy analysis framework. iOracle models iOS access control including sandbox policies, Unix permissions, policy semantics, and runtime context. Second, we demonstrate iOracle's utility through an analysis of four recent jailbreaks. We show a significant reduction in the number of subjects and objects that need to be considered by security analysts. Finally, we identify new vulnerabilities in iOS access control policies. Exploiting these vulnerabilities may allow sandbox manipulation, keylogger data exfiltration, bypassing symlink restrictions, name resolution attacks, and denial of service. We have disclosed these vulnerabilities to Apple.

Kobold: Finally, Kobold has three contributions. First, Kobold provides the first framework for enumerating and evaluating NSXPC remote methods accessible to third party applications. Second, Kobold performs an automated survey of entitlement use among popular App Store applications. During this analysis we discover 17 semi-private entitlements possessed by third party applications that are not normally available to developers. Third, we identify previously unknown daemon crashes and access control policy flaws allowing third party applications to invade user privacy and disable system functionality.

1.3 Thesis Organization

This dissertation aims to address the four challenges associated with modeling overall access control for a closed-source system implementing multiple inter-dependent access control mechanisms. First, due to a lack of clear documentation and Apple's closed source nature, iOS access control mechanisms are poorly understood. Second, Apple's stores the built-in iOS sandbox policies in a proprietary format, and even when decompiled into their original human-readable language, these policies are impractical to query. Third, iOS sandbox policies rely significantly on runtime context and are supplemented by Unix permissions, which must also be extracted and merged with sandbox policy data in order to build a more comprehensive model. Fourth, IPC services may expose a significant attack surface, but this attack surface is difficult to measure due to a lack of source code, documentation, or centralized policies regulating access to these services.

Chapter 2: We provide background information and an analysis of relevant prior work on iOS and modeling of access control policies in Chapter 2. This chapter will introduce relevant terms including sandboxing, entitlements, privacy settings, and jailbreaking. The chapter will also provide a brief overview of relevant non-access control security features present in iOS. Next, Chapter 2 provides an overview of the prior work related to this dissertation.

Chapter 3: In chapter 3 we discuss related academic work in the fields of access control and iOS security. We provide a brief overview of access control policy design literature as well as efforts

to automatically evaluate access control policies. Finally, we survey iOS security research and demonstrate that most of the work is focused on the security of third party applications as opposed to access control and overall system security.

Chapter 4: Recent literature on iOS security has focused on the malicious potential of third-party applications, demonstrating how developers can bypass application vetting and code-level protections. In addition to these protections, iOS uses a generic sandbox profile called “container” to confine malicious or exploited third-party applications. In Chapter 4, we present the first systematic analysis of the iOS container sandbox profile. We propose the SandScout framework to extract, decompile, formally model, and analyze iOS sandbox profiles as logic-based programs. We use our Prolog-based queries to evaluate file-based security properties of the container sandbox profile for iOS 9.0.2 and discover seven classes of exploitable vulnerabilities. These attacks were effective on non-jailbroken iOS devices. We have worked with Apple to resolve the attacks, and we expect that SandScout will play a significant role in the development of sandbox profiles for future versions of iOS.

Chapter 5: iOS security research has primarily focused on the security of third party applications. However, attackers can exploit access control flaws in system processes to create multi-stage attacks known as jailbreaks. A jailbreak can bypass and disable iOS security features, providing heightened privileges, stealth, and persistence. iOS access control policies have the potential to prevent such attacks, but manually verifying the correctness of such policies for hundreds of system executables is infeasible. In Chapter 5 we propose iOracle, a logical framework that models iOS access control semantics, policies, and runtime context. iOracle automatically extracts policies and runtime context from iOS firmware images, developer resources, and jailbroken devices. We evaluate iOracle by detecting flaws that allowed four recent jailbreaks. We then use iOracle to identify five types of previously unknown vulnerabilities in iOS access control policy including sandbox manipulation, keylogger data exfiltration, bypassing symlink restrictions, name resolution attacks, and denial of service attacks. By automating the evaluation of iOS access control, iOracle provides a scalable approach to hardening iOS security by identifying flaws before they are exploited.

Chapter 6: While the privileges of third party applications are significanlty restricted, they may still use IPC services to cause another, more privileged process to perform dangerous operations as a result of confused deputy attacks. Access to remote IPC methods is primarily restricted in two ways: 1) sandbox policies; 2) adhoc checking for entitlements. Through sandbox analysis, we can detect rules restricting access to IPC ports, but the sandbox does not specify declarations of the remote methods available on those ports. Further complicating analysis, adhoc entitlement checks are decentralized and may not be consistently implemented. These characteristics present significant reverse engineering challenges in determining which entitlements, if any, are required to access each service. In Chapter 6 we present Kobold, which addresses these challenges in three

ways. First, Kobold automates the collection of third party applications and the extraction of their entitlements and metadata. This extracted data allows us to determine which entitlements are available to third party applications on the App Store. Second, Kobold enumerates the remote methods accessible to third party applications through the NSXPC interface for IPC. While iOS uses several IPC interface types, the object-oriented nature of NSXPC makes it both popular and amenable to static analysis. Third, Kobold dynamically tests the enumerated remote methods by invoking them and observing system activity and reply messages. Kobold successfully causes several daemon crashes and identifies multiple confused deputy attacks.

Chapter 7: The final chapter of this dissertation summarizes the conclusions made by each chapter. It also observes current trends in iOS access control analysis and suggests areas for future research. Finally, Chapter 7 makes concluding remarks regarding lessons learned throughout the course of this work.

CHAPTER

2 BACKGROUND

2.1 Introduction

Apple's mobile devices are powered by an operating system called iOS (iPhone Operating System). With over one billion iPhones sold, iOS devices have become a fundamental part of modern life. Users trust their smartphones with financial, medical, and personal data. They also rely on their smartphones for navigation, accessing documents, and making mobile payments.

Unlike Android, which is open source and rigorously studied by researchers, iOS is largely a closed source system, and Apple reveals very little about how they secure iOS devices. However, we find that understanding iOS security leads researchers to discover flaws and improve overall system security. For example, Deshotels et al. [Des16], found several vulnerabilities by reverse engineering Apple's obfuscated access control policies. These vulnerabilities could have been obvious to security researchers if the policies had been openly available. Despite the vulnerabilities discovered by reverse engineers, Apple has chosen to continue obfuscating their access control policies. Unfortunately, much of the existing iOS literature is fragmented, obsolete, or over-simplified. In order to make the field of iOS security research more accessible, we have written this article to provide a clear but technical overview of iOS access control.

This article will cover the three[1] primary mechanisms of iOS access control. For each mechanism,

[1] Unix permissions are used extensively in iOS, but we do not cover them in this chapter due to extensive existing literature on the topic.

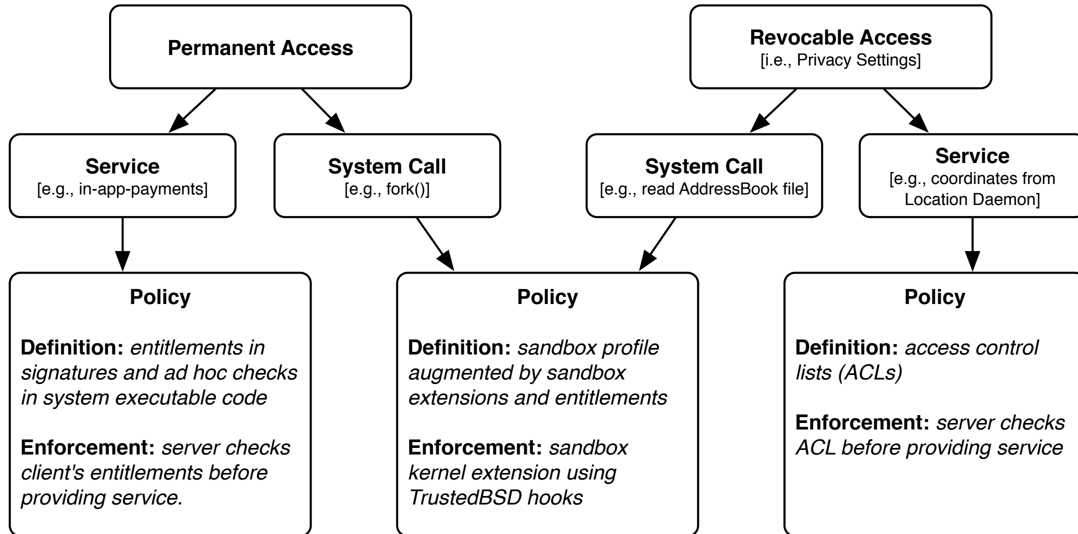

Figure 2.1 Overview of iOS access control according to access revocability and resource being managed.

we will discuss its strengths and weaknesses as well as how the mechanism's policies are defined and enforced. First, Apple provides immutable capabilities to an executable through a mechanism called *entitlements*. Any enforcer can require that a process possess certain entitlements in order to access a protected resource. Second, system calls made by processes in iOS are regulated by a mechanism called *sandboxing*. Sandbox profiles provide the policies that specify whether or not the sandbox kernel extension will allow the system call to proceed. Third, to allow users to dynamically control access to their private data, iOS provides a mechanism called *privacy settings*. The user can grant or revoke access to private data for each application and at any time.

Figure 2.1 categorizes these access control mechanisms based on access (i.e. revocable or not) and resource (i.e. system call or service). Access to resources protected by privacy settings can be arbitrarily revoked by the user. In contrast, access to other resources is defined by immutable policies (e.g., entitlements) that do not support revocation. The two resource types we consider are system calls and services. The sandbox regulates access to system calls, but services provided by processes are regulated through entitlements or privacy settings. Each access type and resource type is managed by mechanisms that define and enforce policies for access control. For example, access control policies for revocable services are defined by mutable Access Control Lists (ACLs). Servers provide enforcement by referencing the ACL when deciding whether or not to allow a client to access a given resource. Note that the terms server and client are used in the context of inter-process communication within the device.

2.2 iOS Basics

iOS was created by modifying Apple's macOS operating system which powers their Macintosh computers. Both iOS and macOS expand upon an operating system called Darwin which runs a kernel called XNU. While the XNU kernel and the Darwin operating system are open source, the frameworks that make up iOS and macOS are proprietary. The XNU kernel consists of three components, the Mach microkernel, code from FreeBSD (Berkeley Software Distribution) , and a driver API called I/O Kit. We will show that these three components play a significant role in iOS access control.

To understand the role of access control in iOS we must consider other security mechanisms. iOS will only run executables signed by developers certified by Apple. In addition to the developer signature, applications on the Apple App Store must be vetted and signed by Apple. While the details of Apple's vetting process remain a mystery, it is assumed that Apple uses a combination of static and dynamic analysis to detect malicious behavior. Apple supplements code signing with two other well-known exploit mitigation techniques, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). The former ensures that no code page is writable and executable at the same time to prevent code injection attacks. The latter technique randomizes code and data segments in memory to make code-reuse attacks such as Return-Oriented Programming (ROP) more cumbersome.

iOS implements Unix file permissions, which is a form of access control. However, we will not dedicate a section to this topic because it is already well documented, and iOS only uses them at a coarse level. While Android has each application running as a unique user, iOS runs all applications (e.g., third party apps) as a user called `mobile`. There are several system processes that run as `mobile` and a few that run as more specialized users. High integrity daemons on iOS run as `root`. User data is also stored in files owned by `mobile`. Therefore, many of the files that need to be protected from third party apps running as `mobile` are already owned by `mobile`.

Note that these security mechanisms are not perfect. Wang et al. [Wan13] demonstrated that malicious developers can bypass Apple's vetting process by crafting vulnerable apps and submitting them to the app store. After the app has passed Apple's vetting process, the developers attack their own app with Return-Oriented Programming (ROP) attacks to modify the app's control flow. By modifying a process's control flow, attackers can cause it to perform malicious actions without injecting new code or violating the code's signature. Code signing makes it more difficult to implement fine-grained ASLR. Therefore, iOS's implementation of ASLR is not sufficient to prevent ROP attacks. Without other forms of access control, a process running as `mobile` would have sufficient Unix permissions to damage the system or access private user data. For these reasons, iOS relies on entitlements, sandboxing, and privacy settings to govern access to many sensitive resources.

2.3 Entitlements

Entitlements are permanent attributes embedded in an executable's signature. With respect to Figure 2.1, entitlements can regulate access to services, and they play a role in sandboxing, but they cannot be revoked or granted at runtime. For example, the Mobile Safari application has an entitlement that allows it to run unsigned code (for just in time compilation optimizations).

Strengths: If a malicious application tries to change its list of entitlements, this will violate its signature and prevent it from launching. The only way to change entitlements is for an application developer to release an updated version of the application with a new signature. Clear distinction between high risk and low risk entitlements makes it easy to prevent an untrusted application from acquiring a high risk entitlement. Entitlements do not always need to grant additional privileges, and they can be used to specifically mark untrusted applications which should not be allowed to perform certain actions. These attributes make entitlements ideal for per-application access control decisions that do not need to change.

Weaknesses: In addition to their strengths, entitlements have various weaknesses to consider. We must trust that developers are not pretending to require more entitlements than their application genuinely needs. In an effort to simplify entitlement decisions for developers and reviewers, Apple might make entitlements excessively coarse grained. Users cannot see or modify the entitlements of the applications they install. The decentralized nature of checking entitlements in multiple processes creates more points of failure where mistakes in implementation can occur. On a jailbroken device, the integrity of code signing is sacrificed, and malicious applications are able modify their signatures with forged entitlements.

2.3.1 Policy

Entitlement policy specification happens in three locations, the code signature, ad hoc checks in service providing processes, and sandbox profiles. First, developers request entitlements within Xcode and apply them to their application as part of the build process. Second, processes offering services can be configured to require a client to possess certain entitlements in order to access the service. Third, a process's sandbox profile might allow it to perform certain actions if it possesses the relevant entitlements.

The entitlements embedded in the code signature can be found near the bottom of an app's executable as plain text strings starting with the magic number `0xFADE7171`. They are stored as key-value pairs in a dictionary format. For example, the entitlement's key could be `platform-application` with a value of `true`. Informally, we might say that a process with this dictionary in its signature has the platform-application entitlement. The value of an entitlement could also be a string (e.g., the identifier of the app) or an array of values.

Determining which entitlements are required for each service is specified in a decentralized fashion. Each process offering services must implement code that checks a client's entitlements before offering services. This decentralized system makes it difficult to verify that all sensitive services require relevant entitlements.

The sandbox profile also influences entitlement policies. As we will show in the next section, sandbox profiles can implement rules that are conditioned on the entitlements possessed by the sandboxed application. This allows multiple applications to share the same sandbox profile. For example, all third party applications and many system applications share a single sandbox profile called container. However, this generic sandbox profile uses each process's entitlements to determine which conditional rules to apply.

MetaPolicies: Some entitlement keys such as `tcc.allow`, `container-required`, and `seatbelt-profiles` act as metapolicies for other forms of access control. The `tcc.allow` key lets an application bypass the privacy settings system for resources specified by its entitlement value. For example, FaceTime can access the camera without permission from the user, and this access cannot be configured in the Privacy Settings menu. The `container-required` entitlement key forces a system application to use the container profile as its sandbox. The entitlement key, `seatbelt-profiles`, is paired with an entitlement value which specifies a sandbox profile to assign to the application.

2.3.2 Enforcement

Processes offering services, (e.g., Health Kit offers the user's health data) enforce entitlements by only providing certain services to processes with the required entitlements. We will refer to the process offering the service as the server, and the process receiving the service as the client. To the best of our knowledge, Levin [Lev] was the first to reverse engineer entitlement enforcement on iOS.

To check that a client possesses an entitlement, a server needs two things, an audit token representing the client and the key of the entitlement to check for. The client's audit token is provided through iOS's inter-process communication system, XPC. The key for the entitlement is simply a string such as `platform-application` in the previous example.

The server sends the audit token and entitlement key as parameters to the `SecTask` API in iOS's Security Framework. `SecTask` then makes system calls related to code signing that return the entitlement dictionary of the client. `SecTask` then finds the value in the entitlement dictionary corresponding to the key and returns this value to the server. If the key is not found in the dictionary, `SecTask` returns `NULL` to the server.

The server then allows or denies access to the service based on the value returned by `SecTask`. For example, a server could request a client's value for the key, `private.signing-identifier`, and only provide the service if the value returned is `mobilesafari`.

Entitlements can also be enforced by the sandbox mechanism through conditional rules. If a

sandbox rule requires the sandboxed process to possess a certain entitlement, the sandbox rule will only allow the operation if the entitlement is present.

2.4 Sandboxing

Sandboxes in iOS are access control mechanisms that allow or deny system calls made by the sandboxed process. The decision to allow or deny the operations is made by the sandbox kernel extension, which references a policy called a sandbox profile. Each sandboxed process is assigned one sandbox profile, but the same profile can be applied to multiple processes. For example, all third party applications and several system applications use a sandbox profile called container.

Strengths: Apple's sandbox mechanism considers the context of system calls. For example, a process might be allowed to read files in a specific directory as opposed to being allowed to read any file. The conditional rules in sandbox profiles make them somewhat flexible. In addition to entitlements, Apple implements unforgeable, sharable tokens called Sandbox Extensions that can be granted dynamically in order to satisfy conditional sandbox rules. Applications with different sets of entitlements and sandbox extensions can effectively have different privileges despite using the same sandbox profile. Sandbox profiles are compiled into a directed acyclic graph, which is allows efficient queries when the sandbox kernel extension needs to allow or deny system calls. As of iOS 9, Apple began storing sandbox profiles inside the kernel, which makes it more difficult for malicious processes to tamper with them.

Weaknesses: Using a generic sandbox profile (i.e., the container profile) for multiple processes can lead to accidentally providing dangerous privileges if the conditional rules are not configured correctly. It can be very difficult to make a sandbox profile more restrictive without breaking backward compatibility for existing applications. Apple does not provide a human readable format for built-in iOS sandbox profiles, which makes it more difficult for external security professionals to audit them. While the sandbox profile can specify file paths, it does not consider inode numbers. This may allow malware to bypass path based rules by abusing hard links (e.g., CVE-2015-7001), which allow access to the same file via different path names. Applications on jailbroken devices can still be sandboxed. However, the sandboxed application can forge entitlements to satisfy any conditional sandbox rules that grant additional privileges based on entitlements.

2.4.1 Policy

Sandbox profiles define the policy that determines which system calls will be allowed or denied by the sandbox enforcer. Sandbox profiles are written in a language called SandBox Profile Language (SBPL), which is derived from the functional programming language, Scheme. Once the profiles are written by a human in SBPL, they are compiled into binary blobs which represent the policies as

directed acyclic graphs for obfuscation and efficient queries. These binary blobs are stored in the sandbox kernel extension. Only the compiled sandbox profiles need to be present on the device. The human readable SBPL profiles are only known to Apple, and they are not stored on the iOS device. In Chapter 4 we present a sandbox decompiler that extracts profiles from iOS 7, 8, and 9[2]. We found 121 different sandbox profiles that had been compiled and stored in the kernel in iOS 9.3.1. In Chapter 4 we also develop a framework for automatically auditing iOS sandbox profiles.

Each process can have zero or one assigned sandbox profile. A process without a sandbox profile is considered unsandboxed, and its system calls will only be restrained by other access control permissions such as Unix file permissions.

SBPL profiles are composed of a list of rules. These rules consist of decisions, operations, filters, and metafilters. Decisions (i.e., allow or deny) determine the decision made if the rule is applied to the current system call. Each profile will have a default decision that determines whether to allow or deny any system calls that do not match a rule in the profile. Operations specify the system call or calls the rule applies to (e.g., file-read-data, file-link, network-outbound). Filters allow the rule to consider the context of the system call (e.g., a file path, a port number, or the name of a service requested). Metafilters act as logical operations on filters (i.e., logical NOT, logical AND, logical OR).

By using metafilters, one can configure a rule to require two filters to be matched, where one filter is a required capability, and the other filter is the protected resource. Such conditional rules allow a generic profile called container to be applied to all third party applications as well as several system applications (e.g., Apple Maps and Mobile Safari).

Some sandbox profiles allow the sandboxed process to grant tokens called sandbox extensions. These sandbox extension tokens are consumed by other sandboxed processes in order to satisfy conditional rules that require sandbox extensions. For example, the container profile requires the sandboxed process to obtain a specific sandbox extension before accessing the user's contact list. Aside from entitlements, conditional rules can also consider the sandbox extensions that have been granted to the sandboxed process. Note that sandbox extensions are an access control capability while app extensions are separate processes that augment applications. Consider the following simplified SBPL example:

```
(allow file-write*
  (require-all
    (subpath "/AddressBook")
    (extension "AddressBook")
))
```

In this example, `require-all` acts as a logical AND operator requiring both filters to be matched in order for the allow decision to be applied. In other words, files in the `/AddressBook` directory can only be written, if the sandboxed process possesses the `AddressBook` sandbox extension.

[2] recently expanded to process iOS 10 profiles

Which Sandbox?: Each sandbox profile can offer different sets of privileges under different conditions. Many of the iOS sandbox profiles are named after the daemons they were designed to regulate. We identify three methods that iOS uses to determine which sandbox profile to apply to a process: 1) The file path of the executable being launched (e.g., this is how the container profile is applied to third party apps); 2) Entitlements in the executable's signature; 3) The process assigns a sandbox to itself by calling a sandbox assignment function with a parameter indicating which sandbox profile to use.

2.4.2 Enforcement

Sandbox enforcement occurs in three stages. First, a sandbox profile is bound to a process during the process's launch phase or when the program calls a function to voluntarily sandbox itself. Second, the system calls made by the process are hooked by the TrustedBSD Mandatory Access Control (MAC) Framework. Third, the sandbox kernel extension, `Sandbox.kext`, will reference the relevant sandbox profile before allowing or denying the hooked system calls. For more information on iOS sandbox enforcement we direct the reader to the iOS Hacker's Handbook [Mil12].

A process can be forked and sandboxed before any of its code is executed. This allows iOS to sandbox untrusted code without risking the code from bypassing the sandbox application. Alternatively, some system processes apply sandboxes to themselved before processing any external input. Once a process has been sandboxed, it cannot remove the sandbox or apply a different sandbox profile to itself.

The TrustedBSD MAC (Mandatory Access Control) Framework enforces access control within the kernel. With TrustedBSD, some system calls can be checked for permission before being allowed, while others are allowed to skip the permission checks. For example, checking system calls made by non-sandboxed operations would incur an unnecessary performance cost. TrustedBSD hooks are distributed throughout the kernel for intercepting system calls that need to be evaluated by the sandbox kernel extension.

Figure 2.2 illustrates the role of the sandbox kernel extension, `Sandbox.kext`, in sandbox enforcement. In step 1, `Sandbox.kext` acts as a centralized authority that approves or disapproves the granting of sandbox extension tokens. This prevents processes from forging sandbox extension tokens or giving them to processes that should not receive them. In step 2, `Sandbox.kext` approves or denies hooked system calls by consulting the compiled decision tree representing the sandbox profile of the process making the call. In steps 3 and 4, `Sandbox.kext` also considers the sandbox extensions and entitlements of the process making the system call. Steps 5, 6, and 7 show that `Sandbox.kext` may use the same sandbox profile to evaluate multiple processes or it may reference a profile created for a specific process. Finally, in step 8, `Sandbox.kext` relies on `AppleMatch.kext` to evaluate regular expressions that appear in sandbox profiles. For example, the sandbox profile can

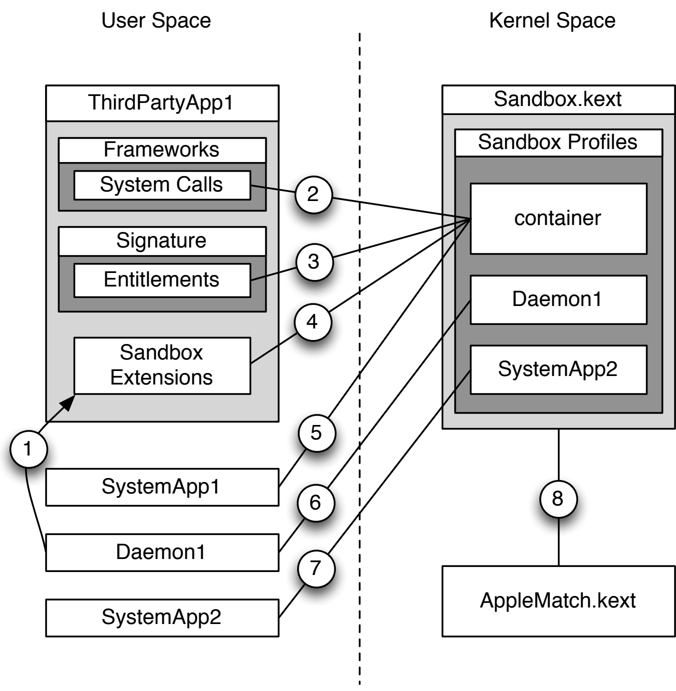

Figure 2.2 Sandbox Architecture

specify that read operations on file paths matching a given regular expression should be allowed.

XNU Components: In addition to BSD, the other components of XNU, Mach and I/O Kit, are evident in sandboxing. Mach-Lookups are IPC system calls from the Mach microkernel. Servers register services called mach-services that are associated with a string identifying the service. Clients then perform mach-lookups with specific strings to request access to mach-services. The sandbox profile determines which mach-lookups a process is allowed to perform. Even if a client can perform a mach-lookup, a server could deny the service based on other access control mechanisms discussed in this article (e.g., entitlements or privacy settings). I/O Kit clients provide interfaces between processes and drivers, but connecting to these interface is done through system calls that must be allowed by the sandbox. The role of XNU's components in iOS access control is covered in more detail by Watson [Wat13].

2.5 Privacy Settings

Privacy settings allow end-users to specify access control to privacy-related resources on a per-app basis. Privacy settings were originally introduced in iOS 6, and Apple has added additional options over time (e.g., controlling access to the camera and media). Unlike the aforementioned

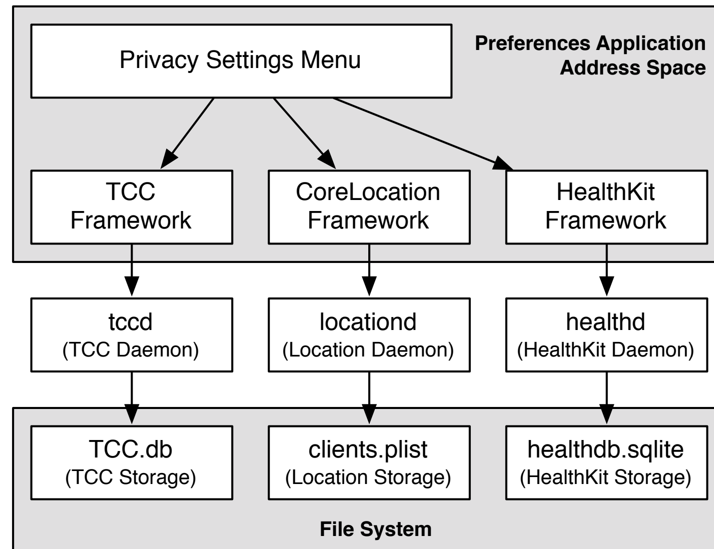

Figure 2.3 Privacy Settings Policy Architecture

entitlements, privacy settings can be changed at any time by the user.

Strengths: Privacy settings allow users to configure access to their private data at a user-friendly level of abstraction. Applications can continue to function even if the user revokes or never grants the privacy settings requested by the application. For example, Google Maps is less useful if it cannot track the user's location, but it can still be used to find directions between two locations. Applications are able to explain why they want access to private data through a prompt when they request a privacy setting. This promotes transparency between developers and users.

Weaknesses: In order to avoid overwelming users, privacy settings are coarse. For example, users can allow an application to access their list of contacts, but they cannot allow it to access only a subset of those contacts. The policies are stored in files that act as access control lists (ACLs). Sufficiently privileged processes that are not associated with privacy settings might still be able to write to these ACL files. For example, Dropbox on macOS has been shown to use root privileges to add itself to an ACL database to gain access to the Accessibility privacy setting without user permission [Dro]. Unfortunately, many users may agree to requests for privacy settings only because they are impatient or apathetic about privacy. There are entitlements that allow privacy setting enforcement to be bypassed by privileged applications (e.g., FaceTime does not need to request access to the camera). On a jailbroken device, applications can forge these entitlements to bypass privacy settings. Finally, it is difficult to fully revoke access to certain private resources. For example, a user can revoke an app's access to the user's photos, but the user cannot know if the application made copies of the photos while it had access.

Table 2.1 tccd managed privacy settings for iOS 8.

Resource	Service
Contacts	kTCCServiceAddressBook
Calendar	kTCCServiceCalendar
Contacts	kTCCServiceAddressBook
Reminders	kTCCServiceReminders
Photos	kTCCServicePhotos
Bluetooth	kTCCServiceBluetoothPeripheral
Microphone	kTCCServiceMicrophone
Camera	kTCCServiceCamera
HomeKit	kTCCServiceWillow
Twitter	kTCCServiceTwitter
Facebook	kTCCServiceFacebook
Sina Weibo	kTCCServiceSinaWeibo
Tencent Weibo	kTCCServiceTencentWeibo

2.5.1 Policy

The Settings (also known as Preferences) application collects all privacy setting configurations into a convenient user interface. However, the internal management of privacy setting policies is much less centralized. Most per-application privacy settings are managed by the TCC (Transparency Consent Control) daemon, but iOS uses separate policy managers for location and health data. We find that the three per-application privacy setting managers all use access control lists to define policies as shown in Figure 2.3.

Table 2.1 enumerates the different services managed by the TCC daemon, `tccd`, as of iOS 8. Note that the last two services (Sina Weibo and Tencent Weibo) are only available to iOS users that enable the Chinese keyboard. Policies for TCC managed privacy settings are stored in the `TCC.db` SQLite database file. The `access` table includes columns for 1) the name of the application, 2) the name of the service (shown in Table 2.1), 3) whether or not the user has been prompted for access, and 4) whether or not the application has access to the service.

Health privacy settings allow users to configure whether applications can read or write data regarding fitness activity, weight, or nutrient consumption. We speculate that this finer-grained configurability caused Apple to implement a separate privacy settings manager for health, rather than simply using TCC. The Health privacy settings management consists of: 1) the HealthKit Framework used to request access to information; 2) the health daemon (`healthd`), 3) the Health Privacy Service app which provides complex user prompts, and 4) the access control list (`healthdb.sqlite`).

Unlike `tccd` and `healthd`, the location daemon, `locationd`, does not reference policies in a database. Instead the policies are stored in the memory of `locationd`. `locationd` also writes the policies to the cache file `clients.plist`. When the device boots, `locationd` reads `clients.pl`

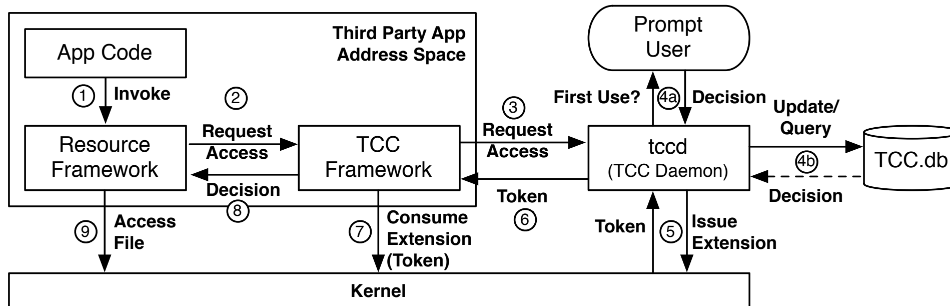

Figure 2.4 Requesting access to a file-based resource through TCC

ist. The `clients.plist` cache can be displayed as XML using the `plutil` utility. The file itself contains an entry for each application that uses location privacy settings. For each application, the file maintains an integer value that represents "Never", "Always", or "While Using the App".

2.5.2 Enforcement

Privacy settings use two forms of enforcement depending on the type of resource they protect. The photo album and address book are both files which are accessed via system calls. Therefore, the sandbox kernel extension enforces access control for the photo album and address book. The other privacy setting protected resources (e.g., camera, microphone, location) are services provided by servers (e.g., daemons). Therefore, these servers enforce privacy settings by only providing services to clients with appropriate entries in ACLs.

Figure 2.4 illustrates the role of sandbox extensions and `tccd` in enforcing a system call regulated by privacy settings. To provide a seamless user experience, sandbox extensions can be granted while the process is running. This allows the process to trigger a prompt to the user, and if the user accepts the prompt, the application gains a sandbox extension. The user's decision is stored in the ACL to avoid prompting again. The container profile can then provide access to the resource that required the extension. Unless the sandboxed process acquires the appropriate sandbox extension, system calls to access the private resource will be denied.

Other resources protected by privacy settings are services provided by servers (e.g., daemons). These servers enforce access control by only providing services to clients that have sufficient authorization according to ACLs. The control flow that allows an application to access a service protected by `tccd` is similar to the one shown in Figure 2.4. However, instead of interacting with the kernel, tccd informs the server if the client is sufficiently authorized to access the service. In this case, the servers are resource providing daemons (e.g., calendar access daemon). The servers load the TCC framework in order to query `tccd` when making enforcement decisions about a client. `tccd` then consults its ACL and tells the server whether or not it should provide the client with access to the

Table 2.2 TCC database experiments.

Row	Resource	Change	In Use	Observed Behavior
1	Camera	Denied	Yes	Camera continued to show live video and take photos.
2	Camera	Denied	No	Camera window appeared but only showed black screen.
3	Camera	Allowed	No	User presented with instructions to allow access.
4	Photo Album	Denied	Yes	Photos from album continued to be available.
5	Photo Album	Denied	No	Photos from album continued to be available.
6	Photo Album	Allowed	No	User presented with instructions to allow access.

resource. We assume this process is similar for enforcement provided by `locationd` and `healthd`.

Enforcement Conflicts: By directly modifying the ACL in `TCC.db` without restarting the affected application, we found that we could confuse various enforcement checks. These experiments should not be considered as attacks since privacy setting changes made in Settings do cause applications to restart. This experiment took place using version 47.0 of the "Messenger" application and Table 2.2 lists our observations. The "In Use" column indicates if we were using the resource at the time we modified the database (e.g., viewing photos or recording video). Based on these observations we make two conclusions. First, frameworks in the application's address space provide explanations to the user if they believe access will be denied. Second, sandbox extensions for privacy settings are not revoked until the application is restarted.

2.6 Conclusions

In this chapter, we have investigated internals of iOS access control mechanisms (i.e., entitlements, sandboxing, and privacy settings). We have explained where each access control policy is stored, and we have stepped through the mechanisms involved in enforcing those policies. The strengths and weaknesses of entitlements, sandboxing, and privacy settings were evaluated, and we have made suggestions for improving them. We showed that entitlements rely on code signing to provide immutable capabilities to processes. Users should consider this before disabling code signing enforcement via jailbreaking, which allows applications to forge entitlements. We found that iOS sandboxing provides efficient policy queries and flexible sandbox profiles for regulating system call access. Unfortunately, the obfuscation and reuse of policies in iOS sandboxing makes them more difficult for security researchers to audit. Finally, we showed that privacy settings provide users with flexibile access control through an intuitive interface. However, privacy settings are coarse grained and in danger of being overridden by applications developed by large corporations (e.g., Dropbox).

CHAPTER

3

RELATED WORK

This dissertation seeks to automate the evaluation of overlapping access control policies in iOS. Therefore, it builds upon prior work in three areas: 1) access control policy design; 2) access control policy evaluation; and 3) iOS security.

3.1 Access Control Policy Design

An access control policy determines whether some *subject* can perform some *action* on some *object*. In complex policies, this decision may also depend on the *context* of the system. Designing policies for complex systems can be a difficult and error prone process. However, several policy design techniques have been proposed that provide several useful tradeoffs.

Multi-Level Security (MLS) systems allow policies to define multiple levels of security for subjects, objects, and actions. In the Bell-Lapadula model [BL73], secrecy is preserved by preventing subjects from writing to any object with a secrecy label lower than the subject's label and preventing subjects from reading any objects with a secrecy label higher than the subjects label. This is colloquially known as "No read up. No write down". The Biba model [Bib77] preserves integrity by implementing complementary rules that prevent low integrity subjects from writing to high integrity objects, and prevent high integrity subjects from reading from low integrity objects. This is colloquially known as "No write up. No read down". While these policies are simple to design, they are often impractical to apply to modern systems in which many trusted and untrusted processes must interact.

Instead of defining specific security levels in order, access control policies can define rules for each subject and object. Lampson introduced the concept of an Access Control Matrix [Lam74], which represents the policy as a matrix in which each row represents a subject, each column represents an object, and each cell contains the actions the subject may perform on the object. Since these matrices may be very sparse, the same policy can be represented instead as lists. An Access Control List is attached to an object and lists the subjects that may access the object and the actions they can take on the object Alternatively, a capability list is attached to a subject, and lists the objects that subject may access and the actions the subject can take on them. iOS Entitlements are similar to capability lists in the way they are bound to the subject, but they do not directly contain the semantics of the actions and objects they allow access to.

Policies can be classified based on who can change them. A Mandatory Access Control (MAC) policy can only be modified by a system administrator. These policies are easier to the evaluate since unexpected policy changes are not possible, but MAC policies lack flexibility. A modern example of a MAC system is SELinux. SELinux is a modern example of a MAC system. A Discretionary Access Control (DAC) policy allows users to define the policy. This provides flexibility, but makes the system significantly more difficult to evaluate for safety [Har76]. An example of a DAC policy is Unix permissions which can be modified by subjects (e.g., chmod, chown). It is arguable whether the Apple Sandbox is MAC or DAC since it is built upon the TrustedBSD MAC framework, but it allows sandbox extensions to dynamically modify privileges.

Jaeger's book on operating system security[Jae08] defines several terms relevant to this thesis. A Protection System is a combination of a Protection State and Protection State Operations. A Protection State determines the operations that subjects are allowed to perform on each object. Protection State Operations are those operations that modify the Protection State. Protection State Operations are especially important to jailbreaks, which must modify several layers of access control. Finally, a Protection Domain is the set of operations a process is allowed to perform on system objects. In complex systems such as iOS, Protection Domains can be difficult to estimate because a process can be comprised of multiple subjects subject to multiple access control policies.

3.2 Access Control Policy Evaluation

Evaluation of non-trivial access control policies is a difficult, but essential process. Harrison et al. [Har76] introduced a concept known as the "safety problem", which queries if an unreliable subject can pass a right to a subject that did not already have that right. They also prove that resolving the "safety problem" for an arbitrary protection system is an undecidable problem. Despite this inherent challenge many techniques have been proposed for automating the evaluation of access control policies. Even if some policies cannot be proven secure, the insight provided by these tools can help identify vulnerabilities in policies.

Gokyo [Jae02] introduces a concept called access control spaces, which are comprised of subject's explicit permissions, constraints, and a set of "unknown" actions that are not listed among the constraints or permissions. The Gokyo tool allows automated detection of "unknown" or conflicting actions within an access control space.

VulSAN [Che09] allows for the comparison of policies from two different Linux security modules, SELinux and AppArmor. The comparison is based on the attack surface available to attackers in various configurations and the number of protection state transitions required to reach some attacker goal state. These protection state transitions are achieved when a process executes another process, and changes the protection state to adapt to the new process. VulSAN builds this model by converting the policies into Prolog facts, and modeling protection state transitions as edges between different policies.

Android shares many similarities with iOS as a modern, mobile operating system. However, as an open source system Android has attracted significantly more research including improvements on Android's access control policies. SELinux is a kernel security module that provides Mandatory Access Control for Linux, and it has been ported to Android in the form of SEAndroid [Sma]. EASEAndroid [Wan15] analyzes SEAndroid policies and uses semi-supervised machine learning to automatically refine those policies. SPOKE (SEAndroid Policy Knowledge Engine) [Wan17] uses dynamic testing to automatically identify unnjustified SEAndroid policy permissions that cause subjects to be over-privileged.

3.3 iOS Security

While much of the work on iOS security has been non-academic reverse engineering, there also exists a non-trivial amount of academic work.

3.3.1 Binary Analysis of iOS Apps

PiOS [Ege11] provides a binary analysis technique for detecting API usage in third party iOS applications. iOS applications use the objective-c runtime system which uses a message dispatch function to determine which methods of various objects will be executed at runtime. The behavior of these dispatch functions can be difficult to predict through static analysis, so it is difficult to build control flow graphs or even determine which API calls an app may make. PiOS begins static analysis at the call site of each dispatch function and uses backtracing to step through each previous instruction and eventually infer the values of registers at the time the dispatch function was called. Therefore, PiOS would often be able to statically predict the parameters of the dispatch function and the API call the dispatch function would invoke. PiOS was applied to detect the access of Private API calls, which are unsafe for a third party application to invoke directly.

Han et al. [Han13a] adopt the PiOS technique to perform a cross-platform analysis of applications that exist on both iOS and Android. Their analysis seeks to determine if applications on one platform access more security sensitive APIs than on the other platform. They find that iOS applications tend to access more security sensitive APIs, and they speculate that this trend may be related to a lack of transparency in the "permissions" that will be required by iOS applications.

iRiS [Den15] improves upon the PiOS technique by combining dynamic analysis techniques with static analysis. iRiS first analyzes applications statically and resolves a majority of API calls from dispatch calls. Then, dispatch calls that cannot be resolved statically are analyzed with iRiS's novel dynamic analysis technique. This dynamic analysis is implemented by porting[1] the Valgrind[2] memory analysis tool to iOS.

Chen et al. [Che16] also perform a cross platform analysis of third party libraries present on iOS and Android. Their insight is that third party libraries can be analyzed more effectively on Android, and can be matched to corresponding libraries on iOS. For example, library `A` might be found to be potentially harmful on Android, so any iOS application that contains the iOS version of library `A` contains a potentially harmful library. Therefore, Chen et al. demonstrate that similarities between the two platforms (especially the app markets) can be effectively leveraged by researchers.

3.3.2 Exploits for Third Party Apps

Several iOS exploitation techniques have been proposed that motivate our investigation into access control. Improved iOS access control policies can mitigate the damage caused by the following attacks

As discussed in Section 2.1, iOS implements several code security mechanisms including Data Execution Prevention (DEP) and code signing. This prevents attackers from modifying code or injecting new code. However, Wang et al. [Wan13] propose a technique for creating malicious applications called Jekyll apps. These applications have benign behavior during the app vetting process. However, after publication to the App Store, an attacker can exploit preconfigured vulnerabilities in the Jekyll apps to deploy Return-Oriented Programming (ROP) attacks. ROP allows the attacker to maliciously modify the application's behavior without violating DEP or code signing. The modified application could then call Private APIs that allowed the application to access resources that could potentially harm the user (e.g., abusing the camera, sending SMS messages, stealing device information).

Han et al. [Han13b] discovered an alternative method for accessing Private APIs and performing similar attacks as those demonstrated in Jekyll apps. Instead of ROP attacks, they were able to use simple obfuscation techniques to bypass the app vetting process, and then used function calls to dynamically load private frameworks and invoke private APIs. They used two strategies to

[1] https://github.com/tyrael9/valgrind-ios
[2] http://valgrind.org/

accomplish this. First, they use functionality in the objective-c runtime similar to Java reflection to pass a string as a parameter to a functions and have the functions dynamically return objects or selectors, which can represent methods in object. These objects and selectors are then used to dispatch messages to objects and could cause them to make private API calls. Second, they use the `d lopen` function call to load a private library based on a string parameter passed to the function. Next, they use the `dlsym` function to pass a string as a parameter and have the function return a function pointer for a private API call, which they then invoke. The actual strings used in the parameters for these attacks can be obfuscated to resist static analysis or dependent of hidden triggers to resist dynamic analysis.

Xing et al. [Xin15] investigate cross-app resource attacks that they refer to as XARA. They investigate such attacks on macOS and iOS. Although the majority of their findings are specific to macOS, they did discover URL Scheme vulnerabilities on iOS. A URL Scheme allows an application to open another application with some context. For example, `mailto:someemail@example.com` would open an application that had registered the `mailto` scheme, and send it the specified email address. Xing et al. demonstrate that an iOS application could register duplicate URL Schemes to intercept messages intended for other applications that had registered the same schemes. Apple's official stance on duplicate URL Schemes in multiple applications is as follows: "If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme."[3]

3.3.3 In-Line Reference Monitors for Third Party Apps

The closed source nature of iOS makes it difficult, but not impossible, for researchers to demonstrate the effectiveness of proposed defenses. Several solutions propose the use of in-line reference monitors to detect and prevent suspicious behavior in third party apps. Unfortunately for iOS users, these techniques have not been adopted by Apple, and suffer from limitations that may make deployment impractical.

A potential defense against ROP and other control-flow attacks is Control-Flow Integrity (CFI) enforcement. CFI prevents control from flowing to unexpected locations that would allow an attacker to modify program behavior. MoCFI [Dav12] provides a CFI implementation for ARM-based devices including iOS and Android. Unfortunately for iOS users, MoCFI was not officially adopted by Apple. Therefore, iOS users are required to jailbreak their devices if they would like to install MoCFI. However, jailbreaking the device is likely to sacrifice several security features such as code signing enforcement.

PSiOS [Wer13] is an In-Line Reference Monitor built upon MoCFI. PSiOS allows users to specify access control policies for which API calls that application should be allowed to make. It also

[3]https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html

provides the control flow integrity benefits offered by MoCFI. However, PSiOS suffers from the same deployment issues as MoCFI and requires a jailbreak in order to be used. It may also be difficult for users to define good access control policies without understanding the requirements of closed-source application or the functionality of undocumented API calls. Finally, Apple independently resolved some of PSiOS's motivating issues through the implementation of Privacy Settings, which allow users to define coarse-grained access control policies for some types of private data.

XiOS [Buc15] addresses the proposed exploits for abusing access to private APIs, by demonstrating a new attack with reduced complexity, and by introducing a defense based on an in-line reference monitor. The new attack exploits patterns in Apple's default memory layout. The inline reference monitor is applied by rewriting application binaries such that API calls are wrapped by security logic that should prevent direct access of private APIs. XiOS does not require the user to have a jailbroken iOS device, but rewriting app binaries does invalidate code signatures. While the applications can be resigned by enterprise certificates, this requirement makes wide scale deployment of XiOS impractical. There are also limitations regarding semi-private entitlements. The original app vendor may have been authorized to sign it with a semi-private entitlement, but a user applying XiOS to the app may not be authorized to sign the app again with the same semi-private entitlement.

CHAPTER

4

SANDSCOUT: AUTOMATIC DETECTION OF FLAWS IN IOS SANDBOX PROFILES

4.1 Introduction

The sale of smartphones has out-paced the sale of PCs [Sal]. The two dominant platforms for these smart phones are Android and iOS [Mar]. There has been a significant amount of academic research on Android, in part, because of its open-source nature. In contrast, iOS is not open-source, and studies of iOS may require significant reverse engineering effort.

Prior research on iOS security has focused on the following three areas. First, works have demonstrated methods for creating iOS malware [Wan13; Buc15; Han13b; Xin15; Kur16; Wan14]. Second, others emphasize methods to detect malicious behavior either statically [Ege11] or dynamically [Den15]. Third, new security mechanisms [Buc15; Dav12; Wer13] have been proposed that hook into application code to provide additional security. All of these works rely on interacting with the code of third-party iOS applications.

We investigate something different: iOS sandbox profiles. These sandbox profiles define access control policies for system calls made by processes. There are 117 sandbox profiles in the iOS 9.0.2 kernel, and many system daemons and applications have dedicated profiles. However, all third-party applications, and some system applications, are confined using the shared "container" sandbox profile. The container sandbox profile is large and complex, leading to the research question: *what*

flaws in the container sandbox profile can third-party iOS applications exploit?

Goals and Contributions: In this paper, we present the SandScout framework to answer this research question. First, we create a tool, SandBlaster, which automatically extracts compiled profiles from a firmware image and decompiles them into their original SandBox Profile Language (SBPL). Second, we formally model sandbox profiles using Prolog by creating a compiler that automatically translates SBPL policies into Prolog facts. Third, we develop Prolog queries that test critical security properties of the container sandbox policy. The queries identify potential security vulnerabilities in the policy. Finally, we create an iOS application that provides assisted verification of potential vulnerabilities on iOS devices.

We use SandScout to evaluate the container sandbox profile for iOS 9.0.2. Sandbox profiles mediate all system calls including file access and inter-process communication (IPC). For this evaluation, we limit our security queries to file-based sandbox policy rules for two reasons. First, non-file-based sandbox policy rules require additional semantics that are not available in the policy. Second, we find significant security vulnerabilities within the file-based sandbox policy rules. We plan to expand our analysis to non-file-based policy rules in future work.

Our analysis of the file-based policy rules in the iOS 9.0.2 container sandbox profile identified seven broad vulnerabilities that are exploitable by third-party applications: (1) methods of bypassing iOS's privacy settings for Contacts; (2) methods of learning a user's location search history; (3) methods of inferring sensitive information by accessing metadata of system files; (4) methods of obtaining the user's name and media library; (5) methods of consuming disk storage space that cannot be recovered by uninstalling the malicious app; (6) methods of preventing access to system resources such as the AddressBook; (7) methods for colluding applications to communicate without using iOS sanctioned IPC. We have reported all of these vulnerabilities to Apple and are working with them to ensure they are fixed in future versions of iOS.

This paper makes the following contributions:

- *We develop the first methods to automatically produce human readable SBPL policies.* Prior work was unable to produce SBPL policies for human review or automated analysis. Our tool extracts and decompiles all sandbox profiles in firmwares for iOS 7, 8, and 9.

- *We formally model SBPL policies using Prolog.* We create an SBPL to Prolog compiler based on a context free grammar we have defined for SBPL.

- *We perform the first systematic evaluation of the container sandbox profile for recent versions of iOS and discover vulnerabilities.* We develop Prolog queries representing security requirements. When applied to the iOS 9.0.2 container sandbox profile, we discover seven classes of security vulnerabilities.

The remainder of the paper proceeds as follows. Section 4.2 provides background information.

Section 4.3 provides an overview of SandScout. Section 4.4 discusses our design. Section 4.5 presents our results. Section 4.6 provides discussion of our limitations. Section 4.7 presents related work. Section 4.8 concludes.

4.2 Background

iOS is the operating system of the iPhone, iPod, iPad, and older versions of AppleTV (newer AppleTV devices run TVOS). iOS is based largely on Apple's desktop operating system, OS X, and the two share many internal similarities.

4.2.1 iOS Security Mechanisms

iOS relies on four broad types of security mechanisms: *application vetting*, *code signing*, *memory protection*, and *sandboxing*. When developers submit an application to the App Store [Appc] for vetting, they sign the application using their developer key. While the specific details of the vetting process are only known to Apple, it is assumed that they use a combination of static and dynamic analysis to detect malicious behavior. If Apple approves of the application, it adds its own signature to the application and makes the application available on the App Store.

An iOS device will only execute code pages coming from binaries with valid signatures. Generally, having a valid signature means the application is signed by Apple. However, devices provisioned for developers or enterprises may also run applications signed by specific developer and enterprise keys. Finally, immutable capabilities called entitlements are stored inside an application's signature. Apple grants developers a certificate that determines which entitlements they may apply to their applications.

In addition to code signing, iOS uses data execution prevention (DEP) and address space layout randomization (ASLR) to mitigate memory attacks. DEP prevents code injection attacks by ensuring that no code page is writable and executable at the same time. ASLR mitigates code-reuse attacks by randomizing code and data segments in memory. Interestingly, code signing complicates the ASLR design and limits its protection, because shuffling code regions may invalidate signatures [Essa]. Prior work [Wan13; Buc15; Mil12; Han13c] has demonstrated several techniques for bypassing application vetting and memory protections.

iOS sandboxes all applications using a mandatory access control policy to limit the abilities of exploited or malicious code. Sandbox policies are enforced by the Trusted BSD mandatory access control framework [Tru] using a kernel extension called `Sandbox.kext`. iOS uses different sandbox policies (called *profiles*) for different applications. Many system applications and daemons have their own profile. However, all third-party applications are controlled by a generic sandbox profile called *container*. The container sandbox profile is also used by several system applications. In

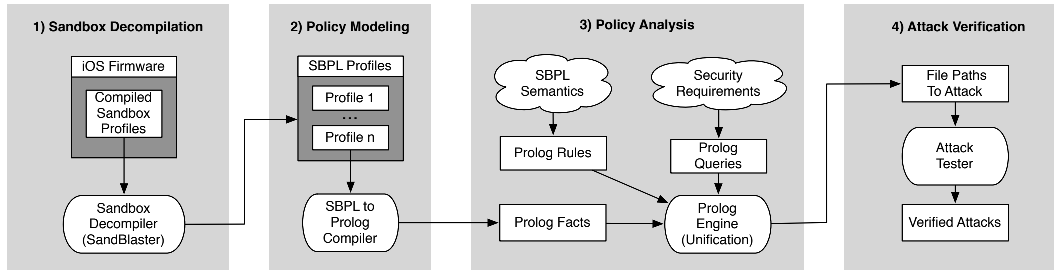

Figure 4.1 Overview of SandScout.

order to support the functionality of many different applications, it is the largest and most complex sandbox profile.

Sandbox profile rules define access to system calls (e.g., file read and write). To be generic, the container sandbox profile uses conditional rules that may require capabilities. There are two primary types of capability considered by the sandbox: *entitlements* and *sandbox extensions*. Mentioned above, entitlements are static capabilities assigned by application's developer during development. Entitlements are key-value pairs, which are stored in a dictionary structure embedded in an application's code signature. Note that entitlement keys are not cryptographic keys, and they simply map to values in the entitlement dictionary. Once the application has been signed, its entitlements cannot be modified without invalidating the signature. In contrast, sandbox extensions are dynamic capabilities that can be granted or revoked at run time. System daemons such as the `tccd` daemon, which helps enforce iOS's user specified Privacy Settings, can grant sandbox extensions.

Finally, while the vast majority of iOS's access control policy is enforced in `Sandbox.kext` using sandbox profiles, there are various access control checks within system daemons. These system daemons maintain their own policies based on user preferences (e.g., for Privacy Settings) and entitlements. In this paper, we limit our investigation to the sandbox profiles and leave these other daemon specific access control policies to future work.

4.2.2 Sandbox Profile Language (SBPL)

Sandbox profiles are written in the SandBox Profile Language (SBPL), which is derived from Scheme. Sandbox profiles are compiled from SBPL into binary blobs that are structured as graphs for efficient queries.

An SBPL sandbox profile consists of a version indicator, a default decision, and 0 or more rules. Sandbox rules can allow or deny access to system calls based on capabilities held by the sandboxed process. A default decision (i.e., deny or allow) defines the decision to make if no sandbox rule matches the evaluated system call. The container profile and the majority of sandbox profiles we have encountered are whitelists that deny by default. Therefore, SBPL examples provided in this

paper assume a default deny policy.

Each sandbox rule consists of a decision (i.e., `allow` or `deny`), an operation (e.g., `file-read-data` or `file-write-create`), and 0 or more filters and metafilters.

Filters: A filter considers the context of the system call and consists of a filter-type and 0 or more filter-values. A filter-type indicates the filter's type (e.g., `subpath`, `literal`, or `regex`). Filter-values represent parameters for the filter-type (e.g., a string indicating a file path).

Metafilters: Metafilters act as logical operations on filters. There are three types of explicit metafilters: `require-all` requires all of its filters to be satisfied (i.e., logical AND); `require-any` requires any of its filters to be satisfied (i.e., logical OR); and `require-not` requires that the filters not be satisfied (i.e., logical NOT). Metafilters can be, and frequently are, nested.

Some filters imply the use of metafilters. The `regex` filter implies a `require-any` metafilter applied to a list of one or more reqular expressions that act as its filter values. The `require-entitlement` filter implies a `require-all` metafilter applied to an entitlement key and the `entitlement-value` filter. The `entitlement-value` filter may also have metafilters applied to it, but these are explicitly stated. Note that the entitlements of a process are stored as key-value pairs in a dictionary structure, so all entitlements have both keys and values. Section 4.4.2 discusses our context free grammar for parsing the SBPL language.

Ideally, a sandbox profile should allow only those privileges a process requires. The container sandbox profile provides flexibility by using metafilters that only provide privileges if a process has required capabilities. Consider the following example SBPL rule:

```
( allow file-read*
  ( require-all
    ( subpath "/Media/Safari" )
    ( require-not
      ( literal "/Media/Safari/secret.txt" )
    )
    ( require-entitlement
      "private.signing-identifier"
      ( require-any
        (entitlement-value "mobilesafari" )
        (entitlement-value "safarifetcherd" )
) ) ) )
```

This rule allows the sandboxed process to read any file other than `secret.txt` in the `/Media/Safari/` subpath, if that process has the required capabilities. In this case, the required entitlement key is `"private.signing-identifier"`, and the entitlement value must be either `"mobilesafari"` or `"safarifetcherd"`. In other words, this rule states that the Mobile Safari app or the `safarifetcherd` daemon can read files other than `secret.txt` in the `/Media/Safari/` directory.

In addition to immutable entitlements, iOS uses dynamic capabilities called *sandbox extensions*, which can be granted and revoked at runtime. The sandbox profile can include conditional rules that require sandbox extensions with syntax similar to the example for shown for entitlements.

4.3 Overview

Apple's application review process is not infallible [Wan13; Buc15; Mil12; Han13c; Byf15; Xia; Pir; Ace; Sna]. The iOS container profile is designed to protect against abuse by third-party applications. However, little is known about the actual policy it enforces. Least privilege sandbox policies are difficult to define correctly [Ala08; ZM04; Sas06; Wan15; Jae03; Hic10a]. Therefore, in this paper, we ask the overarching research question, *what flaws in the container sandbox profile can third-party iOS applications exploit?* That is, we seek to systematically identify vulnerabilities in the container profile. Answering this question requires addressing the following challenges:

- *Sandbox policy extraction.* Built-in sandbox policies are stored in binary form as precompiled graphs. Apple sometimes changes the location and structure of these built-in profiles in updates to iOS. We were unable to find any sandbox decompilation tools that decompiled sandbox profiles into SBPL.

- *Modeling sandbox policy semantics.* The SandBox Policy Language is not officially documented and must be reverse engineered. Unofficial documentation of SBPL [fG!; Mil12], is outdated and only documents a minority of the sandbox operations available for iOS.

- *Automated discovery and verification of potential vulnerabilities.* We first need to understand the mistakes and misconfigurations made by developers working on the Apple sandbox. Then we must define heuristics to detect these misconfigurations. Finally, we must evaluate the consequences of abusing potential misconfigurations detected by these heuristics.

SandScout addresses these challenges in four parts as shown in Figure 4.1. First, we created SandBlaster to automatically extract sandbox profiles from iOS firmware images and decompile them. Second, we created an SBPL to Prolog compiler to automatically convert sandbox profiles into collections of Prolog facts. Third, we model security requirements as Prolog queries to systematically discover facts that violate those requirements. Fourth, we have semi-automated the attack verification process.

We chose to use Prolog for three reasons. First, Prolog was used for evaluation of security policies in prior work [Enc08; Che09]. Second, Prolog is capable of handling the regular expressions that can appear in Apple sandbox profiles. Third, Prolog is sufficiently extensible for incorporating additional iOS security mechanisms in future work.

Our analysis focuses on the container profile because it sandboxes third party applications, for which we can construct a common set of security requirements. SandScout can also analyze the other sandbox profiles extracted from iOS; however since they are primarily used for trusted system apps, the threat model is different.

(1) Sandbox Decompilation: To extract and decompile sandbox profiles, we created SandBlaster. SandBlaster decompiles sandbox profiles directly from iOS firmware images, which can be downloaded directly from Apple [Ios]. SandBlaster is the first tool to fully decompile sandbox profiles for iOS 7, 8, and 9 into human readable and compilable SandBox Profile Language (SBPL). While Blazakis [Bla] previously created a sandbox profile decompiler, his tool cannot decompile modern iOS sandbox profiles (i.e., iOS 7, 8, and 9). Esser [Essb] also created a sandbox analysis tool, but it only produces intermediate information (i.e., graphs), which are insufficient for our analysis.

We chose to work with decompiled SBPL profiles for three reasons. First, we want Apple to be able to use the original SBPL profiles as input to our system. Second, understanding SBPL profiles provides insight into how developers might make mistakes. Third, the ability to modify and run our decompiled SBPL profiles helped us test our results and reverse engineer SBPL semantics.

(2) Policy Modeling: We model iOS sandbox profiles as collections of Prolog facts. We created a compiler, which uses a context free grammar to automatically parse and recursively translate SBPL into Prolog facts. Nesting of metafilters as shown in Section 4.2 makes converting from SBPL to Prolog nontrivial. We handle combinations of logical ANDs and logical ORs by formatting the Prolog facts in disjunctive normal form.

(3) Policy Analysis: We model the security requirements of stakeholders as Prolog queries. The queries discussed in this paper are not intended to be comprehensive, but they provide practical demonstrations of the flaws SandScout can detect. The following is an example of a security requirement: *No third-party application should have direct write access to system files.* A query representing this requirement may match harmless sandbox rules (e.g., write access to `/dev/null`), but we demonstrate that it can also detect significant vulnerabilities. SandScout is extensible and can process more queries than those demonstrated in this paper.

We model profile-independent semantics of SBPL as Prolog rules. For example, the knowledge that `file-read*` access implies `file-read-data` and `file-read-metadata` can be represented as a Prolog rule. Since these Prolog rules are true for every SBPL profile, they only need to be defined once. Note that sandbox rules and Prolog rules are not the same thing. A sandbox rule allows or denies an operation for a given set of filters. A Prolog rule is a clause that represents a logical relationship between Prolog facts.

(4) Attack Verification: To remove any false positives produced by our queries, we have created an iOS application for testing attacks that abuse sandbox misconfigurations. This app implements a collection of Objective-C functions for testing operations on file paths (e.g., moving files, querying

databases, creating hard links, etc.). The application also includes functions that perform more complex attacks such as copying a given number of 10 MB files to a given directory in order to consume storage space. The app reports on which attacks are successful and outputs error messages for those attacks that fail.

Summary of Findings: We have used SandScout to evaluate the container sandbox profile from iOS 9.0.2. SandScout detected sandbox rules vulnerable to seven attacks. Each of these attacks has been disclosed to Apple, and has been successfully tested on iOS 9.3.1 (Latest version at the time of experiments). These attacks can be more broadly categorized as follows.

- *Bypassing Privacy Settings:* By creating a hard link to the AddressBook database while an app has access to it, that app can keep access even after the user revokes access through Privacy Settings. The app can place the hard link into a directory accessible to other apps that have never been granted access to the `AddressBook` through Privacy Settings.

- *Privacy Leaks:* We have identified several system files containing sensitive user data that the container profile allows third-party applications to read. These unprotected files contain information on the user's location search history, media contents, the user's name, and the names of computers that have synced to the device. Third-party apps can also read the metadata of all directory files and learn potentially sensitive information about the user and the device. We also identify 4 file paths that are both readable and writable to third-party apps, which allows applications to easily leak information to other apps.

- *System Damage:* Third-party apps can abuse write access to system files by deleting, moving, or changing permissions to prevent legitimate access to these files. These apps can also consume all storage space on the hard drive by creating new system files or appending data to existing ones. This storage space is not released by uninstalling the third-party app nor does it appear in the Storage Manager as being used by the app.

4.4 Design

SandScout detects attacks against iOS sandbox profile vulnerabilities in four steps. First, we automatically decompile the sandbox profiles with our tool, SandBlaster. Second, we use our SBPL to Prolog compiler to automatically model the decompiled sandbox profiles as Prolog facts. Third, we use Prolog rules and queries to automatically detect sandbox misconfigurations that violate security requirements. Fourth, we use our attack testing application to evaluate the consequences of abusing these misconfigurations.

4.4.1 Decompiling Sandbox Profiles

As discussed in Section 4.2, sandbox profiles are written in the SandBox Profile Language (SBPL) and compiled into binary blobs representing graphs used to rapidly query the policies they define. Our tool, SandBlaster, extracts and decompiles sandbox profiles from this compiled format back into their original language.

Note that SandBlaster expands upon the work of Blazakis [Bla11] and Esser [Essb]. Distinctions between SandBlaster and prior work are discussed in Section 4.7. A full, technical description of SandBlaster is available in our technical report [Dea16]. Here, we limit our description to the key novel contributions of the tool.

We use a combination of our own scripts, existing tools [Dmg; Dsc; Lzs; Jok; Vfd], and information shared by reverse engineers [Bla11; Essb] to perform the initial steps of sandbox profile extraction. This process consists of decrypting iOS firmware, extracting binary profiles, and processing filter types and filter values.

The novelty of SandBlaster is the conversion of the graph structure of a compiled sandbox profile into valid, human readable SBPL. This conversion requires reconstructing graph connections into their respective metafilter components. Within the compiled sandbox profile, each sandbox operation (e.g., `file-read-data`) is represented by a directed acyclic graph. This graph contains two terminal nodes for the `allow` and `deny` decisions. Nonterminal nodes represent filters (e.g., `literal"/var/myFile"`). Edges represent the presence or absence of metafilters (e.g., `require-all`). An example graph is shown in step 1 of Figure 4.2.

Each nonterminal node has two edges: `match` and `unmatch`. The `match` edge is followed when the filter is matched, and the `unmatch` edge is followed if the filter is not matched. The decision to allow or deny a system call is made based on the terminal node reached after traversing the graph. In a default deny profile, a `match` edge connecting to `allow` and an `unmatch` edge connecting to `deny` means the node has no metafilters. In this case, if the filter the node represents is matched, the operation is allowed. However, other connections for the `match` and `unmatch` edges can represent various metafilters. Table 4.1 shows all possible `match` and `unmatch` combinations and the logical equivalent of the relevant metafilters. Note that metafilters can be nested, so the value of `other` is evaluated recursively when it appears.

Graph to SBPL Demonstration: We use Table 4.1 to explain each of the four steps in converting the graph in Figure 4.2 into SBPL. Note that in these graphs, a solid line represents a `match` edge, and a dotted line represents an `unmatch` edge. 1) Node `B` moves to deny on a `match` and allow on an `unmatch`, so we can apply the `require-not` metafilter to `B`. Negation has the effect of swapping a node's `match` and `unmatch` edges. 2) Node `C` moves to `allow` on a `match` and a nonterminal on an `unmatch`, so we can apply `require-any` to `C` and the nonterminal. In other words, if `C`'s filter does not match, then we should attempt to match the other nonterminal's filter before denying the system

Table 4.1 Logic for Match/Unmatch Edges*

Match	Unmatch	Logical Equivalent
allow	deny	$self$
deny	allow	$\neg self$
allow	non-terminal	$self \vee other$
non-terminal	allow	$\neg self \vee other$
non-terminal	deny	$self \wedge other$
deny	non-terminal	$\neg self \wedge other$
non-terminal1	non-terminal2	$(self \wedge other1) \vee$ $(\neg self \wedge other2)$

* Assuming a default deny sandbox profile

call. 3) Node `A` moves to a nonterminal on a `match` and `deny` on an `unmatch`, so we can apply `require-all` to `A` and the nonterminal. In other words, if `A`'s filter does not match, then we should deny the system call and not attempt to match the other nonterminal's filter. 4) Finally the result of processing and merging all nonterminal nodes is a collection of SBPL metafilters applied to filters.

4.4.2 Modeling Sandbox Profiles in Prolog

SandScout uses Prolog to analyze iOS sandbox profiles. Each sandbox profile rule is converted into a collection of Prolog facts defined as follows:

```
decision(operation,[listOfFilters]).
```

Recall that each sandbox profile rule may contain many levels of nested metafilters. Instead of encoding the nested metafilter logic directly in Prolog, we first expand the boolean equation into disjunctive normal form (DNF). The DNF form lends itself nicely to encoding the logic in Prolog. In the above Prolog fact template, `[listOfFilters]` represents the conjunction of list elements (i.e., `require-all`), and multiple Prolog facts represent the disjunction of those conjunctions (i.e., `require-any`). Finally, negation is represented by a Prolog functor applied to a filter.

The following is an example of Prolog facts from a deny default profile. Here, the process gains `file-read*` access to `/myFile` if it has the `A` extension or does not have the `B` extension.

```
allow(file-readSTAR,
  [literal("/myFile"),extension("A")]).
allow(file-readSTAR,
  [literal("/myFile"),not(extension("B"))]).
```

Converting SBPL into such Prolog facts is non-trivial for three reasons. First, each filter may require a different set of filter values. Second, metafilters can be nested indefinitely and the `regex` and `require-entitlement` filters have implied metafilters. Third, we must output our Prolog facts

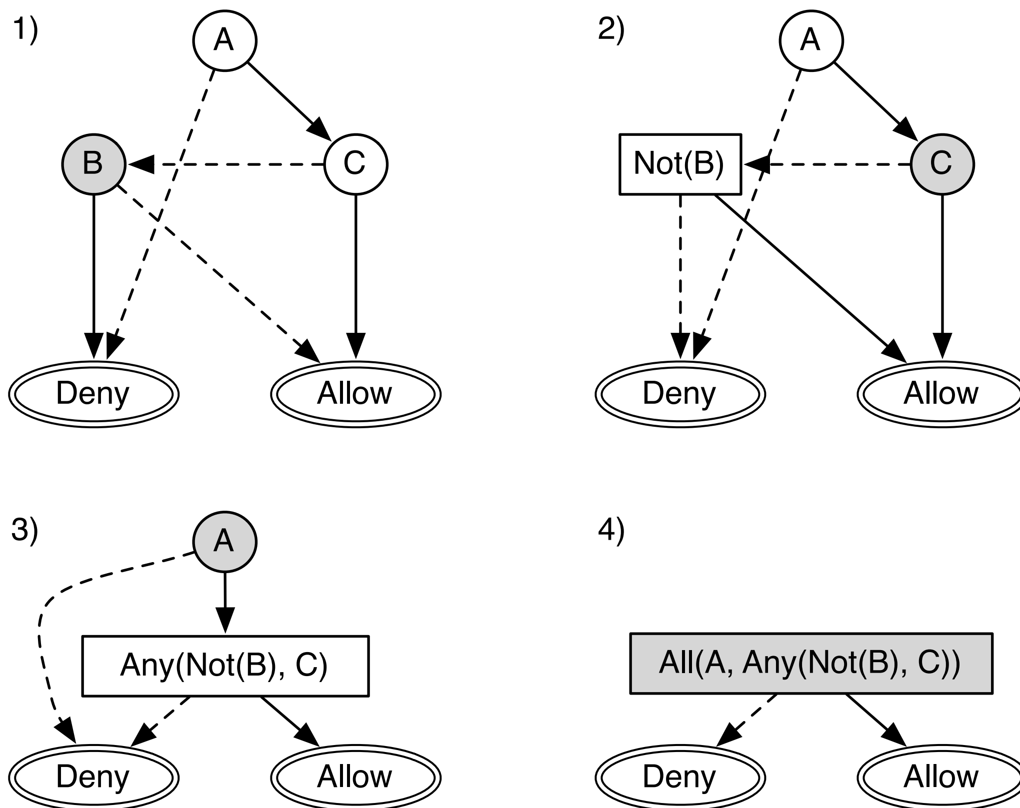

Figure 4.2 Converting Graph to SBPL.

in disjunctive normal form. To address these challenges, we created an SBPL to Prolog compiler using the ply [Ply] Python library for Lex and Yacc.

Lex uses regular expressions to tokenize an input. This allows us to match reserve words and distinguish between types (e.g., strings, regular expressions, and booleans). A simplified list of SBPL token definitions is provided in Figure 4.3. A more complete listing would include many more reserve words. However, for the sake of our compiler, it was sufficient to match most reserve words with the TK_VARIABLE token.

Yacc uses a context free grammar that recursively processes a tokenized input. The grammar we defined for SBPL is presented in Figure 4.3. Our implementation assumes a correctly written SBPL profile is taken as input. However, the implementation could be expanded to detect additional syntactic or semantic errors in SBPL profiles. Our current grammar allows us to recognize and process metafilters and implied metafilters. For example, when Yacc detects a `requireAll` expression we appropriately process the results of the `objList` expression inside it.

While Yacc distinguishes metafilters, we must produce our output in DNF. Our algorithm produces DNF by processing a list of strings for each sandbox rule. For `requireAll` expressions, we append all elements of the processed `objList` to each string in our list. For `requireAny` expressions, we create a new string for each element of the processed `objList`. Each new string is prepended with existing strings from our list of strings on the list. When a `require-not` metafilter is detected, we apply the `not` Prolog functor to the object inside the `require-not`. Our grammar asssumes that `require-not` metafilters will not contain other metafilters, however, they may contain implied metafilters. If the object inside the `require-not` is a `regex` filter, we must process the implied `require-any` metafilter. To do this, we use De Morgan's laws and treat the result as a `require-all` metafilter applied to each negated `regex` filter.

We attempt to preserve as much similarity as possible when converting SBPL to Prolog, but some characters could not be preserved. For example, the `file-write*` operation must be converted to `file-writeSTAR` in Prolog because Prolog does not recognize '*' as part of a functor name.

4.4.3 Policy Analysis

SandScout analyzes sandbox profiles using Prolog queries. In this subsection, we describe how to construct useful queries. Doing so also requires defining a collection of Prolog rules that model SBPL semantics (e.g., `file-readSTAR` is one of the read operations). Finally, we describe the three queries used for our vulnerability evaluation in Section 4.5.

Note that our current queries are limited to file access operations on the container sandbox profile. The complete information for file access control policy is available within the policy itself. Other policies (e.g., those that protect inter-process services and driver services) require a deeper understanding of the semantic operations within system processes. We plan to build automated program analysis tools to consider these semantics in future work. Furthermore, since we found the file access control operations to contain a significant number of vulnerabilities, we limit this paper to those operations. Finally, while SandScout can process other sandbox profiles, we chose to focus on the container profile because it is shared by all third-party applications and hence provides the greatest attack surface.

4.4.3.1 Modeling SBPL and iOS Semantics

To effectively query the Prolog version of a sandbox profile, we must first encode additional semantics of SBPL and iOS. We accomplish this using additional Prolog rules.

The first type of Prolog rules we define address areas where file access filters overlap. For example, `subpath("/var")` and `literal("/var/myFile")` will both match `/var/myFile`. We use this technique to limit one of our queries to those files in `/private/var/mobile/` which are more likely to contain user data than other system files. Our ability to detect overlaps for regular expressions

is limited. We use the regex [Pro] library package for SWI-Prolog [Swi] to determine if a `literal` file path satisfies a regular expression. However determining if two regular expressions or a regular expression and a subpath overlap is more complex. If Prolog is provided with a finite set of literal file paths to test, this can be accomplished, but it is not part of our current implementation.

The container sandbox profile is used to confine a variety of applications that may be assigned different capabilities. Therefore, it is desirable to ask queries from different environment settings. For example, we can define Prolog rules for describing the set of all capabilities, system capabilities, or capabilities of third party applications. Note that entitlements and extensions are capabilities a process may possess, but some capabilities are only available to system applications.

We found that all sandbox rules providing access to third party directories required the `sandbox.container` extension. Note that all third party applications have dedicated directories where they may read and write private files. We consider files outside of these dedicated third party directories to be system files.

Finally, we encountered a special filter called `vnode-type`. This filter matches any file that has the type specified by the `vnode-type`'s filter value (e.g., `vnode-type(directory)`). Therefore, this filter has the potential to match files regardless of their file path, and it should be considered when making queries.

To encode the semantics of the iOS environment, we provide several predefined lists. Note that the '_' character in prolog will match any value. `Caps` is the list of all capabilities (i.e., `[extension(_),entitlement(_)]`). `SysCaps` is the list of all capabilities reserved for system applications. `Files` is the list of all filters that match file paths (i.e., `[literal(_),subpath(_),regex(_)]`). `SysPaths` is the list of all filters that match file paths to system files. Note that we consider any file not inside a directory dedicated to a third party application to be a system file (e.g., the Address Book or Preference files). `Reads` is the list of all read operations. `Writes` is the list of all write operations.

4.4.3.2 Example Policy Queries

We now describe the three policy queries that we use to analyze the container sandbox profile in Section 4.5. These queries are stated as invariants that must hold over the policy. Any Prolog facts that match these queries are potential violations. Note that the queries listed below are simplified for readability.

1. To prevent damage to the system, full write access to system file paths, is reserved for apps with system capabilities.

   ```
   ?- allow(file-writeSTAR,Filters),
     member(X,Filters),member(X,SysPaths),
     intersection(Filters,SysCaps,[]).
   ```

2. To preserve user privacy, read access of any kind to system file paths in `/private/var/mobile/` must require capabilities.

```
?- allow(Operation,Filters),
  member(Operation,Reads),
  ((member(X,Filters),member(X,SysPaths),
  overlapPaths(X,
    subpath("/private/var/mobile/")));
  (intersection(Filters,Files,[]),
  member(vnode-type(_),Filters))),
  intersection(Filters,Caps,[]).
```

3. To prevent unauthorized collusion among third-party applications, rules providing any combination of write and read access to system files must require capabilities.

```
?- allow(Operation1,Filters1),
  allow(Operation2,Filters2),
  member(Operation1,Reads),
  member(Operation2,Writes),
  member(X,Filters1),member(X,SysPaths),
  member(Y,Filters2),member(Y,SysPaths),
  overlapPaths(X,Y),
  intersection(Filters1,Caps,[]).
  intersection(Filters2,Caps,[]).
```

4.4.4 Attack Testing Application

The example queries in Section 4.4.3.2 may direct us to file paths that are not interesting or exploitable (e.g., a readable system file that does not contain sensitive information). To assist in validating the analysis results and detecting significant attacks, we created an application to test several types of file system attacks against iOS. The attack testing application also made it easier to create proof of concept attacks when reporting our findings to Apple. If a test fails because access is denied or the file path provided is invalid, an appropriate error message is provided.

Note that it is important that the attacks are tested on an iOS device, as the Xcode iOS simulator fails to validate attacks confirmed on real devices. We speculate that the iOS simulator has a simplified file system in which the files we attacked did not exist or were not accessible. The iOS device running the attack application does not need to be jailbroken. However, jailbroken devices can provide additional feedback and insight for creating and evaluating attacks.

Table 4.2 lists the functions provided by our application. The `setPermissions` function changes the Unix permissions (i.e., read, write, or execute) for a file or directory. The `deleteAndHold` function

Table 4.2 Attack Verification Functions

Test	Parameters
requestAddressBook	
fileExists	filePath
readFileMetaData	filePath
readFileContent	filePath
createDirectory	filePath
deleteAndHold	filePath
createFileWithContent	filePath
appendFileWithContent	filePath
deleteFile	filePath
lsDirectory	directory
consumeStorage	directory, numFiles
queryDatabase	filePath, query
createSymLink	source, destination
createHardLink	source, destination
moveFile	source, destination
setPermissions	filePath, permissions

deletes a file and replaces that file with a directory of the same name. We use this attack to prevent iOS from using the affected file path. The `consumeStorage` function copies a given number of 10 megabyte files to a specified directory. The `requestAddressBook` function requests access to the AddressBook from the user. If the user grants access, the application gains the `AddressBook` sandbox extension.

4.5 Results

In this section we quantify and categorize the sandbox misconfigurations detected by our Prolog queries. We also present seven classes of attacks that abuse the sandbox misconfigurations detected by SandScout.

4.5.1 Prolog Query Results

We ran each of the Prolog Queries mentioned in Section 4.4.3.2 on a collection of Prolog facts representing the container profile for iOS 9.0.2. The results of these queries were then evaluated using our attack testing application on a jailbroken iPhone 5s running iOS 9.0.2. For each Prolog fact matched by a query, we confirm that the file access operation indicated by the fact was actually allowed on iOS 9.0.2. We also look for unique and significant attacks that are possible because of these facts. Table 4.3 presents the results of our evaluation.

Table 4.3 Query Results for iOS 9.0.2

Metrics	Query 1	Query 2	Query 3
Matched Facts	10	39	20
False Negatives	0	1	1
Exploitable Facts	10	3	8
Exploitable Paths	9	2	4

In total, SandScout produced 1520 Prolog facts to represent the container profile from iOS 9.0.2. Prolog queries can provide a significant reduction in the search space of rules an analyst must evaluate when searching for flaws. For example, Query 1 only produces 10 matching facts.

The False Negatives row represents facts that suggested more restrictions than we encountered in testing. To the best of our knowledge this occurred twice during our tests. First, we encountered a regular expression suggesting that we could write data to in a specific directory as long as the file names matched the expression. During testing on our jailbroken iOS 9.0.2 device, we found that we could create and write data to any file in the directory. Second, we were able to read the contents of the `/private/var/mobile/Library/Preferences` directory when we should have only been allowed to read its metadata. Our non-jailbroken iOS 9.3.1 device was not able to perform these unusual actions. Therefore, we speculate that these false negatives are due to imperfections in SandBlaster or artifacts of the jailbreak process.

Some file paths could not be used for attacks for reasons other than access control. Some allowed filepaths cannot be used to read user data because there is no file at the path we are allowed to read. If a third-party application has read access to a filepath (e.g., `/var/userSecrets.txt`), but such a file does not exist, then Apple may not consider the access to be dangerous. Many of the system files our test application was allowed to read did not contain interesting information. Finally, files in `/dev` do not function as normal files, and we do not consider them in our tests for Query 3. For example, having read and write access to `/dev/null` does not mean it can be used for collusion. Further investigation of attacks against `/dev` files is left as future work.

The Exploitable Facts row shows the number of facts that led us to unique, significant attacks. The Exploitable Paths row represents the number of unique file paths we were able to attack. For example, there may be multiple exploitable facts indicating access to the same file path, but we would consider all of these to be 1 exploitable file path. Matches to our queries that are not classified as Exploitable Facts should not be ignored. For example, files that did not exist in the file system during our testing might be created under conditions we are not aware of. It is also possible that some files are only present on devices with unique functionality, such as AppleTV devices or an iPad Pro.

4.5.2 Verified Attacks

We have discovered seven classes of attacks that abuse misconfigurations in the container sandbox profile's file access rules. Each of these attack classes has been disclosed to Apple and has been tested successfully on a non-jailbroken iPod Touch 6 running iOS 9.3.1 (latest version at the time of experiments).

Many of the vulnerabilities discovered by SandScout can be addressed by modifying the sandbox profile. Apple has included fixes for most of the vulnerabilities in the upcoming release of iOS 10, which we did not yet have the opportunity to verify at the time of writing. However, some of the vulnerabilities required architectural changes, which are actively being addressed. To protect against these attacks, Apple plans to monitor applications in the App Store for corresponding behaviors.

4.5.2.1 Bypassing Privacy Settings

The following is a Prolog fact that matches Query 1 because the `AddressBook` extension is not a system capability.

```
allow(file-writeSTAR,
  [subpath("/Library/AddressBook/"),
    extension("AddressBook")]).
```

Full write access allows for the creation of a hard link to the `AddressBook` database, while an app has access to it. The hard link allows the application to maintain access to the `AddressBook` even after the user revokes access through Privacy Settings. Hard link based access is not limited to the app's home directory. Malware can also place the hard link in a directory accessible to all third-party applications (e.g., `/private/var/mobile/Library/Caches/com.apple.keyboards/`). We found that `/private/var/mobile/Library/Caches/com.apple.keyboards/` could be used for collusion with Query 3. This allows colluding applications to access the `AddressBook` regardless of the user's Privacy Settings. We disclosed this attack to Apple and they partially resolved it through CVE-2015-7001 by adding a new sandbox operation, file-link, which governs the ability to create hard links. By using SandBlaster, we detected that the following rule was added to the container profile in iOS 9.1.

```
(allow file-link
  (require-not
    (subpath
      "/Library/AddressBook")))
```

This sandbox rule prevents us from creating hard links to files in the `AddressBook`'s directory. However, we have identified two methods to bypass this rule and perform the attack despite Apple's patch. First, we can simply move the `AddressBook` directory to a new location, make our hard links,

and move the AddressBook directory back to its original location. This technique succeeds because we have file-write* access to the AddressBook subpath, and moving a file does not change the file's inode. Second, our tests suggest that malicious hard links to the AddressBook are not removed when updating to newer versions of iOS. Therefore, devices attacked before iOS 9 would still be compromised after upgrading to later versions, because the new sandbox rule only prevents the creation of new hard links to the protected file paths.

Due to the complexities of this attack, a policy-based solution was not sufficient. Apple indicated that they plan to move AddressBook access out of process to address the attack.

4.5.2.2 Privacy Leaks

The container sandbox profile allows third-party applications to read several system files that contain user data. Some of these files contain sensitive data, and we consider the leakage of this data to be a breach of user privacy. The following are a subset of the Prolog facts that match Query 2.

```
allow(file-readSTAR,
  [subpath("/Media/iTunes_Control/iTunes/")]).
allow(file-readSTAR,
  [subpath("/Library/Caches/GeoServices/")]).
allow(file-read-metadata,
  [vnode-type(directory)]).
```

iTunes Privacy Leaks: The /private/var/mobile/Media/iTunes_Control/iTunes directory is readable by any third-party application. Within this directory are at least three files containing private data related to iTunes and backing up the iOS device. First, there is a database that contains titles and metadata for iTunes purchases including books, movies, music, podcasts, etc. Second, there is a file containing the user's name and the names of computers the device has backed up to. Third, there is a property list file that lists applications the user has installed via iTunes. The information leaked in this directory is valuable for targeted advertising and device fingerprinting. Even music taste alone has been found to reveal significant information about a user[Voi05]. To address this attack, an additional privacy setting was added to iOS. The new privacy setting regulates access to user media.

Maps Privacy Leaks: The /private/var/mobile/Library/Caches/GeoServices directory is readable by any third-party application. Within this directory are databases that contain the locations a user has searched for in the Apple Maps application. This application is the default mapping app for iOS devices. Third-party applications can read these databases and extract the locations a user has searched for without obtaining permission to access location data. Third-party applications can abuse this information to create targetted ads or to blackmail users by threatening to reveal the history of their location searches. To address this attack, iOS 10 will move the geo-services cache to

/Library/Caches/geod and make the directory only accessible by the geod daemon.

Metadata Privacy Leaks: The container profile makes metadata of all directories and symbolic links on the iOS file system readable by third-party applications. The size and timestamps of various directories allows third-party applications to infer information about the user. The following three examples are only a few of the inferences that can be made with the metadata available: 1) time of each photo taken; 2) the last time an audio recording was created; 3) the last time a game was played. We also find that drafts of SMS messages are sometimes stored in directories named after the phone number of the recipient of the message. For example, a draft of a message to the phone number, 15551234567, would be stored in the directory in /var/mobile/Library/SMS/Drafts/+15551234567/. A third-party application can query the existence of directories named after certain numbers to determine if the user is sending SMS messages to certain people. Note that this last case is interesting, because the metadata Prolog fact brought it to our attention, but is not technically the cause of the flaw. The ability to read the metadata of the directory enhances the attack by also leaking the times that the user began or modified the SMS draft. However, we are not aware of SBPL filters that can limit the ability to query for the existence of files. Therefore, Apple may need to extend SBPL to address the SMS privacy leak vulnerability. As a short-term fix, Apple plans to prevent third-party apps from using stat on directories in mobile/Library/SMS.

Unauthorized Collusion: iOS provides official inter-app communication channels, but these require special capabilities. However, Query 3 allowed us to identify 4 unique file paths that can be abused for unauthorized communication between applications without such capabilities. /private/var/mobile/Media/com.apple.itunes.lock_sync and /private/var/mobile/Library/Keyboard/LocalDictionary are files that third-party apps can read and write. /private/var/mobile/Library/Caches/com.apple.keyboards/ is a directory where third-party apps have full read and write access. /private/var/mobile/Library/DeviceRegistry/ is a directory where third-party apps can create any directories with names consisting of numbers and capital letters. To send a message, an app could create such directories, and to receive a message, another app could check for the existence of or read the metadata of those directories. To address these attacks, iOS 10 will remove write access to com.apple.itunes.lock_sync, LocalDictionary, and DeviceRegistry. Apple indicated an ongoing effort to remove the com.apple.keyboards directory from iOS.

4.5.2.3 System Damage

We have identified two types of write-based attacks that cause system damage because of the misconfigurations detected by Query 1. Each of these attacks can be used for ransomware, because the malicious app can undo the damage after the attacker is paid. These attacks can be undone if the user performs a factory reset of the device which deletes all user data. Restoring from a backup image of the device can also undo the damage, but many users may not have backups. Both solutions

are troublesome for a user and may cause the loss of valuable information. To address the below described attacks, iOS 10 will remove write access to respective files. However, some cases such as the `AddressBook` directory require architectural changes, as discussed in Section 4.5.2.1.

Storage Consumption: Third-party apps can consume all available storage space on the device by creating files in system directories or appending large amounts of data to system files. This space is not recovered by uninstalling the app, nor does it appear in the Storage Manager as being used by the app. We found that copying a large file is the most efficient and stealthy method of consuming space. On an iPod Touch 6th generation, we can consume storage space at a rate of approximately 100 megabytes per second with negligible use of the CPU or memory. Attacking the `AddressBook` directory in this way will cause the Storage Manager to blame the Contacts application for consuming a large amount of storage space. However, the Contacts application is a system application, and it cannot be uninstalled.

Blocking Access To System Files: A third-party app can delete system files if it has write access to, and replace these files with directories of the same name. This prevents iOS from repairing the file, because the directory is in the way. The directory block is effective because iOS often creates files with randomized file name extensions and then renames them to a non-randomized file name. If the non-randomized file name is being held by a maliciously placed directory, the renaming operation will fail. We speculate that this technique helps evade link based attacks by replacing any links via the renaming operation instead of directly writing data to a predictable file path. Consider the following Prolog fact from the container profile of iOS 9.0.2.

```
allow(file-writeSTAR,
  [regex("^/EmojiPreferences[.]plist"/i)]).
```

Note that the regular expression does not end in a $ symbol, which means it only needs to match the beginning of a string. This allows iOS to create files with randomized names such as `EmojiPreferences.plist.sfjk32a` and rename them to `EmojiPreferences.plist`. Deleting the `AddressBook` database and replacing it with a directory causes observable damage to the system. The four effects of the attack are: 1) The Contacts app will show an empty list instead of contacts. 2) Adding new contacts will fail. 3) The Contacts app will not appear in the storage manager. 4) Backing up the device with iTunes will fail.

Third-party applications can delete or move system directories they have write access to. They can also change the Unix permissions of these directories. These actions prevent system applications from being able to access the system files in those directories.

4.6 Limitations

In this section we discuss the limitations of each component of SandScout. We also propose future work to address these limitations.

Sandbox Decompiler: SandBlaster is a reverse engineering tool, and the sandbox profiles it produces have not been proven to be semantically equivalent to the originals. However, we find that it provides significant insight, and it was sufficient to lead us to numerous vulnerabilities. If Apple adopts the SandScout framework, this will not be a concern for them because they have the original profiles in SBPL format.

Another limitation of SandBlaster is its dependence on leaked firmware keys for decrypting iOS firmware. Firmware keys for an iOS version are usually published [Key] by reverse engineers a few weeks after the firmware version is released. Note that firmware keys are more readily available than jailbreaks. At the time of writing, the latest public jailbreak is for iOS 9.1, but the latest released firmware keys are for iOS 9.3.1.

Policy Modeling: Our SBPL to Prolog compiler makes four assumptions. First, it assumes the SBPL profile is written correctly. With additional engineering, our compiler could detect errors in SBPL, but we saw this as unnecessary for SandScout. Second, we assume the version information will appear on the first line, and the default decision on the second line. Third, we assume that the filters and metafilters we have encountered already are the only ones we need to compile. A new filter or metafilter may not match the expressions in our grammar, and the implementation would need to be updated. Fourth, we assume that `require-not` metafilters will not contain other metafilters. SandBlaster helps us control for this, and we can remove this assumption through additional engineering.

Policy Analysis: Our Prolog queries are limited to file access. Reverse engineering the other operations controlled by the sandbox is left as future work. We do not claim that our queries have comprehensively detected all flaws in file access. However, we believe that we have identified a sufficient number of vulnerabilities to demonstrate SandScout's utility.

Our queries do not consider overlaps between regular expressions and regular expressions or regular expressions and subpaths. We speculate that this can be accomplish by providing Prolog with a finite list of literal file paths. Prolog could then determine if any of the file paths satisfy both regular expressions or the regular expression and the subpath.

Attack Verification: We used a jailbroken iPhone 5s running iOS 9.0.2 for our attack verification. We chose to use a jailbroken device because it gave us more control and awareness of the file system. We chose to analyze the container profiles from iOS 9.0.2 because this was the latest version of iOS that we had running on a jailbroken device. However, it is possible that artifacts of the jailbreak affected our tests. In two cases, we encountered false negatives and were able to perform actions

that our decompiled profile suggested would be denied. To address this concern we confirmed that each of our 7 attack classes worked on a non-jailbroken iPod Touch 6 running iOS 9.3.1.

Finally, it is possible that we may have missed attack opportunities during attack verification. For example, some of the files we had read access to seemed to contain obfuscated data. With more analysis these may be found to contain sensitive information.

4.7 Related Work

The initial iOS security research in academic venues focused on privacy threats in third-party applications. PiOS [Ege11] uses static taint analysis to detect privacy leaks. Han et al. [Han13a] compare iOS and Android applications, finding iOS applications access significantly more privacy-sensitive APIs than Android applications.

More recent iOS application security research has focused on the potential for malware. Wang et al. [Wan13] proposed Jekyll attacks, which consider a malicious developer that hides vulnerabilities within an application. The work demonstrates a fundamental limitation in Apple's vetting process. The concepts behind Jekyll apps were independently discovered by Han et al. [Han13b] and further enhanced by Bucicoiu et al. [Buc15]. Jekyll apps leverage the ability to call APIs in private frameworks. Wang et al. [Wan14] created proof of concept attacks for infecting iOS devices with malicious applications via exploiting the iTunes synching process. iRiS [Den15] uses a combination of static and dynamic analysis to detect indirect invocation of APIs in private frameworks and has detected an ad library that abuses private frameworks. Xing et al. [Xin15] demonstrate a new attack vector for iOS malware by exploiting cross-application interfaces. Finally, Kurtz et al. [Kur16] studied ways for iOS applications to fingerprint devices. In the process of their analysis, they identified flaws in the iOS sandbox; however, they give no detail on how the flaws were discovered.

There have been several efforts to improve iOS security. Davi et al. [Dav12] propose MoCFI to add control-flow integrity to iOS applications. If applied, MoCFI would significantly mitigate Jekyll apps. MoCFI was extended by Werthmann et al. [Wer13] to enforce fine-grained access control rules. Their PSiOS tool instruments each function call to validate the function to be called along with the provided parameters. Unfortunately, PSiOS and MoCFI cause unacceptably high performance overhead and require jailbreaks or significant changes to iOS for them to be implemented. To address these issues, XiOS [Buc15] deploys static binary instrumentation to insert a reference monitor into existing iOS applications. This reference monitor aims at hiding crucial runtime addresses populated by the dynamic loader. While XiOS prevents the exploits used in existing Jekyll-related attacks, the reference monitor resides in the same address space as the malicious application. Therefore, an adversary can potentially compromise the reference monitor or launch runtime attacks to bypass the XiOS policy checks.

Much of the public knowledge about the Apple's sandboxing mechanism is due to the work

of non-academic reverse engineers. Blazakis was first in describing the internals of the Apple sandbox [Bla11], and has released a set of tools to aid sandboxing analysis. However, significant changes to the iOS sandbox in iOS 7 prevent his tool from functioning properly on iOS 7 or later. The container sandboxing profile for iOS 4 has been largely studied by Zovi [DZ11]. He also shared a simplified representation of the iOS 4 container profile. Iozzo [Ioz] gave a presentation on Apple's sandbox in which he suggests potentially vulnerable areas where researchers might find attacks. His slides include graph visualizations of some sandbox profiles for OS X. Kydyraliev [Kyd] and the iOS Hacker's Handbook [Mil12] describe the internal mechanisms of Apple's dynamic capabilities called sandbox extensions. An unofficial documentation of the Apple sandbox language has been given in a public whitepaper [fG!]. However, this guide is significantly outdated and now it only covers a minority of the sandboxing operations available for iOS. Esser [Essb] updated Blazakis's tools to create a sandbox extractor and pseudo-decompiler which converts builtin iOS sandbox profiles into an intermediate graph representation. Our own sandbox decompiler uses functionality from the tools of Blazakis and Esser, but ours is the first tool to fully decompile sandboxes for iOS 7, 8, and 9.

We are the first to systematically analyze sandbox profiles for iOS. Watson [Wat01] provided a brief overview of iOS access control, but his work did not analyze specific policies. Policy analysis itself is a broad area of research. Here, we highlight several works using logic-based programming in Prolog. PALMS [Hic10b] models SELinux policy in Prolog to study information flow properties of its MLS enforcement. Similarly, PIDSI [Rue08] uses Prolog to verify flow properties of SELinux policy governing trusted programs. Note that the iOS SBPL and SELinux policy languages are semantically different. SBPL assumes one subject, whereas SELinux includes many subjects (domains). Therefore, many properties modeled in Prolog (e.g., Trusted Computing Base identification) for SELinux do not directly apply to SBPL. Chen et al. [Che09] used Prolog to model SELinux and AppArmor policies in order to evaluate the protection quality of the policies with respect to various attack scenarios. In contrast, we seek to automatically identify misconfigurations that violate the security requirements of stakeholders. An early version of Kirin [Enc08] uses Prolog to define policy invariants over Android permissions assigned to third-party applications to detect dangerous functionality. Our SandScout queries are conceptually similar, but SBPL is significantly more complex than Android's permission model, including system calls and regular expressions.

4.8 Conclusions

This paper presented the first systematic study of the iOS container sandbox profile, which confines third-party applications. Our SandScout framework automatically extracts and decompiles binary sandbox profiles into human readable SBPL. SandScout then translates the SBPL policies into Prolog facts. By modeling sandbox policy as a logic-based program, we are able to construct queries to test security properties. We construct three file-based queries for the container sandbox profile

and use them to analyze iOS 9.0.2. We then use an assisted verification tool to further refine the set of potential policy vulnerabilities. In studying the query results, we identify seven classes of exploitable vulnerabilities. Each of these vulnerabilities was also confirmed on a non-jailbroken iOS 9.3.1 device.

Our analysis of the iOS container sandbox profile is only the first step in systematically evaluating the access control in iOS. First, our Prolog queries are limited to file-based properties. The container sandbox profile also governs other security relevant system calls such as Mach IPC. Evaluating this policy requires additional semantics from the iOS environment, which we plan to incorporate in future work. Furthermore, we plan to extend our analysis to those aspects of iOS access control outside of the sandbox.

4.9 Acknowledgments

We thank Adwait Nadkarni, Micah Bushouse, Ben Andow, Isaac Polinsky, Akash Verma, and the Wolfpack Security and Privacy Research (WSPR) lab as a whole for their helpful comments. We also thank Dino Dai Zovi for his advice on reverse engineering iOS sandbox profiles.

This work was supported in part by the Army Research Office (ARO) grants W911NF-16-1-0299 and W911NF-14-1-0537, the National Science Foundation (NSF) CAREER grant CNS-1253346, the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (643964, SUPERCLOUD), and the German Federal Ministry of Education and Research within CRISP. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

```
TK_ALLOW        = "allow"
TK_DENY         = "deny"
TK_VERSION      = "version"
TK_DEFAULT      = "default"
TK_REQANY       = "require-any"
TK_REQALL       = "require-all"
TK_REQNOT       = "require-not"
TK_REQENT       = "require-entitlement"
TK_DEBUGMODE    = "debug-mode"

TK_LP           = r'\('
TK_RP           = r'\)'
TK_VARIABLE     = r'[^\"\n#\ \(\)][^\n\ \(\)]*'
TK_STRING       = r'"[^"]*"'
TK_REGEXP       = r'\#"[^"]*"'
TK_BOOL         = r'\#[tf]'

profile         : version default ruleList
version         : TK_LP TK_VERSION TK_VARIABLE TK_RP
default         : TK_LP dec TK_DEFAULT TK_RP
dec             : TK_ALLOW | TK_DENY
ruleList        : rule ruleList |
rule            : TK_LP dec TK_VARIABLE objList TK_RP
                | TK_LP dec TK_VARIABLE TK_RP
objList         : TK_LP object TK_RP objList
                | TK_LP object TK_RP
                | require objList | require
require         : requireAny | requireAll | requireEnt
requireAny      :  TK_LP TK_REQANY objList TK_RP
requireAll      :  TK_LP TK_REQALL objList TK_RP
requireEnt      :  TK_LP TK_REQENT TK_STRING objList TK_RP
                |  TK_LP TK_REQENT TK_STRING TK_RP
object          : TK_VARIABLE TK_STRING
                | TK_VARIABLE regexList
                | TK_VARIABLE TK_VARIABLE
                | TK_VARIABLE TK_VARIABLE TK_STRING
                | TK_REQNOT TK_LP object TK_RP
                | TK_REQNOT TK_LP simpleReqEnt TK_RP
                | TK_VARIABLE TK_BOOL
                | TK_DEBUGMODE
                | TK_VARIABLE TK_LP TK_VARIABLE TK_STRING
                  TK_VARIABLE TK_RPAREN
regexList       : TK_REGEXP regexList
                | TK_REGEXP
simpleReqEnt    : TK_REQENT TK_STRING
```

Figure 4.3 SBPL Context Free Grammar.

CHAPTER

5

IORACLE: AUTOMATED EVALUATION OF ACCESS CONTROL POLICIES IN IOS

5.1 Introduction

iOS (iPhone Operating System) supports Apple's mobile devices including iPods, iPads, and iPhones. With a billion iPhones sold and a decade of hardening, iOS has become ubiquitous, and uses several advanced security features. Therefore, the impact and scarcity of iOS exploits has led to the creation of sophisticated attacks. For example, exploit brokers like Zerodium pay million dollar bounties[1] for multi-stage attacks called jailbreaks. A weaponized jailbreak can bypass and disable iOS security features to provide the attacker with elevated privileges, stealth, and persistence.

To combat such exploits, iOS enforces an assortment of access control policies. These policies collectively define an overall protection system that restricts operations available to malware or compromised system processes. However, policy flaws allow untrusted subjects to perform privilege escalation attacks that maliciously modify the protection state.

Jailbreaks exploit a combination of policy flaws and code vulnerabilities. For example, if a jailbreak author discovers a kernel vulnerability, the protection state may prevent the attacker from reaching it. To reach the vulnerability, the jailbreak must use policy flaws to modify the protection state such that the vulnerable kernel interface becomes accessible. In order to prevent such exploits,

[1] https://zerodium.com/program.html

we ask the research question *"What policy flaws exist in the iOS protection system?"*

Existing tools can provide relevant data, but are unable to meet the challenges of modeling the iOS protection system. For example, SandScout [Des16] is a tool that models iOS sandbox profiles in Prolog, but it does not model runtime context, Unix permissions, or policy semantics. These features are necessary to model policy flaws in system processes and to reduce the complexity of queries.

In this paper, we propose iOracle, a framework for logically modeling the protection system of iOS such that high level queries about access control qualities can be automatically resolved. To process queries, iOracle maps access control subjects and objects to relevant policies and evaluates those policies with respect to runtime context. iOracle also supports multiple layers of abstraction based on modeled policy semantics such that queries can be less complex. For example, a process may be governed by multiple complex policies, but iOracle can abstract away from the individual policies and their esoteric semantics to answer questions about the overall protection domain of the process.

iOracle uses Prolog to provide an extensible model that can resolve queries about the iOS protection system. First, static and dynamic extraction techniques produce Prolog facts representing sandbox policies, Unix permissions, and runtime context. Second, iOracle's Prolog rules simplify query design by modeling the semantics of Unix permissions and sandbox policies. Finally, a human operator discovers policy flaws by making Prolog queries designed to verify traits of the protection system. For example, one could query to confirm that no untrusted subject can write to a given file path. If iOracle detects a violation of this requirement, it identifies relevant runtime context and policy rules allowing the operation so that the human operator can further investigate the policy flaw.

We evaluate iOracle in two ways. First, we perform a case study of four recent jailbreaks and show how iOracle could have significantly reduced the effort in discovering the policy flaws exploited by them. Second, we use iOracle to discover five previously unknown policy flaws and show how they allow privilege escalation on iOS 10. *We have disclosed our findings to Apple.*

We make the following contributions in this paper.

- *We present the iOracle policy analysis framework.* iOracle models the iOS protection system including sandbox policies, Unix permissions, policy semantics, and runtime context.

- *We demonstrate iOracle's utility through an analysis of four recent jailbreaks.* We show a significant reduction in executables to be considered by security analysts.

- *We identify previously unknown policy flaws.* These policy flaws include self-granted capabilities, capability redirection, write implies read, keystroke exfiltration, and chown redirection.

We limit the scope of this work in two ways. First, modeling code vulnerabilities is out of scope

for this paper. Therefore, constructing new jailbreaks is not a goal of iOracle because jailbreaks also require code vulnerabilities to compromise the behavior of system processes or the kernel. However, future work could combine the iOracle model with a set of code exploits as input to an automated planner. Second, we limit iOracle to modeling file access operations. As noted in Section 5.2, hard-coded checks and a lack of documentation make it difficult to model access to inter-process services in iOS. If future work models access control policies for these services, iOracle can be extended to include the new data in a more comprehensive model of the protection system.

The remainder of this paper proceeds as follows. Section 5.2 provides a background on iOS security mechanisms. Section 5.3 overviews the iOracle framework approach and findings. Section 5.4 describes the design of iOracle. Section 5.5 provides a case study of recent jailbreaks and evaluates iOracle's utility in triaging executables with policy flaws. Section 5.6 evaluates iOracle's ability to discover new policy flaws. Section 5.7 quantifies the protection systems for 15 iOS versions. Section 5.8 presents additional policy flaws detected while implementing iOracle. Section 5.9 discusses the limitations of iOracle. Section 5.10 presents related work. Section 5.11 concludes.

5.2 Background

iOS is a modified version of macOS that supports Apple's mobile devices (i.e., iPhones, iPods, and iPads). iOS uses multiple access control mechanisms including Unix permissions, capabilities, sandboxing, and hard-coded checks. Modern iOS devices (iPhone 5S and later) use two kernels, a primary kernel (XNU), and a secure kernel (Secure Enclave). However, Secure Enclave supports a separate operating system (SEPOS) and is outside the scope of this paper.

Unix Permissions: Unix permissions provide privilege separation for different users and groups of users. Each process runs with the authority of a specific user and a set of groups. Each file is owned by a user and a group, and it has a set of permissions that determine which users and groups can access it. These permissions, determine read, write, and execute permissions for the file's user owner, group owner, and for all other users. On iOS, most processes run as one of two users, `root` (UID 0), and `mobile` (UID 501). Third party applications and many system processes that do not need high levels of privilege run as `mobile`. In general, `root` can access everything regardless of Unix permissions, but `mobile` should be limited to accessing personal data and third party resources. However, `root` authorized processes can still be restricted by sandboxing or hard-coded checks as discussed later in this section. Finally, there are several protection state operations that modify Unix permissions at runtime (e.g., chown or chmod commands).

Sandboxing: Processes in iOS may run under the restriction of a sandbox profile. Sandbox profiles are compiled into a proprietary format and define access control policies that allow or deny system calls based on their context. All third party iOS applications and several system applications (i.e.,

those created by Apple) use a sandbox profile called *container*. Other system processes may use one of approximately 100 other sandbox profiles or they may run without a sandbox. Sandbox profiles are written in SBPL (SandBox Profile Language), which is an extension of TinyScheme.[2] These profiles consist of one or more SBPL rules. Each rule consists of a decision (i.e., `allow` or `deny`), an operation (e.g., `file-write*`), and a set of contextual requirements called filters. An SBPL filter can express the context of the object (e.g., file paths or port numbers) or they can express context of the subject (e.g., capabilities or user id). If the context of the system call matches the operation and all filters in the rule, then the decision is applied. If the context of the call does not match any rules, then a default decision is applied.

iOS Capabilities: Each process in iOS can have zero or more capabilities assigned to it. iOS uses two types of capability mechanisms: entitlements and sandbox extensions. Entitlements are immutable key-value pairs embedded into a program's signature at compile time. Sandbox extensions are unforgeable token strings that can be dynamically issued and accepted (the official term is "consumed") by processes. Therefore, entitlements are suitable for policies that will not change, and sandbox extensions are used in policies that may be modified at run time.

Hard-Coded Checks: Apple often uses hard-coded checks when regulating information and services shared through Inter-Process Communication (IPC). For example, a process can contain logic to ignore IPC requests from processes that do not possess a certain capability. System daemons can also contain logic to consult specialized databases files representing access control policies for revocable services. These databases are primarily used to regulate access to private user data (e.g., location data, user photos, contacts). The decentralized and ad hoc nature of hard-coded checks makes them difficult to extract and model. Therefore, iOracle does not model hard-coded checks and restricts its scope of access control policies to Unix permissions and the sandbox.

5.3 Overview

As a motivating example, let us assume a security researcher has identified a file containing sensitive data that needs to be protected from untrusted executables. For example, users can specify that system applications should not have access to their location data. The researcher would like to generate a list of all executables able to read this file based on iOS access control policies so that they can identify any policy flaws allowing access for the wrong executables.

To produce the list of executables manually, the researcher could check access control policies for each executable on the system. However, the scale, complexity, and decentralized nature of these policies makes the task especially daunting. In iOS 10.3, there are 754 system executables that could be assigned any of 140 different sandbox profile policies. The rules in these sandbox

[2] http://tinyscheme.sourceforge.net/home.html

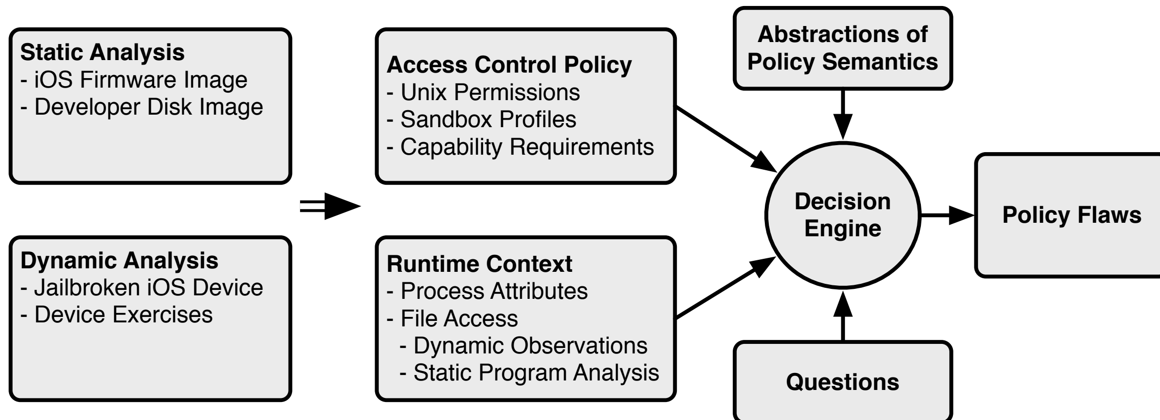

Figure 5.1 iOracle Overview

policies can be conditioned upon capabilities possessed by the subject. In total, iOracle detects over 1,000 different classes of capabilities (i.e., entitlement keys and extension classes), each of which can have various values assigned to them. The researcher must also determine effective UIDs and GIDs of executables and map the UIDs to groups they belong to. This runtime authority is then compared to the file's Unix permissions, user owner, and group owner to determine if read access is allowed. iOracle detects 20 different UIDs and 77 GIDs in iOS 10.3. Finally, the analyst should consider protection state operations (e.g., sandbox extensions, chown) which can change the protection state. iOracle automates this task by providing a framework for extracting and modeling iOS access control policies, relevant contextual data, and policy semantics.

Figure 5.1 depicts the architecture of the iOracle framework. Static and dynamic analysis are used to extract policies and context from firmware images, Developer Disk Images (DDIs), and jailbroken devices. Next, we construct logical rules providing abstractions that model the semantics of iOS access control policies. In this paper, we use questions designed to identify policy flaws, but questions about other aspects of the protection system can also be input as queries. The data, semantics, and questions are combined into a decision engine which can output potential policy flaws. The remainder of this section overviews these steps.

Data Extraction: The iOracle framework uses a variety of static and dynamic analysis tools to automatically extract policy data and runtime context from iOS firmware images, Xcode (which provides Developer Disk Images), and jailbroken devices. Examples of data extracted statically are sandbox profiles, file metadata, program entitlements, program binaries, and security configuration files (e.g., `/etc/passwd`, `/etc/groups`). Extracted program binaries are automatically analyzed using a custom IDA backtracer script to collect hard-coded parameters of security relevant functions (i.e., sandbox initialization, chown, chmod). iOracle dynamically extracts the following data for processes

running on a jailbroken device: file access operations, user authority (UID), group authority (GID), and sandbox extensions possessed. This extracted data is then parsed and formatted as Prolog facts as listed in Table 5.1.

If iOracle is designed to help find jailbreaks, but is also dependent on data from jailbroken devices, this would create a circular dependency. Therefore, iOracle uses jailbroken devices to supplement knowledge of runtime context, but is not dependent on them. iOracle primarily uses official, downloadable firmware images and developer resources as the source of policy data. Since information from jailbroken devices (Table 5.1, rows 1-3) rarely changes across versions, iOracle can use data from older, jailbroken versions to make inferences about newer, non-jailbroken versions. Several of our queries and findings can be resolved using only the static data acquired from firmware images and DDIs.

Access Control Model: iOracle models iOS access control semantics as a collection of Prolog rules. For example, this model determines which Unix permission bits are relevant for a given subject, object, and operation and evaluates queries with respect to those permissions and other relevant factors. By using a hierarchy of Prolog rules, iOracle models multiple levels of abstraction that allow it to map a high level query to relevant low level Prolog facts. For example, a query may ask which subjects can write to a given object. The solution to this query depends on several lower level queries that are processed by Prolog rules representing the access control model. These rules match runtime context of subjects and objects to respective policy requirements such that unbound variables are resolved and a solution to the query is found based on facts available. Details of this model are provided in Section 5.4.2.

Analysis and Evaluation: We use iOracle to extract facts from 15 iOS versions spanning iOS 7, 8, 9, and 10. We perform a quantitative analysis of these facts and present our findings in Section 5.7. Next, we use iOracle to successfully triage executables exploited in the jailbreaks presented in Section 5.5. In Section 5.7.2 we further study Apple's code and policy modifications in response to jailbreaks by comparing iOracle models of various iOS versions. Finally, we use iOracle to identify the following five types of previously unknown policy flaws (three others discussed in Section 5.8).

1. Self-Granted Capabilities – Sandbox policies determine which sandbox extensions can be granted and consumed by the subject. We search for flawed profiles that allow subjects to both grant and consume the same extensions without restrictions. We find multiple policies that allow arbitrary file access via self-granted extensions.

2. Capability Redirection – File-Type sandbox extensions declare a file path when they are granted. However, we find that these extensions can be arbitrarily redirected using symbolic links.

3. Write Implies Read – Sandbox policies can only represent file paths and do not track inode numbers. We find files at writable, non-readable paths that can be moved to readable paths.

4. Keystroke Exfiltration – Third party keyboards use a very restrictive sandbox profile that should

prevent them from exfiltrating keystroke logs. We find that pseudoterminals can be used to exfiltrate data to a colluding third party app.
5. Chown Redirection – We identified chown operations that can be redirected via symbolic links created by `mobile` UID subjects. By redirecting chown operations an attacker can gain privileges similar to root access.

5.4 iOracle

iOracle is an extensible framework allowing researchers to make high-level queries about the iOS protection system. Achieving this goal requires overcoming two challenges: 1) extracting the access control policies and relevant system context; and 2) constructing a knowledge base that supports abstraction for high-level queries.

5.4.1 Policy and Context Extraction

This subsection discusses our design decisions and tools used to extract the data needed to construct a knowledge base. Apple declined our request for their access control policy data. Additionally, the iOS simulator in Xcode oversimplifies the file system and therefore is unsuitable for iOracle. Therefore, iOracle extracts policies and context from iOS firmware images (distributed by Apple as updates), DDIs (extracted from Xcode), and jailbroken iOS devices. The result of the policy and context extraction is the Prolog facts listed in Table 5.1.

5.4.1.1 Static Extraction and Analysis

We statically extract the following types of data from iOS firmware and DDIs: 1) file metadata and Unix configurations; 2) program attributes; 3) sandbox assignment; 4) sandbox profile rules.

Official iOS firmware images and DDIs contain sandbox profiles, system executables, file metadata, and Unix user/group configurations. The DDI is mounted by Xcode over the `/Developer/` directory of an iOS device in order to support development features such as debugging. It contains additional system executables that can play a significant role in jailbreaks as discussed in Section 5.5. We statically process the firmware and DDIs for each secondary iOS version ≥ 7 (i.e., 7.0, 7.1, 8.0, 8.1, 8.2, 8.3, 8.4, 9.0, 9.1, 9.2, 9.3, 10.0, 10.1, 10.2, 10.3).

File Metadata and Unix Configurations: We extracted file metadata including the Unix permission bits, file owners, file path, and link destination of each file. This data was acquired using the macOS `gfind` utility to traverse a directory that combines firmware image and DDI for each version. Since `gfind` only provides a very coarse granularity of file type (e.g., regular file, symlink), we extract the files from the disk images and use the `file` utility on Linux to collect more fine-grained file types

(e.g., Mach-O armv7 executable). We also extract Unix user and group data from `/etc/passwd` and `/etc/groups` respectively.

Program Attributes: We use `jtool`[3] to extract symbols, code signatures, and entitlement key-value pairs from each system executable. We use the `strings` utility on Linux to extract strings from each system executable.

We created a custom Interactive DisAssembler[4] (IDA) script to backtrace hard coded parameters for chown, chmod, and sandbox initialization functions. Our backtracer is engineered to infer register values while considering architectural differences in armv7 vs arm64 binaries and logic used in Position Independent Execution (PIE). This backtracer is similar in concept to those implemented by PiOS [Ege11] and iRiS [Den15]. However, PiOS and iRiS are not publicly available, and were designed to process objective-c dispatch functions, while we need to infer parameters for chown, chmod, and sandbox initialization functions.

Sandbox Assignment: A sandbox profile is assigned to an executable based on three factors: 1) entitlements; 2) file path of the executable; and 3) self-assignment functions. A self-assigning executable calls a sandbox initialization function with a sandbox profile as a function parameter. Our backtracer data allows us to determine which profile will be applied to executables that sandbox themselves by inferring these parameters.

Sandbox Profile Rules: We obtained the code for SandBlaster [Dea16] and SandScout [Des16] from their authors and extended them. We used SandBlaster to extract sandbox profiles from iOS firmware images and decompile them from Apple's proprietary binary format into human readable SBPL. Apple made significant performance optimizations that changed the proprietary sandbox format in iOS 10, so we added new functionality to SandBlaster to process these. We used SandScout to compile the SBPL sandbox profiles into Prolog facts. The original SandScout models each profile in isolation with an emphasis on the container profile. Therefore, we made modifications to produce facts that more easily allow comparison between profiles and to process new sandbox filters.

SandScout can list sandbox filters for each rule, but it requires the operator to design sandbox filter semantics into queries. To address this issue, iOracle automatically matches subjects and objects to relevant sandbox rules based on iOracle's built-in model of semantics for ten types of sandbox filter as discussed in Section 5.4.2.

5.4.1.2 Dynamic Extraction and Analysis

We perform dynamic analysis on jailbroken iOS devices by continuously running a series of tools while a human performs actions on the device. Known jailbreaks exploit the interface between the iOS device and a desktop, and they abuse access to file paths in the `Media/` directory. Therefore we

[3]http://newosxbook.com/tools/jtool.html
[4]https://www.hex-rays.com/products/ida/

Table 5.1 Policy and Runtime Context Prolog Facts

Description	Extraction	Functor
File Access Observations	dynamic	fileAccessObservation/4
Process Ownership	dynamic	processOwnership/3
Sandbox Extensions	dynamic	sandbox_extension/2
Sandbox Profile Rules	static	profileRule/4
Entitlements Possessed	static	processEntitlement/2
Signature Identifier	static	processSignature/2
Executable Strings	static	processString/2
Executable Symbols	static	processSymbol/2
Directory Parents	static	dirParent/2
File Type (From Header)	static	file/2
Unix User Configuration	static	user/7
Unix Group Membership	static	groupMembership/3
Vnode Types	static	vnodeType/2
Sandbox Assignment	static (backtraced)	usesSandbox/3
Function Parameter	static (backtraced)	functionCalled/3
File Inode Number	static	fileInode/2
File GID	static	fileOwnerGroupNumber/2
File UID	static	fileOwnerUserNumber/2
File Permission Bits	static	filePermissionBits/2
File Symlink Target	static	fileSymLink/2
File Type (Unix Types)	static	fileType/2

perform three actions on the device: 1) backing up the device via iTunes; 2) taking a photo; and 3) making an audio recording. We collect the following types of data via dynamic analysis: 1) sandbox extensions; 2) file access operations; and 3) process user authority.

Since iOS devices cannot downgrade to run older iOS versions, jailbroken devices are less readily available than firmware images. Therefore, we perform dynamic analysis on a device for each major version to supplement static analysis from the same major version. For example, the dynamic analysis data from our iOS 7.1.2 device supplements our static data for both iOS 7.0 and 7.1. Our four jailbroken devices include an iPhone 4 with iOS 7.1.2, an iPod 5th Gen with iOS 8.1.2, an iPhone SE with iOS 9.3.2, and an iPod 6th Gen with iOS 10.1.1.

Sandbox Extensions: Sandbox extensions act as dynamic capabilities granted to and consumed by a process at runtime in order to satisfy conditions in that process's sandbox profile. For example, a sandboxed third party application can only access the Address Book database if it has consumed the `addressbook` sandbox extension. We use `sbtool`[5] to dynamically log the sandbox extensions possessed by each process running on the device.

Files Accessed: Unsandboxed processes can access (i.e., read, write, or execute) any file on the file system that the Unix permissions allow them to access. Therefore, the set of files that such unsandboxed processes can access is often too large to be useful. The set of files that these processes

[5]http://newosxbook.com/articles/hitsb.html

actually access during runtime and the types of access that occur (e.g., modify, chown) are more useful for detecting policy flaws. We collect file access observations using `filemon`[6] to log various file system operations, the process that performed them, and the files affected. Note that these observed file access operations are intended to be used as an optional heuristic to triage exploitable file paths. iOracle still models the set of files accessible to unrestricted executables (i.e., no sandbox or root).

Process User Authority: We use `ps` to determine the effective UID and GID of each process running on the device. This dynamically captured information is especially relevant in finding the processes that run as `root` and should therefore be classified as high integrity.

Dynamic Analysis Limitations: The `sbtool` sandbox extension extraction feature only runs correctly on our iOS 10.1.1 device. Therefore, each model of the iOS versions created in our study uses sandbox extension data from iOS 10.1.1. The effective UID and GID of a process may change under different run time scenarios (e.g., a process could be run with either root authority or less privileged authority). Finally, we do not claim complete coverage of iOS functionality. Therefore, our results represent an inherently lower bound on the process authority, file operations, and sandbox extensions that may occur.

5.4.2 Knowledge Base Construction

We model the iOS protection system by constructing a knowledge base in Prolog. This construction requires reformatting the output of various tools into Prolog facts as listed in Table 5.1. We then designed Prolog rules that resolve high-level queries into a hierarchy of subqueries that find the facts required to satisfy the high-level query. A simplified hierarchy of rules is shown in Figure 5.2. Note that the lowest level rules (e.g., literalFilter, checkRoot) consult Prolog facts generated during extraction.

Prior work (i.e., SandScout) uses Prolog facts to model sandbox policies, but relies on a human to embed relevant semantics into complex queries requiring significant expert knowledge. For example, SandScout could return a set of sandbox filters related to `file-write` operations, but it lacks the context or modeled semantics to automatically match those filters to file paths. While it is possible to construct queries by directly referencing Prolog facts, high-level questions involving multiple policies require specifying an unmanageable number of conditions.

To address this issue, iOracle uses a hierarchy of Prolog rules to keep queries at a more practical level of abstraction. The following question will act as a running example for the remainder of this section: *What set of processes P can create files at filepath f?* While the question appears simple, answering it requires consideration of many facts, including Unix permissions, process authority, sandbox rules, and sandbox assignment.

[6]http://newosxbook.com/tools/filemon.html

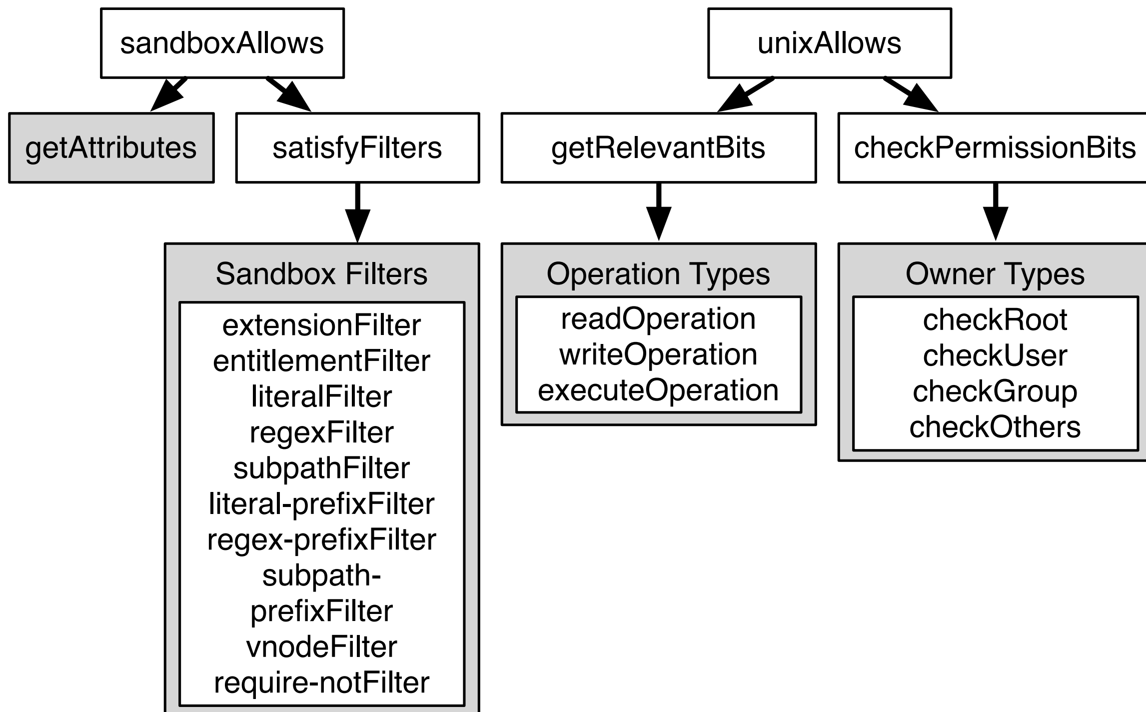

Figure 5.2 Simplified Hierarchy of Prolog Rules

The remainder of this section discusses the purpose and design of each rule iOracle uses to provide abstractions over policy semantics. Note that users of iOracle also maintain the ability to directly reference facts for simple queries such as checking the entitlements possessed by a given executable.

Sandbox and Unix Policy Interaction: For a sandboxed, non-root process to perform a file operation, both the sandbox and Unix policies must allow the operation. These policies sometimes have different requirements for similar operations. Creating a file is one such example. To create a file, the sandbox must allow write access for the filepath. In contrast, Unix permissions require write access to the parent directory of the filepath in order to create a file. Therefore, the query for the running example is made as follows:

```
?-dirParent(Parent, Path),
  unixAllows("write", Parent, Process),
  sandboxAllows("file-write*",Path,Process).
```

In this query, `dirParent` captures a filepath, `Path`, and its parent directory, `Parent`. `unixAllows` and `sandboxAllows` query the Unix permissions and sandbox policy, respectively.

sandboxAllows: The sandbox access control mechanism depends on the default policy of the match-

ing profile. The vast majority of sandbox profiles in iOS are default `deny`, so the iOracle rules assume a default `deny` policy. Our Prolog rules supporting sandbox decision abstraction are designed to match relevant context to sandbox rules that allow a given operation. The `sandboxAllows` rule is defined as:

```
sandboxAllows(Operation,Object,Process) :-
  getAttributes(Process,Entitlements,Extensions,User,Home,Profile),
  profileRule(Profile,Decision,Op,Filters),
  satisfyFilters(Filters,Entitlements,Extensions,Home,Object).
```

To match a sandbox rule to a system call's context requires three sources of information: the operation, the subject's context, and the object's context. The operation can be specified directly in our query (e.g., `file-write*` for full write access to a file), and matched directly to sandbox profile facts. The subject is the sandboxed process, and the object is a file path. Not all objects in sandbox rules are files, but iOracle is designed to model file access. The `getAttributes` rule maps a process to its respective entitlements, extensions, etc.

Matching the subject and object context to a sandbox rule requires satisfying all filters listed in the sandbox rule. Modeling the semantics of each filter type is non-trivial, and is performed in iOracle by defining a Prolog rule for each of 10 filter types as shown in Figure 5.2. For example, one filter could specify that the filepath satisfy a regular expression while another requires a certain Vnode type. A notable exception that we also model is the `require_not` filter, which requires that a given filter not be satisfied. Since we need to process a list of filters, we recursively process each filter and declare the rule to be matched if all filters are satisfied. Consider the following fact for the disjunctive normal form of a sandbox rule.

```
profileRule(profile("example_profile"),decision("allow"),
  operation("file-write*"),
  filters([require_entitlement("system-groups",[]),
  extension("system-daemon"),
  require_not(vnode-type(character-device)),
  regex('^.*\.db$'),
  subpath("/private/var/containers/")])).
```

Each filter in the rule must be satisfied for the rule's `allow` decision to be applied. Therefore, a process with a `true` value for the `system-groups` entitlement key and a sufficient extension value for the `system-daemon` extension class could write a non-character-device file in `/private/var/containers/` that ends in `.db`.

Subject Context Sandbox Filters: Three sandbox filters relate to the access control subject (process) context: `prefix`, `require-entitlement`, and `extension`. The semantics of each filter are modeled by Prolog rules that together determine if a process's context matches a given sandbox rule's filters.

The prefix filter uses Apple defined variables to act as the prefix of a file path. For example, the filter prefix(${HOME}/foo.txt) requires the subject file to be the foo.txt file in the process user's home directory. Therefore, the rule to model this filter must reference process ownership facts and facts that determine the home directories of iOS users. If the subject runs with the authority of user mobile, the filter would match the mobile/foo.txt[7] filepath.

The require-entitlement filter specifies an entitlement key-value pair and is only satisfied if the sandboxed process has the entitlement key-value pair embedded in its signature. We model this requirement by searching for entitlement facts that satisfy the filter. If such facts do not exist, the process lacks the required entitlement.

The extension filter specifies the sandbox extension class that must be possessed by the process in order to satisfy the filter. However, unlike the require-entitlement filter, the extension filter represents more flexibility due to the extension's value. While the sandbox profile can specify the extension class, the extension value is not specified in the profile. In addition to the class, sandbox extensions also have a type and a value. If the extension type is file, then the value will be a subpath that filepaths may match. For example, an extension value of /tmp allows access to /tmp and files inside /tmp. We evaluate extension filters by referencing sandbox extension facts generated by dynamic analysis. The following Prolog code is used to satisfy file type extension filters:

```
%The filter to satisfy is for a sandbox extension.
satisfyFilters(extension(ExtClass),_,Ext,_,file(ObjectPath)):-
  %Does subject have required file type sandbox extension class?
  member(extension(class(ExtClass),type("f"),value(ExtValue)),Ext),
  %Does object file path match extension value?
  satisfyFilters(subpath(ExtValue),_,_,_,file(ObjectPath)).
```

Object Context Sandbox Filters: There are sandbox filters based on the context of the access control object. Objects include files, network ports, and mach-services; however, for the purposes of this paper, we only consider files. The literal filter matches an exact file path. The subpath filter matches all file paths within a given subpath (e.g., all files in /var/mobile/). The regex filter matches file paths that match a given regular expression. Each of these filters may contain variables to represent a prefix to the filepath (e.g., ${HOME} would be replaced by the subject's home directory when resolving the filter. These filters are evaluated by comparing filter values to facts about files found in the file system or the file paths accessed during dynamic analysis.

unixAllows: For the Unix policy to allow creating a file, the parent directory must be writable. Unix permission semantics are not proprietary, but they are non-trivial to model. In general, the Unix permission mechanism will allow an operation to proceed if any of the following conditions hold:

[7] /private/var/mobile/foo.txt

[8] Figure inspired by presentation on Pangu 9.
http://blog.pangu.io/wp-content/uploads/2016/08/us-16-Pangu9-Internals.pdf

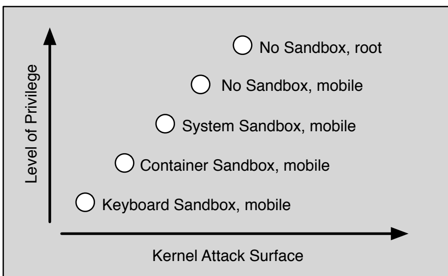

Figure 5.3 Privilege Levels and Kernel Attack Surface[8]

1) the process user is `root`; 2) the process user is the owner of the file and the owner has permission to perform the operation; 3) the process user is a member of the group that owns the file and the group owner has permission to perform the operation; and 4) the Unix permissions allow users other than the user owner or group owner to perform the operation. We also model exceptions such as parent directories that are not executable or user owners being denied access while others are granted access (e.g., 077 Unix permissions). Our rules modeling Unix policy decisions reference file metadata facts to get file context such as file ownership and permission bits. These rules reference facts on process ownership and group membership for process context.

5.5 Case Study: iOS Jailbreaks

A primary use case of iOracle is the discovery of policy flaws that enable jailbreaks. In this section, we investigate four recent jailbreaks in order to characterize the different types of policy flaws that have enabled them. We broadly separate our discussion into name resolution based flaws and capability based flaws. We then conclude the section by demonstrating iOracle's ability to direct a security analyst to executables likely to be exploited.

5.5.1 Understanding iOS Jailbreaks

A jailbreak is a collection of exploits that place Apple-mandated iOS security features (i.e., code signing, sandboxing, and Unix permissions) under the jailbreaker's control. This ability to disable security features can be abused by malware to gain persistence and elevated privileges. For example, the Pegasus[9] malware combined a trio of exploits called Trident to jailbreak the victim's iOS device via a malicious web page. Jailbreaks represent a significant threat to iOS users as well as a powerful tool for attackers.

Early jailbreaks such as L1meRain performed exploits during the device's boot sequence [Lev16]. However, as of the iPhone 4S, Apple improved hardware security and boot-time jailbreaks became

[9]https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf

less feasible. Modern jailbreaks attack the system after it has booted, and their components can be divided into userland exploits and kernel exploits. Jailbreaks typically use a series of userland exploits to reach a vulnerable kernel interface in order to deploy a kernel exploit. Figure 5.3 illustrates various levels of privilege on iOS. As the attacker gains privileges, the kernel attack surface increases.

Jailbreaks exploit a combination of policy flaws and code vulnerabilities. For example, a code vulnerability may provide the attacker with elevated control of a system process, but policy flaws must still be exploited to bypass access control mechanisms. We refer to these code vulnerabilities and policy flaws as "jailbreak gadgets" since they can be assembled into a chain where one gadget provides the privileges or control required to exploit the next gadget. We categorize 4 jailbreaks into 2 families and study their jailbreak gadgets as inspiration for iOracle queries. iOracle is designed to detect policy flaws, but it does not identify code vulnerabilities. However, we still discuss code vulnerabilities to provide a better understanding of the attacks.

The following survey of known jailbreaks and their gadgets is based on Levin's book chapters [Lev16], conference presentations,[10] blog posts,[11] and our own investigations.

Due to space constraints and scope limitations, some jailbreak gadgets have been simplified or excluded. Figures illustrating the privilege escalation attacks discussed in the remainder of this section are available in Section 5.7.2, Figure 5.4 and Figure 5.5.

5.5.1.1 Name Resolution Based Jailbreaks iOS 7-8

The jailbreaks in this family share the same start and goal states and primarily use name resolution attacks to elevate their privileges. We define the start state as a limited interface with the Apple File Conduit Daemon (`afcd`), which is accessed via a computer connected to the iOS device. This interface with `afcd` is a suitable starting point because it allows the creation of symlinks in `Media/`, a directory which several potential confused deputies must traverse. Within userland, we define the goal state as write access to the root partition, which is normally mounted as read-only.

There are three layers of security between the start and goal state that prevent the attacker from directly remounting the root partition: 1) the limited interface with `afcd` only allows read and write access to files in `Media/`; 2) a dedicated sandbox profile restricts which system calls `afcd` can make; and 3) Unix permissions only allow `afcd` to access files available to Unix user `mobile`.

evasi0n 7 (iOS 7): The user interface defined by `afcd` prevents the jailbreaker from creating symlinks that redirect to files outside of `Media/`. This symlink restriction is enforced when a symlink is created, but no enforcement occurs when a symlink is moved. However, when a relative symbolic link is moved, it may resolve to a new filepath. Therefore, jailbreakers can create and then move symlinks with `../` sequences in them to bypass this restriction.

[10] https://cansecwest.com/slides/2015/CanSecWest2015_Final.pdf
[11] http://proteaswang.blogspot.com/2017/

With the interface restrictions bypassed, afcd can write through the links to files outside of Media/, but it is still sandboxed. When afcd is launched, it calls a sandbox initialization function to sandbox itself. Therefore, if the export symbol for this library call is overwritten and redirected to a different function, the sandbox will never be applied. This technique is called export symbol redirection, and it does not violate code signing since export symbols are not covered by the code signature. The afcd sandbox allows it to deploy symlinks in tmp/ and perform a name resolution attack against installd which must traverse directories in tmp/. installd is used as a confused deputy to modify afcd's libraries to deploy the export symbol redirection attack and disable afcd's sandbox.

At this point afcd is unsandboxed, but running as the Unix user mobile. However, a root authorized executable called CrashHousekeeping performs a hard coded "chown to mobile" operation on Logs/AppleSupport. Since Logs/ is writable by mobile, afcd can replace AppleSupport with a link to the device file for the root partition. This name resolution attack causes CrashHousekeeping to change the owner of the root partition to mobile, achieving the goal state.

TaiG (iOS 8): In iOS 8, afcd can still create symbolic links in Media/. Instead of relying on confused deputies available during normal activity, the TaiG jailbreak exploits an obsolete, but vulnerable executable called BackupAgent. Although BackupAgent2 has been in use since at least iOS 4, its predecessor BackupAgent can still be found on the iOS file system. The TaiG authors reverse engineered the protocol to communicate with BackupAgent and interfaced with it via USB connection prompting it to perform a recovery operation. The recovery requires BackupAgent to move files from the Media/ directory into a backup staging directory.

TaiG performs a name resolution attack by using a chain of two different symbolic links: 1) Link1 is moved by BackupAgent into the backup staging directory; 2) Link2 is moved by BackupAgent into the backup staging directory passing through Link1 as the destination of the move operation is resolved. When BackupAgent moves the second link, it resolves the first symbolic link, effectively placing the second link anywhere that BackupAgent can write.

Even if the root partition is read-only, the directories inside it can be used as mount points and overwritten with attacker content by using a mounting agent as a confused deputy. Therefore, TaiG uses BackupAgent to overwrite MobileStorageMounter's working directory with a symlink to a directory in Media/. This name resolution attack changes MobileStorageMounter's working directory from a high integrity directory to a low integrity directory. TaiG deploys malicious disk images into the new working directory and proceeds to exploit MobileStorageMounter such that a fake disk image is mounted over /Developer. Malicious configuration files in the fake disk image allow more disk images to be mounted over the root partition to achieve the goal state.

Name Resolution Insights: Before the attacker can perform a name resolution attack, they must find an intersection of an accessible directory and a confused deputy working in the directory. Therefore,

Table 5.2 Triage of Likely Attack Vectors and Confused Deputies Based on Known Jailbreak Gadgets

Query	iOS Version	Jailbreak	Executables on System	Executables Detected	Target Executable	Target Detected
chown/chmod name resolution attack confused deputy	7.0	evasi0n 7	314	2	CrashHousekeeping	Yes
low integrity can create files in tmp/	7.0	evasi0n 7	314	60	afcd	Yes
high integrity works in tmp/	7.0	evasi0n 7	314	39	installd	Yes
low integrity can create files in Media/	8.0	TaiG	411	3	afcd	Yes
high integrity works in Media/	8.0	TaiG	411	25	BackupAgent	Yes
triage executables with _mount symbol	8.0	TaiG	411	4	MobileStorageMounter	Yes
capabilities for full control with non-container sandbox	8.0	Pangu 8	411	1	neagent	Yes
capabilities for full control with non-container sandbox	6.1*	Pangu 9	259	1	vpnagent	Yes
capabilities for full control with non-container sandbox	9.0†	Pangu 9	564	1	N/A	N/A

* Pangu 9 installs an executable with required capabilities from iOS 6.1, bypassing expired signature with an exploit.
† iOracle additionally detected an executable with required capabilities on iOS 9.0 that may obviate the need to use an expired signature.

it is useful to know which high integrity processes work in a given directory and to know which low integrity processes can access a given directory. In some cases, a name resolution attack can be triaged to specific file paths (e.g., chown or chmod operations on hard coded filepaths), so it useful to observe these operations dynamically or predict them statically with our backtracer. Separation of duties limits the usefulness of each confused deputy (e.g., the `BackupAgent` is unlikely to mount a partition, but `MobileStorageMounter` can). This separation of duties allows us to use iOracle to identify interesting executables based on rarely used, security sensitive function calls. Finally, legacy code (e.g., `BackupAgent`) represents a security risk as it expands the options available to attackers and may contain vulnerabilities. Therefore, iOracle models all executables on the firmware, even those considered to be legacy code.

5.5.1.2 Capability Based Jailbreaks iOS 8-9

We categorize Pangu 8 and 9 as capability based jailbreaks. Instead of name resolution attacks, these jailbreaks exploit exceptions in access control policies for processes with specific capabilities. We define the start state as access to the `debugserver` which is mounted as part of the iOS DDI and accessible via USB connection. The container sandbox profile is too restrictive to deploy the Pangu 8 and 9 kernel exploits, but other profiles are less restrictive. Therefore, we define the goal state as full control of a process that is not sandboxed with the container profile.

Pangu 8 (iOS 8): One method of gaining control of an executable is to have it import an attacker defined library. `debugserver` does this by manipulating environment variables before launching an executable. While `debugserver` in iOS 8 is sandboxed, our reversal of its sandbox profile shows that it can execute any file outside of the `Containers/`[12] directory, which holds third party apps. However, code signing requirements prevent jailbreakers from arbitrarily injecting third party libraries into system executables.

Unfortunately for Apple, `neagent` (Network Extension Agent) exists outside of `Containers/`,

[12] /private/var/mobile/Containers

has an entitlement called skip-library-validation, and runs with the vpn-plugins sandbox profile. In order to support third party VPN applications, neagent uses the skip-library-validation entitlement to bypass code signing requirements when loading libraries. Therefore, debugserver is able to modify environment variables and load the jailbreaker library into neagent. This library provides the attacker with full control of neagent, and the less restrictive vpn-plugins sandbox profile allows the kernel exploit to be deployed. Thus, the goal state is achieved.

Pangu 9 (iOS 9): Apple modified the debugserver sandbox in iOS 8.2 and later such that it can only execute processes with get-task-allow entitlement (in Mach systems the taskport can be used for debugging). The Pangu team stated that they could not find an executable on iOS 9 with the get-task-allow entitlement,[13] but iOracle finds that neagent on the DDI for iOS 9.0 does have the entitlement. Regardless, the jailbreakers decided to search older versions of iOS for executables with the entitlement and found vpnagent (neagent's predecessor) on the DDI for iOS 6.1. vpnagent also uses the vpn-plugins sandbox profile, making it an ideal target to deploy the kernel exploit.

However, vpnagent is not installed on iOS 9 and its signature is not valid for iOS 9. It is not sufficient to install vpnagent as a third party application because debugserver would not be able to execute it, and the container profile would be applied to it. Therefore, the jailbreak installs vpnagent as a system application by exploiting an input validation vulnerability in a file moving service provided by assetsd. Next, the jailbreak uses a disk mounting exploit to cause MobileStorageMounter to import signatures from old disk images into the list of acceptable signatures.

At this point, the iOS 6.1 vpnagent has been installed on iOS 9, and its signature is now recognized by the system as valid. vpnagent does not possess the skip-library-validation entitlement, but debugserver can load third party libraries when debugging with the get-task-allow entitlement. Therefore, the goal state is achieved when debugserver executes vpnagent in debug mode and loads the jailbreak library.

Capability Insights: The DDI, which is mounted on an iOS device via Xcode, contains several resources useful to jailbreakers and should not be ignored. This insight is the reason iOracle uses the DDI as a source of input for static extraction. Capability based policies should also consider older executables that have been signed by Apple. Therefore, we created additional to scripts automatically run iOracle queries on multiple versions of iOS. Combinations of skip-library-validation or get-task-allow entitlements and non-container sandbox profiles are dangerous. However, the facts extracted by iOracle make finding these combinations trivial.

Other Modern Jailbreaks: evasi0n 6 (iOS 6), Pangu 7 (iOS 7) and Yalu (iOS 10) are modern jailbreaks that we have not discussed in detail. At a high level, the exploits used in evasi0n 6 are very similar to those used in evasi0n 7 and TaiG. In iOS 7, the container profile for third party applications was sufficiently privileged for the Pangu 7 iOS application to exploit the kernel without elevating its priv-

[13] https://www.youtube.com/watch?v=vCLf7tdjabY

ileges in userland. Yalu was also able to deploy its attacks from within the container sandbox profile. Yalu does so by exploiting mach services that are accessible to third party apps in order to perform a mach-port name resolution attack. This attack allows the Yalu application to intercept a credential called a `taskport` being sent as a mach-message. The `taskport` belongs to an unsandboxed, `root` authorized process called `powerd`, and provides Yalu with debugger control over `powerd`.

5.5.2 Evaluating iOracle

We evaluate iOracle's effectiveness at detecting policy flaws by using it to triage executables exploited in jailbreaks as confused deputies or attack vectors. The set of executables returned by each iOracle query includes the executable exploited by the jailbreak, which we refer to in the following text as the *target*. The number of executables detected for each query is provided in Table 5.2. Note that the queries used are intentionally more generic than the jailbreak gadgets targeted.

We define a *high integrity* executable (likely to be used as a confused deputy) as an executable that is unsandboxed, runs as root, or is sandboxed with a default allow policy. We use the term *low integrity executable* (likely to be used as an attack vector) to refer to other executables.

evasi0n 7 (iOS 7): The evasi0n 7 jailbreak used a name resolution attack in the `tmp/` directory and a name resolution attack against a hard coded chown operation. To triage the attack in `tmp/`, we search for either the attacking process (low integrity) or the confused deputy (high integrity). We query to find all low integrity executables with write access to files inside of `tmp/` and identify 60 executables including the target `afcd` for iOS 7.0. We query to find all high integrity executables that reference `tmp/` in their strings or were dynamically observed to access files in `tmp/`. This query identified 39 high integrity executables likely to work in `tmp/` including the target `installd` for iOS 7.0. For the chown attack, we query for executables with hard coded chown operations targeting file paths in directories that are writable by user `mobile`. This query returned two executables including the target `CrashHousekeeping` for iOS 7.0. In addition to the filepath exploited in the jailbreak, iOracle revealed two more exploitable filepaths chown'ed by `CrashHousekeeping`.

TaiG (iOS 8): The TaiG jailbreak used name resolution attacks in the `Media/` directory as well as exploiting a confused deputy that could mount disk images. To triage the attack in `Media/`, we search for either the attack vector or the confused deputy. We query to find all low integrity executables with write access to files inside of `Media/` and identify three executables including the target `afcd` for iOS 8.0. We query to find all high integrity executables that reference `Media/` in their strings or were dynamically observed to access files in `Media/`. This query identified 25 high integrity executables likely to work in `Media/` including the target `BackupAgent` for iOS 8.0. To triage executables that could mount disk images we query for system executables that contain the `_mount` symbol. This query detects four executables including the target `MobileStorageMounter`.

Pangu 8 (iOS 8): The Pangu 8 jailbreak required an executable with three attributes: 1) can be

executed by `debugserver`'s sandbox; 2) has the `skip-library-validation` entitlement; and 3) is not constrained by the container sandbox profile. iOracle has facts for executable entitlements, assigned profiles, and our abstraction models sandbox policy semantics. We used iOracle to search for executables with the attributes required and found that for iOS 8.0, `neagent` is the only executable that satisfies these requirements.

Pangu 9 (iOS 9): The `debugserver` profile now requires a process to possess the `get-task-allow` entitlement to be executed by `debugserver`. The jailbreak also still requires an executable that is not assigned the container sandbox profile. Our query for these two attributes showed that `neagent` on the iOS 9.0 DDI meets these requirements. We speculate that if Pangu had used `neagent` from the iOS 9.0 DDI, fewer exploits would have been required. However, Pangu chose to use exploits that allowed them to install system executables from older versions of iOS (i.e., iOS 6.1). iOracle confirms that `vpnagent` from iOS 6.1 has the required capabilities, and finds that `vpnagent` from iOS 7.0 and 7.1 could have also worked. In total we found two unique executables with the required attributes across all versions analyzed including the target `vpnagent`.

5.6 Previously Unknown policy flaws

In addition to testing iOracle on known policy flaws, we search the iOS protection system for previously unknown policy flaws. This section lists a total of five new policy flaws detected by iOracle. Other flaws are presented in the Section 5.8.

Responsible Disclosure: In August 2017, Apple confirmed receipt of an early draft of this paper disclosing the following findings. However, at the time of writing, Apple has neither confirmed nor denied the vulnerabilities detected by iOracle.

5.6.1 Self-Granted Capabilities

The sandbox profile of a process determines which sandbox extensions it can grant and which extensions in can effectively consume. Potential privileges gained via sandbox extensions are usually limited by additional filters in the sandbox profile. Therefore, we refer to an extension filter that is not paired with other significant filters as an unrestricted extension filter.

We queried for sandbox profiles that allow a subject to grant extensions to itself such that the subject gains access to arbitrary files. More specifically, the profile allows the subject to grant extensions that match unrestricted extension filters in file access rules. Consider the following pair of profile rule facts from the `quicklookd` profile, which allows `quicklookd` to give itself extensions

that provide read access to any file on the system.

```
%allowed to grant quicklook extension
profileRule(profile("quicklookd"), decision("allow"),
  operation("file-issue-extension"),
filters([extension-class("com.apple.quicklook.readonly")])).

%read access with quicklook extension
profileRule(profile("quicklookd"),decision("allow"),
  operation("file-readSTAR"),
  filters([extension("com.apple.quicklook.readonly")])).
```

If an attacker gains control of `quicklookd`, it can elevate its privilege through self-granted extensions and significantly compromise the user's privacy. Our query identified 2 profiles (i.e., `quicklookd` and `AdSheet`) on iOS 10.3 that allow a sandboxed process to grant unrestricted extensions to itself. `AdSheet` allows a process to grant itself read access to all but one filepath on the system (due to a require-not filter restriction). During this analysis we found that even third party applications could grant sandbox extensions, but these seem too restricted to be exploited. Apple should augment these sandbox rules allowing arbitrary file access based on extensions with additional filters to limit the malicious potential of this protection state operation.

Impact: Gaining read access to arbitrary files may not contribute directly to a jailbreak, but it is still a privilege escalation that could impact user privacy or assist in reverse engineering.

5.6.2 Capability Redirection

We find that it is possible to perform a name resolution attack such that a confused deputy will be redirected and effectively grant sandbox extensions with attacker defined values. When a process grants an extension, it must specify a class and a value for the extension. The class is a string that can match filters in a sandbox profile. For a file type extension, the value is a file path that will specify a subpath that objects may fall into. Similar to a chown operation, any symlinks in the file path of the extension value will be resolved before granting the extension. An attacker can replace the filepath normally targeted by the extension granting process with a symbolic link pointing to a filepath of the attacker's choice.

For example, `afcd`'s sandbox allows write access to `mobile/Media`,[14] and it is granted an unrestricted extension with the value `mobile/Media` when it is launched. If `afcd` were to replace `Media` with a symbolic link, it would be granted an unrestricted extension with a value determined by the link destination upon its next launch, providing `afcd` with read/write access to the destination of the link. To create the symbolic link, the Unix permissions must also allow `afcd` write access to the

[14] /private/var/mobile/Media

mobile directory. iOracle shows that write access is allowed because afcd runs as UID mobile, and user mobile owns the mobile directory.

We query for sandboxed processes on iOS 10.3 with write access to filepaths corresponding to the values of unrestricted sandbox extensions they possess. Our query identified seven processes that can perform this sandbox manipulation to modify their sandbox restrictions and gain read/write access to any file on the device. Two additional processes can gain access to all but one file on the device due to a require-not filter. Among these nine processes are afcd and the default email client MobileMail. afcd has a history of being exploited, and MobileMail is likely to be exposed to attacks.

If an attacker gains control over one of these nine processes, they can exploit the policy flaws to bypass sandbox restrictions on file writing operations. The attacker would be restrained as user mobile, but this policy flaw could play a significant role in jailbreaks as its effect is similar to sandbox escape. To mitigate this attack Apple can pair the flawed sandbox rules with additional filters that restrict the file paths accessible via sandbox extensions.

Impact: With respect to Figure 5.3, these policy flaws are similar to sandbox escapes allowing a jailbreak to progress from the "System Sandbox, mobile" stage toward the "No Sandbox, mobile" stage.

5.6.3 Write Implies Read

Sandbox rules can match a file path, but unlike Unix permissions, they do not follow a file when it moves. Therefore, an attacker can move a file to a filepath where less sandbox restrictions apply to the file. Creating hard links in less restricted file paths has the same effect. For example, a sandbox profile may allow write access to files in /write/, and allow write and read access to files in /write_read/. An attacker can read files in /write/ by moving them to /write_read/ which is a readable path.

We query sandbox profiles for files that can be written but not read according to the sandbox policy. Our queries detected 3 sandbox profiles on iOS 10.3 where read access to unreadable files can be acquired by abusing write access and changing file paths. The default allow profile assigned to BackupAgent is among the detected profiles because it denies read access to a specific file path, but does not deny write access to that file path.

Impact: Gaining read access may not contribute to a jailbreak, but it is still a privilege escalation that could impact user privacy.

5.6.4 Keystroke Exfiltration

Apple allows third party developers to design custom keyboards for iOS. These third party keyboards have a restrictive sandbox profile that should prevent keyloggers from exfiltrating key stroke data.

The keyboard profile does not allow access to the Internet and file write access is very restricted. Attackers could use covert channels to exfiltrate this data (e.g., manipulating global inode numbers), but these are slow and inconvenient. Therefore, we queried for filepaths where a third party keyboard has write access and a third party application has read access. Our query revealed that third party keyboards and third party applications can both read and write to a set of psuedoterminals in the `/dev/` directory.

We created proof of concept applications that share information by reading and writing to psuedoterminals on a *non-jailbroken iOS 10.2 device*. One application exports data by writing to `/dev/ttyp1`, the slave of the pseudoterminal pair. The other application accesses the data by reading from `/dev/ptyp1`, the master of the pseudoterminal pair. Once a third party keyboard has exfiltrated key logs to a third party app, the app can exfiltrate the data over the Internet.

Impact: With respect to Figure 5.3, this policy flaw allows a malicious third party keyboard to move sensitive data from a subject at the "Keyboard Sandbox, `mobile`" privilege level to a subject at the "Container Sandbox, `mobile`" privilege level.

5.6.5 Chown Redirection

High integrity system executables regularly modify Unix permissions and file ownership. However, some of these operations are susceptible to name resolution attacks similar to the one exploited by evasi0n 7 to gain write access to the root partition. We use iOracle to search permission changing file access operations (i.e., chmod/chown) performed by high integrity processes (confused deputies) on iOS 10. Of the file paths targeted by these operations, we search for those that are writable by sandboxed, `mobile` user processes (attack vectors). The query results revealed that BackupAgent2 chowns files in `Media/` such that the file owner becomes `mobile`. Since the untrusted, but sandboxed `afcd` process has write access to files in `Media/`, it can be used as an attack vector to deploy a name resolution attack against `BackupAgent2`'s chown operations.

This attack is reachable with full control of the sandboxed `afcd` process, but the sandbox could deny access to files regardless of their Unix permissions. Therefore, this policy flaw is most useful to an attacker that has escaped the sandbox, but is running as user `mobile`, The attacker can use this policy flaw to redirect chown operations such that arbitrary files become owned by `mobile`, which compromises Unix policies by making the untrusted `mobile` user the owner of files that had previously been inaccessible.

Impact: With respect to Figure 5.3, this policy flaw allows a jailbreak to progress from the "No Sandbox, `mobile`" privilege level to the "No Sandbox, `root`" privilege level.

Table 5.3 Measuring the Increasing Complexity of iOS Access Control

Description/iOS Version	7.0	7.1	8.0	8.1	8.2	8.3	8.4	9.0	9.1	9.2	9.3	10.0	10.1	10.2	10.3
sandbox profiles	63	63	95	95	99	100	100	117	117	116	121	136	137	138	140
used sandbox profiles	49	49	73	73	72	75	75	86	86	88	91	108	108	108	111
unused sandbox profiles	14	14	22	22	27	25	25	31	31	28	30	28	29	30	29
unsandboxed executables	248	250	311	311	342	369	372	412	415	418	431	537	537	539	545
sandboxed executables	66	66	100	100	102	106	107	152	152	154	158	194	194	197	209
percent sandboxed	21	20.9	24.3	24.3	23	22.3	22.3	27	26.8	26.9	26.8	26.5	26.5	26.8	27.7
executables sharing container	12	12	22	22	26	27	28	52	52	52	53	74	74	77	83
facts generated for container	1048	1051	1238	1245	1296	1322	1337	1475	1478	1476	1651	2342	2438	2539	2380
sandbox operations	114	114	114	114	114	114	114	119	123	124	125	131	132	132	137
non-mobile sandboxed processes	3	3	5	5	5	5	5	8	8	8	9	14	14	14	16
root processes	23	23	29	29	29	29	29	37	37	37	37	38	38	38	38
mobile processes	54	54	93	93	93	93	93	113	113	113	113	109	109	109	109
other user processes	4	4	6	6	6	6	6	8	8	8	8	8	8	8	8
unique entitlement keys (system apps)	312	320	503	505	544	562	567	689	693	694	746	936	937	955	986
unique sandbox extensions	17	17	31	31	31	33	33	38	38	38	42	49	49	49	49
default allow profiles	1	1	1	1	1	1	1	3	3	3	3	2	2	2	2
default deny profiles	62	62	94	94	98	99	99	114	114	113	118	134	135	136	138
Unix users	11	11	14	14	14	14	14	15	15	15	17	20	20	20	20
Unix groups	67	67	69	69	69	69	69	71	71	71	74	77	77	77	77
files on firmware rootfs and DDI images	68k	70k	89k	90k	98k	100k	101k	111k	111k	111k	114k	139k	139k	141k	143k

5.7 Comparison of iOS Versions

We performed an analysis of the data extracted for 15 versions spanning iOS 7, 8, 9, and 10. The results of this analysis are provided in Table 5.3. We also compare the data extracted across versions to detect policy or context changes made by Apple in response to jailbreaks.

5.7.1 Access Control Complexity

System files, system exectables, sandbox profiles, container profile complexity, and Unix users have all approximately doubled from iOS 7.0 to 10.3. In the same time, the number of unique capabilities (i.e., entitlement keys and extension classes) has approximately tripled. This rate of increasing complexity in subjects, objects, capabilities, and policies emphasizes the need for frameworks that automate access control evaluation as manual analysis becomes intractable.

Note that the majority of executables on iOS 10.3 are still unsandboxed. Among these unsandboxed processes is the default Messenger app, `MobileSMS`. As an executable that must process external input, we expected the Messenger application to use a sandbox profile. In fact, a sandbox profile called `MobileSMS` was present on iOS 7.0 through iOS 9.2, but it was never applied to any executables. Since iOS 9.3, the `MobileSMS` profile stopped appearing on iOS.

5.7.2 Detecting Responses to Jailbreaks

We use iOracle's ability to automatically process multiple versions of iOS to detect access control patches and the iOS versions they appear in. These access control patches may take the form of

new sandbox profiles, new sandbox rules, changed behaviors of potential confused deputies, etc.

5.7.2.1 Name Resolution Jailbreak Responses

Figure 5.4 illustrates the privilege escalation attacks used by the evasi0n 7 (iOS 7) and TaiG (iOS 8) jailbreaks. These jailbreaks are discussed in more detail in Section 5.5.
evasi0n 7 – In iOS 7.0, `installd` was unsandboxed, but our queries indicate that it was assigned a sandbox profile in iOS 10.0. We found that installd no longer contained strings referencing filepaths in `tmp/` as of iOS 9.0. In a similar patch, the `afcd` sandbox profile was changed in iOS 7.1 removing its ability to access `tmp/`. iOracle detects that `CrashHousekeeping` performs chown operations on files in `/private/var/mobile/Library/Logs/` on iOS 7.0. However, in iOS 7.1, the hard coded chown operations are no longer detected.
TaiG – Through experimentation with libimobiledevice [15], we found that the `afcd` interface on iOS 9 no longer allows the creation of symlinks with `../` in the destination path. Since this symlink restriction appears to be a hard coded check built into `afcd`, it was not detected by our iOracle queries. In iOS 8.0, `BackupAgent` and `BackupAgent2` were unsandboxed, but our queries indicate that they were assigned a sandbox profile in iOS 9.0. This profile is one of the only default allow profiles, and the few operations that were denied seem focused on the filepaths exploited by TaiG. Therefore, default allow profiles can be used to disrupt known exploits while allowing all other functionality.

5.7.2.2 Capability Based Jailbreak Responses

Figure 5.5 illustrates the privilege escalation attacks used by the Pangu 8 (iOS 8) and Pangu 9 (iOS 9) jailbreaks. These jailbreaks are discussed in more detail in Section 5.5.
Pangu 8 – Comparing the sandbox profile facts of `debugserver` between iOS 8 and 9 reveals an interesting change. The `debugserver` profile in iOS 9 adds the `debug-mode` filter as a requirement for the `process-exec*` operation. Based on the Pangu 9 requirements, we assume the `debug-mode` filter requires the executed subject to possess the `get-task-allow` entitlement.
Pangu 9 – Beginning in iOS 10.0, the `container-required` entitlement was added to `neagent`. We speculate that `container-required` overrides the entitlement that assigns the `vpn-plugins` profile (`neagent` has both, but only one profile can be used). The container profile makes `neagent` significantly less useful for deploying kernel exploits.

[15]https://github.com/libimobiledevice

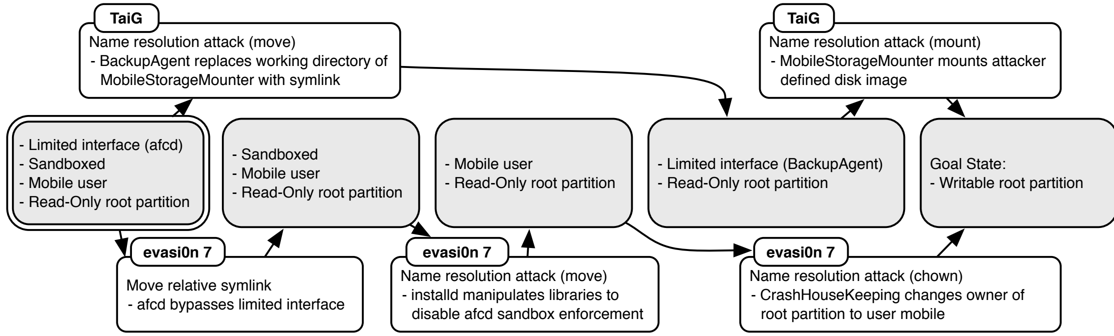

Figure 5.4 Name Resolution Based Jailbreak Steps

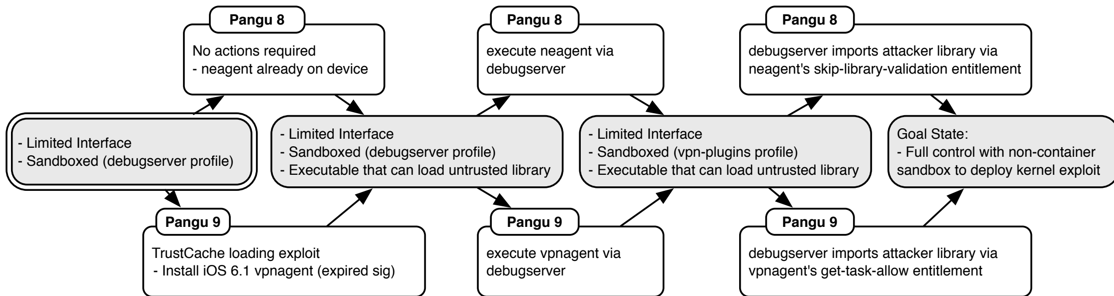

Figure 5.5 Capability Based Jailbreak Steps

5.8 Other Policy flaws

In addition to the five policy flaws presented in Section 5.6, we discovered three other flaws while implementing iOracle.

5.8.1 Denial of Service

Several system applications (e.g., Voice Memo, Camera, Safari) rely on files in the `Media/` directory. Therefore, if an attacker abuses access to `afcd`, it can disrupt the functionality of these applications. Using iOracle, we know that `afcd` has write access to all files in `Media/` and can create non-regular type files there. We also queried iOracle to detect filepaths within `Media/` that are written by system executables. As a proof of concept attack, we used `libimobiledevice` to control `afcd` on iOS 10.2 and replaced databases in `Media/Recordings/` with directories containing dummy content. If the user attempts to make an audio recording with Voice Memos, the application will fail to save the recording because the database it requires has been replaced by a directory, and it cannot delete the directory.

The impact of this vulnerability is limited. However, it emphasizes the fragility of system pro-

cesses with respect to file integrity.

5.8.2 Address Book Privacy Setting Bypass

This vulnerability was not detected by using iOracle, but rather was the result of insights gained while modeling iOS access control semantics. Apple uses sandbox extensions as revocable capabilities. However, malicious applications can resist revocation. Revocation is resisted by storing the sandbox extension token value (which only changes on system reboot) in a file or other form of persistent storage. After revocation, an application can reclaim a revoked sandbox extension by calling `sandbox_extension_consume` with the stored extension token as a parameter. We designed a proof-of-concept application that uses this technique to maintain access to the user's address book after access is revoked through privacy settings.

The impact of this vulnerability is moderate. The attack bypasses privacy settings and allows access to user data that should be protected by the sandbox. The attack also provides insight into the challenges of revocable privileges. CVE-2015-7001 and CVE-2016-4686 suggest that Apple has already increased address book security twice in attempts to prevent this type of attack.

5.8.3 Symlink Restriction Bypass

The `afcd` interface on iOS 9 does not allow the creation of links with `../` in the destination. This restriction prevents `afcd` from creating symlinks that direct to files outside of `Media/`. However, third party applications can still create symbolic links with any filepath as the destination. If a third party applications places a symlink in `Media/`, then `afcd` can create a second link that redirects to the first link without using `../` in the path. Therefore, by combining multiple symbolic links, `afcd` can create links in `Media/` that redirect to arbitrary filepaths.

We query the container profile for filepaths in `Media/` where a third party application has write access. Our queries indicate that third party applications on iOS 9.3.5 have write access to `Media/lock_sync`.[16] Therefore, a chain of links can be created by a third party application and `afcd` such that directories in `Media/` are redirected to directories under attacker control. The third party app can link `Media/lock_sync` to any destination, and `afcd` can replace files or directories in `Media/` with links to `Media/lock_sync`. For example, the following chain of links can be formed `Media/Recordings→Media/lock_sync→../../../attackerTarget`.

The impact of this vulnerability depends on its applicability to iOS 10 and the prevalence of devices restricted by their hardware to iOS 9.3.5. Third party write access to `lock_sync` was removed in iOS 10 in response to vulnerabilities detected by SandScout [Des16]. Therefore, our proof of concept does not apply to iOS 10. However, it can affect iOS 9.3.5 (the latest version supported by

[16] /private/var/mobile/Media/com.apple.itunes.lock_sync

32 bit iOS devices). On iOS 9.3.5, this vulnerability can act as a starting point in jailbreak attacks to perform name resolution attacks similar to those used in evasi0n 7 and TaiG.

5.9 Limitations

It is possible that some tools such as SandBlaster and our backtracer could produce incorrect facts. Since iOS is closed source and poorly documented, it is impractical to obtain ground truth, which limits our ability to verify the correctness of some of our extracted policies and contextual data. This limitation is inherent to working with a closed source commodity operating system. Where feasible, we mitigate these limitations through sanity checks, reproducing experiments on jailbroken and stock devices, and cross referencing literature. Our evaluation of iOracle's accuracy is based on its ability to detect known and unknown policy flaws, and we find it accurate enough for practical use.

Other limitations can be overcome with additional engineering effort and expanding our scope. The following steps would improve the accuracy of our model: 1) distinguishing between TTY and character device files; 2) modeling POSIX ACLs (added in iOS 9); 3) modeling the Unix permission directory sticky bit; 4) modeling the `filemode` sandbox filter (added in iOS 9); 5) reverse engineering differences between the `HOME` and `FRONT_USER_HOME` prefix variables; 6) incorporating `default allow` sandbox profiles into high level queries; and 7) implementing Prolog rules to identify when two regular expressions share a common matching string.

Finally, when using iOracle, analysts must have some domain-knowledge to design relevant queries. However, the Prolog rules discussed in Section 5.4.2 allow analysts to make high level queries without understanding low level details of Unix permissions or Apple Sandbox filters. These Prolog rules can be extended to model other access control mechanisms or to classify subjects and objects (e.g., list private files).

5.10 Related Work

iOracle evaluates access control for iOS system executables, whereas most prior academic iOS security research focuses on third party applications. Han et al. [Han13a] and Egele et al. [Ege11] investigate potential privacy leaks in third party iOS applications. iRiS [Den15] improves privacy leak analysis by integrating static and dynamic analysis techniques to detect dangerous API calls. XARA [Xin15] exploits flaws in iOS inter-process communication to provide a third party app with unauthorized access to sensitive data. Wang et al. [Wan14] propose a method for a compromised PC to inject malicious third party apps onto an iOS device by exploiting the iTunes syncing mechanism. Kurtz et al. [Kur16] investigate methods for third party apps to fingerprint iOS devices. SandScout [Des16] models all iOS sandbox policies, but its evaluation is limited to the policy for third party applications. Wang et al. [Wan13], and Bucicoiu et al. [Buc15] investigate Return Ori-

ented Programming (ROP) attacks in third party applications. In response to these ROP attacks, Davi et al. [Dav12], Werthmann et al. [Wer13], and Bucicoiu et al. [Buc15] propose new security mechanisms to provide control flow integrity and fine grained access control for third party apps. Han et al. [Han13b] investigate the potential for third party applications to abuse access to Private APIs. Chen et al. [Che16] detect potentially harmful Android libraries and then detect their iOS counterparts based on features shared across both platforms.

Both non-academic and academic security research has provided domain knowledge embedded into iOracle. Books by Levin [Lev16] and Miller et al. [Mil12] provide detailed descriptions of jailbreaks and security mechanisms. Several security researchers have shared findings after reverse engineering the iOS sandbox mechanism[17,18,19] [Bla11; DZ11]. Finally, Watson [Wat13] provides a survey of access control extensibility in which he discusses several access control mechanisms including iOS sandboxing.

Prior work creates logical models of access control systems. Chaudhuri et al. [Cha08] use Datalog to model dynamic access control systems (e.g., creating processes) including Windows Vista and Asbestos. SEAL [NK11], a language similar to Datalog, is designed for specifying and analyzing label-based access control systems such as Windows 7, Asbestos, and HiStar. Chen et al. [Che09] use Prolog to model and compare attack graphs for SELinux and AppArmor.

The multi-stage nature of jailbreak gadgets could be represented as state transitions in an attack graph. Sheyner et al. [She02] use the NuSMV model checker to automatically construct attack graphs representing networks. MulVAL [Ou05] uses Datalog to create a logic-based attack graph that integrates network configurations with data from reported vulnerabilities. Saha [Sah08] extended MulVAL to include complex security policies (e.g., SELinux), logical characterization of negation, and more efficient reconstruction of the attack graph after changes are made. Sawilla and Ou [SO08] develop an algorithm that uses vulnerabilities and attacker privileges to prioritize vertices in a network attack graph.

iOracle is related to prior work in Android. Gasparis et al. [Gas17] learn from legitimate rooting applications in order to detect Android malware containing rooting exploits. SEAndroid [Sma] ports SELinux to Android, and EASEAndroid [Wan15] automatically refines SEAndroid policies by using semi-supervised learning. SPOKE [Wan17] models the attack surface of SEAndroid using functional tests.

---

[17] http://www.slideshare.net/i0n1c/ruxcon-2014-stefan-esser-ios8-containers-sandboxes-and-entitlements

[18] http://2013.zeronights.org/includes/docs/Meder_Kydyraliev_-_Mining_Mach_Services_within_OS_X_Sandbox.pdf

[19] http://newosxbook.com/files/HITSB.pdf

5.11 Conclusions

In order to automate the evaluation of the iOS protection system, we constructed iOracle. Working with a closed-source system, we modeled the iOS protection system to detect policy flaws. We performed a case study of four recent jailbreaks and iOracle helped detect the executables exploited by them. Finally, iOracle has led us to five previously undiscovered policy flaws.

iOS access control must continue to increase in complexity in order to meet the demands of new features and increasingly sophisticated attacks. The iOracle framework allows security researchers to scale analysis efforts to keep pace with increasing complexity.

5.12 Acknowledgments

We thank Micah Bushouse, Brad Reaves, and the WolfPack Security and Privacy Research (WSPR) lab as a whole for their helpful comments. We also thank Dennis Bahler for his advice on Prolog and other logic programming languages.

This work was supported in part by the Army Research Office (ARO) grants W911NF-16-1-0299 and W911NF-16-1-0127, the National Science Foundation (NSF) CAREER grant CNS-1253346. This work has been co-funded by the DFG as part of projects P3, S2 and E4 within the CRC 1119 CROSSING. This work has been funded by University Politehnica of Bucharest, through the "Excellence Research Grants" Program, UPB-GEX2017, Ctr. No. 19/2017. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

CHAPTER

6

A HOUSE OF MANY DOORS: EVALUATING ACCESS CONTROL FOR REMOTE NSXPC METHODS ON IOS

6.1 Introduction

Apple's iOS App Store offers over 2 million applications [Appb], and in 2017, the App Store was used by half a billion customers per week [Appa]. Apple's customers can be comforted by the fact that iOS applications implement multiple accesss control features to mitigate malicious behaviors. For example, a third party application must run within the confines of a sandbox that limits the resources it can directly access. Instead, iOS applications access sensitive resources indirectly through Inter-Process Communication (IPC) Services. For example, a third party application does not have direct access to a user's calendar, but can use services provided by a calendar managing daemon. However, a third party application can still ask a more privileged process to perform some action that damages the system or violates the user's privacy. This type of malicious behavior is called a confused deputy attack.

IPC-Based confused deputy attacks are not a new problem. Grace et al. [Gra12] implemented Woodpecker, which uses data-flow analysis on pre-loaded Android applications to enumerate dangerous services exposed to other applications. However, several features (e.g., dynamic dispatching

for method calls) make data flow analysis less practical for iOS binaries. To the best of our knowledge, no systematic enumeration of remote methods accessible to third party apps has been performed on iOS. Existing IPC fuzzers for iOS [Wan; Bee; Kyd], probe for code flaws such as type confusion or dereferencing vulnerabilities that can be exploited to obtain arbitrary code execution. These fuzzers do not attempt to enumerate remote methods or identify confused deputy attacks. Sand-Scout [Des16], and iOracle [Des18], detect access control policy flaws in iOS, but their scopes are limited to file system operations.

IPC on iOS often uses capability requirements and standardized interface abstractions to increase security and simplify development. Service providing processes can use hard coded conditions allow or deny access to services based on a client's capabilities (i.e. observable attributes of the client). These capabilities are usually in the form of entitlements, immutable key-value pairs bound to an executable's code signature at compile time. Standardized interface abstraction, allows developers to call library functions instead of implementing complex message serialization and dispatch schemes. While several IPC interface types can be found in iOS for legacy reasons, the state-of-the-art interface type is called XPC.[1] The object-oriented version of XPC is referred to as NSXPC (Next Step XPC).

Given the current state of IPC on iOS and the dearth of research on the topic, we seek to answer the following research question: "Which security and privacy sensitive NSXPC methods are accessible to third party applications?"

Answering this question requires addressing three research challenges. First, We must define the set of entitlements available to third party applications. We find that there are two sets of entitlements available third party applications, a public set accessible to all developers, and a semi-private set that Apple provides only to select developers. For example, the Uber application was found to possess a potentially dangerous, entitlement normally unavailable to third party applications [Ube]. Second, we must enumerate the set of NSXPC services accessible to third party applications. The executables that provide these services are closed source and there is no centralized policy mapping services to their entitlement requirements. Third, we must determine which NSXPC services are security or privacy sensitive. The semantics of these services are not publicly documented and as noted previously, data flow analysis is less practical on iOS than on Android.

In this paper, we present *Kobold*,[2] a framework for studying NSXPC services in iOS. Kobold uses binary program analysis to enumerate the space of NSXPC services accessible to third party applications and dynamically tests those services to detect potential confused deputy attacks. Kobold leverages two key insights. First, programs that use standardized IPC interfaces (e.g., NSXPC) contain predictable patterns in their compiled code that are identifiable via static analysis. Second, error messages returned by unauthorized attempts to access IPC services can provide a model of the iOS IPC access control policy. Kobold demonstrates that pattern-based static analysis can be

[1] To the best of our knowledge, Apple has not expanded this acronym

[2] A spirit from German folklore that haunts houses, mines, and ships.

use to enumerate IPC implemented with standardized interface abstractions. Kobold also shows that combining this enumeration with feedback-driven fuzzing is a practical approach to inferring decentralized access control policies.

This paper makes the following contributions:

- *We present Kobold, the first framework for evaluating NSXPC access control policies.* Kobold enumerates the NSXPC services accessible to third party applications and applies automated heuristics to determine which services are likely to be exploited.

- *We perform the first measurement of third party application entitlements.* We analyze approximately six thousand popular third party applications to determine which semi-private entitlements Apple distributes to an undisclosed subset of third party developers.

- *We identify previously unknown security issues including three categories of confused deputy attacks and fourteen daemon crashes.* Our findings include crashes for root authority daemons, unprivileged access to Mobile Device Management (MDM) functionality, and activation of the microphone without user permission.

The remaining sections are outlined as follows. Section 6.2 provides background information on iOS IPC and access control. Section 6.3 overviews Kobold and our findings. Section 6.4 details the implementation of Kobold. Section 6.5 presents the results of the third party entitlement survey. Section 6.6 analyzes the ports, methods, arguments, and entitlement requirements enumerated by Kobold. Section 6.7 demonstrates Kobold's ability to detect previously unknown IPC policy flaws and crashes. Section 6.8 provides Kobold's limitations. Section 6.9 discusses related work. Section 6.10 concludes.

6.2 Background

iOS is Apple's operating system for mobile devices (i.e., iPhone, iPad, iPod). It is very similar to macOS, watchOS, and tvOS which are all based on the XNU (X is Not Unix) kernel. XNU is a hybrid kernel that combines the Mach microkernel, FreeBSD, and a driver framework called I/O Kit. Mach provides much of the Inter-Process Communication (IPC) functionality through mach-messages. FreeBSD provides the file system and the TrustedBSD Mandatory Access Control (MAC) Framework, which allows Apple to hook system calls and implement sandboxing. Finally, as interfaces between user space and kernel space, I/O Kit drivers are often the target of fuzzing. The remainder of this section will explain Mach IPC and access control mechanisms that regulate IPC on iOS.

6.2.1 Mach IPC

At a fundamental level, IPC on iOS is built upon the Mach microkernel. The primitive components of Mach IPC are mach-messages and mach-ports. A mach-port is a unidirectional queue with one receiver and one or more senders. Bidirectional communication (e.g., remote methods with replies) is accomplished by using two mach-ports, one for each direction. Mach-Messages are sent to mach-ports, where they can accumulate until the receiver is ready to process them by popping them from the queue. These mach-messages can contain instructions for remote method providers. The message's instruction could indicate a function to run, the parameters to use, and information to help send a reply when the function completes.

A server process can host remote methods by registering a name for a mach-port, and clients can send messages to that port in order to request the remote methods. The mach-port name registration is facilitated by launchd, which also assists clients in connecting to mach-ports. For example, the location daemon, locationd, offers remote methods on the mach-port named "com.apple.locationd.-registration". A client process can access these methods by asking launchd to connect it to the "com.apple.locationd.registration" mach-port. If the connection is successful, the client can then send messages to the server via the mach-port. If the messages are well formed, and the client has sufficient capabilities, the server will the execute the methods for the client (e.g., locationd could provide access to the user's coordinates).

A process can register zero or more mach-ports, and each of these mach-ports can be associated with one or more remote methods. For example, a remote method provider can link a mach-port to a dispatch table representing multiple methods. When the remote method provider receives a mach-message, the message is parsed and triggers a specific function to run based on the dispatch table. Therefore, one process can register multiple mach-port names, that are each associated with one or more remote methods.

The process of encoding and decoding mach-messages is complex, error prone, and security sensitive. Therefore, interface abstractions are provided by Apple to make it easier to develop clients and remote method providers. At the time of writing, the state-of-the-art interface type is XPC and its object oriented variant, NSXPC. In object-oriented IPC, an object and its methods reside in the remote method providing process, but the client can access the object as though it existed in the client's address space. For example, once a client connects to a mach-port hosting an NSXPC remote method, that client can invoke methods of a remote object by using the same syntax used for local objects. Therefore, a remote method providing process using NSXPC can register multiple mach-port names that each provide access to remote objects, and each remote object exposes one or more remote methods.

In order to mitigate type confusion attacks [Fer], remote methods exposed with NSXPC have

strict parameter types that must adhere to a protocol called NSSecureCoding.[3] Any attempts to invoke these methods with invalid arguments or vague parameter types are immediately rejected. Therefore, Kobold must perform three tasks: 1) identify the mach-ports associated with NSXPC interfaces; 2) find the names of remote methods provided by remote objects; and 3) obtain the expected arguments of those remote methods.

6.2.2 IPC Access Control

The App Store is the only public source of applications for non-jailbroken devices, which provides Apple with centralized control over iOS software. Only those applications that pass Apple's vetting process can enter the app store, and those found to be problematic can be quickly removed. iOS will only run applications that have been signed by a legitimate developer certificate, and attempts to modify code should invalidate the application signature. However, code signing and app vetting are not sufficient to stop all attacks, and researchers have demonstrated several attacks to bypass these defenses [Wan13; Han13b; Xin15]. In order to mitigate such attacks, Apple has implemented multiple layers of access control including capability systems and the sandbox.

6.2.2.1 Entitlements

The capability most relevant to IPC access control is called an entitlement. Entitlements are key-value pairs statically embedded into an executable's code signature. The entitlement key is a string representing some semantic clues to the entitlement's functionality (e.g., inter-app-audio). The entitlement value represents a description of the key and has a data type (e.g., boolean, string, array) based on the purpose of the entitlement.

Apple uses entitlements to help determine which privileges are accessible to each application. The most dangerous entitlements (e.g., bypassing code-signing restrictions) are private and reserved for executables created by Apple. Less sensitive entitlements (e.g., inter-app audio) are publicly available to third party developers who can add them to apps by toggling switches in Xcode during development. A third, poorly understood class of semi-private[4] entitlements are not available through Xcode toggles, but can still be found in a number of third party apps on App Store. Since entitlements determine the privileges of a process, understanding them is critical when evaluating attack surfaces on iOS.

Ideally, each application would be clearly labeled with its entitlements and documentation explaining the privileges provided by those entitlements. However, entitlements can not be seen without using binary analysis tools, and very little documentation exists even for the publicly

[3] https://developer.apple.com/documentation/foundation/nssecurecoding?language=objc

[4] https://forums.developer.apple.com/thread/77704

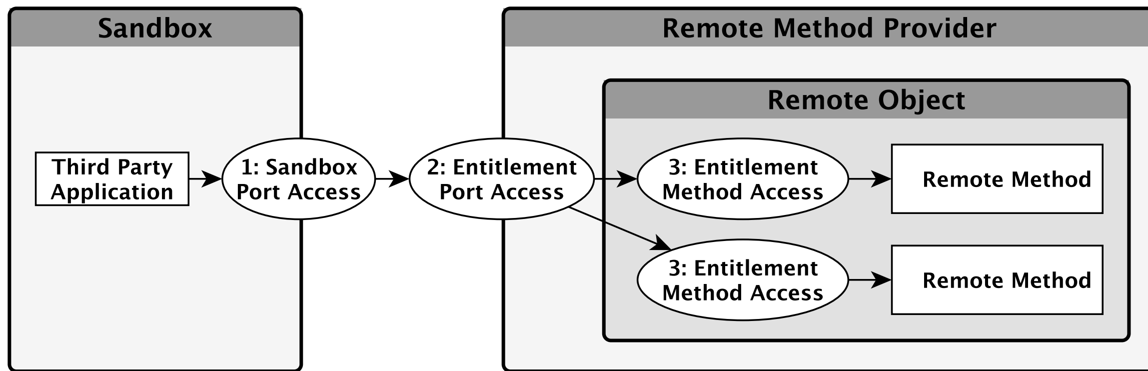

Figure 6.1 Three Stages of NSXPC Access Control: 1) Sandbox Access to Port; 2) Entitlement Checks for Port; 3) Entitlement checks for Remote Method

available entitlements. While an end user may be comfortable with the entitlements listed in Xcode, any application they install could possess a more dangerous, semi-private entitlement with no warning to the user. Due to the distribution of undocumented semi-private entitlements, proving that an entitlement is private (accessible only to Apple) requires a proof of negation showing that no third party application on the App Store has the entitlement. Therefore, it is not possible to determine which entitlements are semi-private without analyzing all 2.2 million apps in the App store. Although an app cannot change entitlements without violating the code signature, there is an opportunity for developers to set different entitlements as part of an application update, which involves signing the new version of the app. Therefore, Apple can grant new semi-private entitlements at any time to new applications or as part of an application's update.

6.2.2.2 Enforcement

Figure 6.1 illustrates the three locations of NSXPC IPC access control enforcement. At stage one, the sandbox can allow or deny requests to connect to specific mach-port names. At stage two, the remote method provider can accept or deny attempts to connect to one of its mach-ports based on the client's capabilities. Finally, at stage three, each remote method can accept or deny attempts to invoke them based on the client's capabilities.

The Apple sandbox regulates the system calls available to sandboxed applications (e.g., third party applications). However, this restriction is at a coarse level of granularity for IPC and only allows or denies access to mach-port names. Therefore, if a mach-port is associated with multiple NSXPC methods, the sandbox must allow access to zero or all of these methods. At stage one of Figure 6.1, a third party application attempts to connect to a mach-port, which invokes a system call that the sandbox can allow or deny. This decision is made based on the default sandbox policy for third party applications, which is called "container". The sandbox policy can also have conditional

rules that allow an application to connect to mach-ports only if the application possesses certain entitlements. For example, the following simplified sandbox policy snippet will only allow access to the “siri.vocabularyupdates” mach-port if the app has the “siri.synapse” entitlement. Note that entitlements are key-value pairs, but if only the entitlement key is specified, it is assumed that the entitlement value must be a boolean type assigned to True.

```
(allow mach-lookup
  (require-all
    (global-name "siri.vocabularyupdates")
    (require-entitlement "siri.synapse")))
```

Unlike the sandbox policy, the next two stages of access control are not defined by a centralized policy (e.g., a sandbox policy). Instead, the enforcement is based on ad-hoc conditions hard-coded into remote method providers. Remote method providers can check the entitlements of clients by using the SecTaskCopyValueForEntitlement API [Lev]. This API allows a process to specify an entitlement key and a client, and the API will return the value associated with that entitlement key for the specified client. Then the remote method provider can decide whether or not to provide access based on the returned key value (e.g., True, False, “read-only”). While this type of policy is intuitive to develop, it is difficult to verify correctness of the policy due to its decentralized nature.

At stage two of Figure 6.1, the third party application has made the system call to connect to a mach-port, but the remote method provider can still reject the connection. This decision is made based on hard-coded conditions that check the attributes of the client (e.g., Unix user authority or entitlements). At stage three, the connection has been approved, but each of the remote methods may contain hard-coded conditions that govern access to that method. For example, one mach-port could expose a remote object with 9 remote methods that do not require any entitlements and 1 remote method that requires the client to possess a specific entitlement. This level of granularity allows developers to assign remote methods with similar functionality to the same mach-port while applying different access control policies based on their security sensitivity.

6.3 Overview

With the risk of confused deputy attacks, IPC is an important attack surface for iOS. However, due to the closed-source nature of iOS and decentralized conditions hard coded into service providers, this attack surface is difficult to enumerate. Even after enumerating the IPC services accessible to third party applications, the security sensitivity of each service must be evaluated. For example, a service that allows a client to move arbitrary files is dangerous, a service that resets other processes is annoying, and a service that accepts invitations to multiplayer games may be harmless. The remainder of this section steps through Figure 6.2 to overview our approach to addressing these

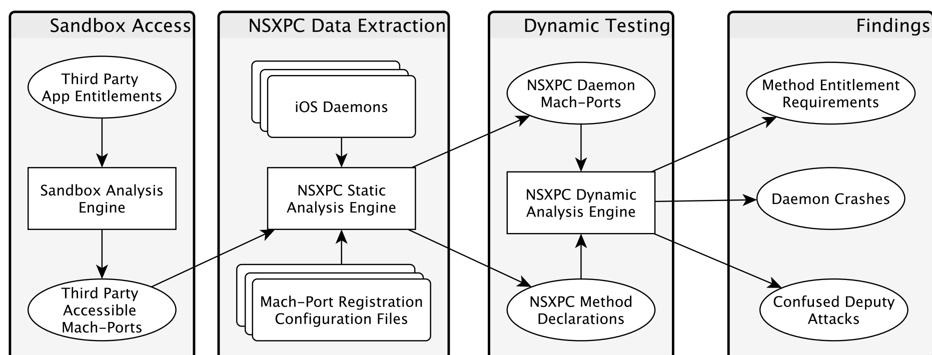

Figure 6.2 Kobold Overview

challenges and answering the research question, “Which security and privacy sensitive NSXPC methods are accessible to third party applications?”

Input: Kobold requires three sources of input, an iOS firmware image, a sample set of third party applications, and an iOS device (optionally jailbroken). iOS firmware images are publicly available from Apple(cite) and contain executable binaries and configuration files that can be statically analyzed to extract NSXPC service information. Kobold automatically scrapes a sample set of third party applications from the App Store to approximate the set of semi-private entitlements available to third party developers. Kobold’s automated scraper collects approximately 6000 of the most popular applications. Finally, the iOS device is required for dynamic analysis, and a jailbreak expands these dynamic analysis options.

Static Analysis: Kobold uses static analysis of iOS firmware and third party applications to infer sets of semi-private entitlements, accessible mach-ports, and remote methods. Third party applications are automatically scraped from the app store. Then, entitlement keys and values are extracted from each scraped third party application in order to survey the distribution of semi-private entitlements. Kobold reverse engineers and models the “container” sandbox policy, which regulates system calls (e.g., mach-port connections) for third party applications. Kobold also creates a model of a third party application possessing all public and semi-private entitlements and cross references it with entitlement requirements found in the modeled “container” sandbox policy. Kobold uses these two models to infer the mach-ports accessible to a third-party application with respect to the sandbox (i.e., Stage 1 of Figure 6.1). These mach-ports are mapped to the executable files that provide them by analyzing configuration files extracted from iOS firmware images. Finally, NSXPC service headers containing remote methods and their parameter types are statically extracted from the executable binaries of system daemons present in iOS firmware images.

Dynamic Analysis: Kobold requires an iOS device for dynamic analysis since the iPhone simulator provided by Xcode does not sufficiently represent IPC services. Directly accessing NSXPC services

with an iOS application is undocumented and forbidden by compile-time checks in Xcode. However, in Section 6.4, we bypass Xcode's compile-time checks and create a custom iOS application written in Objective-C for directly invoking NSXPC services. Kobold uses this remote NSXPC method invoking application as a proof-of-concept third party application abusing access to the mach-ports and remote methods we detected during static analysis. Many of these remote methods will trigger error messages that contain useful content for our analysis (e.g., "Error: the 'mobileinstall' entitlement key with a string 'enumerateInstalledApps' value is required to access this method"). Without Apple's explicit permission, we cannot legitimately create an application with semi-private entitlements. However, we can use a jailbroken device to bypass this restriction. Therefore, we can dynamically test the effects of invoking methods with and without arbitrary entitlements in our application's signature. While NSXPC has strict parameter *type* requirements, we can vary the parameter *values* in order to trigger errors, method reply messages, user perceivable device activity, and daemon crashes. Crash reports triggered by remote method invocation can be viewed on a stock device through the Settings menu. Additional dynamic analysis techniques can be deployed on a jailbroken device (e.g., monitoring file operations) to provide additional insight into the system activity triggered by the app's service invocations.

Results: Kobold uses the output of static and dynamic analysis to infer the sets of NSXPC services that are both security sensitive and accessible to third party applications. Kobold identifies 17 semi-private entitlements, 123 third party accessible remote NSXPC methods, 14 daemon crashes, and 3 categories of confused deputy attack.

The entitlement survey detects multiple undocumented semi-private entitlements. For example, we found that the Netflix application possesses an undocumented semi-private entitlement related to video streaming. Among the applications in our sample, we found that this entitlement is only possessed by Netflix. Other video streaming applications in our sample set (e.g., Hulu and Amazon Prime Video) do not use this entitlement.

Kobold led us to discover three confused deputy attacks. First, the File Provider daemon can be prompted to leak state information. This information allows a third party application to infer potentially private information about applications that infer File Provider functionality (e.g., inferring the names of files stored in file providers). Second, remote methods related to voice dictation services can be abused to activate the microphone without user permission. Third, inconsistent entitlement requirements in MDM services allow third party applications (without MDM entitlements) to block website access and disable keyboard features on iOS devices that have not been enrolled for MDM.

In total, 14 unique crashes were detected while dynamically testing the enumerated remote methods. These crashes affect 10 daemons with some daemons crashing with multiple unique stack traces depending on the method called. Three of the crashing daemons run with root authority (i.e., locationd, powerlogHelper, and UserEventAgent). One crash revealed an inconsistently enforced entitlement requirement.

6.4 Kobold

We divide our implementation of Kobold into three tasks it automates. First, it performs a survey of the entitlements available to third party applications. Second, it enumerates the NSXPC services accessible to third party applications. Third, it evaluates the security sensitivity of accessible NSXPC services in order to highlight services likely to allow confused deputy attacks.

6.4.1 Identify Third Party Entitlements

Since entitlement requirements in the sandbox can determine the set of mach-ports accessible to third party applications, our first step is to enumerate the entitlements that a third party application can possess. Kobold's entitlement surveying framework consists of two stages. First, we automatically download the .ipa (iPhone application archive) files representing iOS applications from the Apple App Store. Second, we extract metadata and entitlement data from each .ipa file and model the data as Prolog facts.

Modeling Sandbox Entitlement Checks: Kobold expands upon an existing model of iOS access control called iOracle [Des18] in two ways: 1) modeling sandbox rules for mach-port access; and 2) modeling the entitlements available to third party applications. iOracle provides a logical model of file system operations by modeling the iOS sandbox and Unix Permissions as a collection of Prolog facts and rules. Since iOracle provides a general framework for modeling sandbox rules, Kobold expands upon this framework with new Prolog facts and rules modeling mach-port access in the sandbox and third party entitlements. By combining the model of third party entitlements and the model of sandbox rules for mach-port access, Kobold can automatically map third party accessible entitlements to sandbox rules that require those entitlements. Kobold also uses Prolog queries to determine which entitlements are public and semi-private.

We developed an app scraper for Kobold, but we do not claim to have developed the first App Store scraper. Kobold uses accessibility options and AppleScript to manipulate iTunes on macOS, and prior work by Orikogbo et al. [Ori16] uses a Windows virtual machine to manipulate iTunes. We collected our app samples in September 2017. Apple has officially removed the iOS app market from the default version of iTunes, but installing[5] an alternative version[6] will restore the iOS app market functionality.

Kobold's app scraper collects applications in three steps. First, A web scraper collects all application identifier numbers from Apple's online index of applications[7] and identifies the 240 most popular applications of each app genre. Second, these app identifier numbers are used to create

[5]https://www.macworld.com/article/3230135/software-entertainment/how-to-install-itunes-1263-and-replace-itunes-127.html

[6]https://support.apple.com/en-us/HT208079

[7]https://itunes.apple.com/us/genre/ios/id36?mt=8

URLs[8] that request app metadata (e.g., app price, developer name) for each application available in the US through an iTunes API.[9] Third, we use a combination of deeplinks within iTunes, accessibility options, and AppleScript to automatically download free applications in the form of .ipa files.

The "unzip" utility can be used to extract a metadata file called iTunesMetadata.plist, a .app file (another archive), and other supporting files from a .ipa file. While application binaries are encrypted, the encryption only affects the code section of the Mach-O executable, so the entitlements can be extracted without decrypting. We use the codesign utility to extract entitlements and use custom bash scripts to parse the results and reformat them into Prolog facts. An application's .ipa file name may not indicate the actual name of the application, so we extract relevant data from the app's iTunesMetadata.plist file. Since this file is formatted as a plist (Property List), we use custom scripts and the plistlib[10] Python library to extract the app's ID number, bundle id, and developer name. This information is merged into the Prolog facts for each entitlement creating facts with the following format:

```
appStoreEntitlement(
    itemId("408709785"),
    artistName("Apple"),
    bundleId("com.apple.mobilegarageband"),
    collection("fall2017"),
    entitlement(key("inter-app-audio"),
    value(bool("true")))).
```

To determine which entitlements are public (available to all third parties), we internally developed an application in Xcode with all capability toggles enabled, and built a release version of the app. We then used the macOS codesign utility to extract all of the entitlements from the internally developed application and label the resulting keys as public entitlement keys.

To identify semi-private entitlement keys we query for those entitlement keys possessed by the collection of third party applications (i.e., developer name is not "Apple"), and we remove any entitlement keys already labelled as public. The semi-private entitlement keys (available to some third parties) that remain are possessed by third party applications in the wild, but are not accessible to our internally developed application.

6.4.2 Enumerate Accessible NSXPC Services

To invoke an NSXPC service, a client must correctly specify two targets, a mach-port name and a remote method associated with the service. Kobold uses two static analysis techniques and two dynamic analysis techniques to find these mach-ports and methods in order to enumerate the

[8] https://itunes.apple.com/lookup?id=553834731
[9] https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/
[10] https://docs.python.org/2/library/plistlib.html

NSXPC services accessible to third party applications. First, the mach-ports that the "container" sandbox policy allows access to are extracted statically. Second, protocol headers likely to contain remote methods for NSXPC services are extracted statically. Third, an internally developed application dynamically attempts to connect to the statically extracted mach-ports. Fourth, the internally developed application also attempts to invoke each of the statically extracted methods.

Since the "container" sandbox policy for third party applications is a whitelist, it explicitly lists which mach-port names a third party application can connect to. The sandbox policy can be found in the iOS firmware image, but is precompiled into a proprietary binary format. Therefore, we use SandScout [Des16] to reverse engineer and model the "container" policy and query for this list of accessible mach-port names. However, some of the sandbox rules governing access to mach-ports are conditioned on entitlement requirements. We expand upon a Prolog powered iOS access control modeling framework called iOracle [Des18] to map public and semi-private entitlements to these requirements to infer a larger set of mach-ports that are accessible through the sandbox.

Since NSXPC is an object-oriented interface, both the client and service provider are expected to have a list of method declarations called a protocol. However, these protocols are not publicly available and must be extracted from the binary executables of service providers found on the iOS firmware image. We use a static analysis tool called class-dump[11] to extract object-oriented features (i.e. protocol method declarations) from iOS daemon executable files. Using class-dump, we search service providers for interface classes associated with either NSXPCConnection or NSXPCListener classes. Class-dump extracts the protocols implemented by these interfaces, and we extract method declarations from those protocols. The extracted protocols are not guaranteed to represent NSXPC services, but we treat them as an over-approximation that can be refined through dynamic testing.

The next step is to map mach-ports to the remote methods they provide access to. Protocols can be trivially mapped to executables since the protocols were extracted by running class-dump on a specific executable. However, the mach-ports were collected from the sandbox policy which does not express the executable that uses each mach-port. This mapping of mach-ports to executables can be obtained statically by analyzing a cache of mach-port name registrations stored in "xpcd_-cache.dylib". However, parsing this file is non-trivial. First, kobold identifies a section in the .dylib binary format that represents a .plist file and extracts that section using jtool.[12] This plist file is then converted from a binary format into xml by using the plutil utility. Finally, the xml formatted plist file can be parsed with regular expressions to extract a mapping of service providing executables to the mach-ports they use.

At this point, mach-ports have been mapped to executables, and protocols have been mapped to executables. However, an executable could use more than one mach-port and more than one protocol. Therefore, we have significantly reduced the possible combinations, but we still need

[11]http://stevenygard.com/projects/class-dump/
[12]http://www.newosxbook.com/tools/jtool.html

to disambiguate invalid mach-port to protocol combinations within an executable. Kobold addresses this ambiguity by attempting each combination at run time and using message feedback to determine which mach-port to protocol combinations are valid.

The Xcode IDE for iOS forbids developers from calling NSXPC APIs in their code. However, through reverse engineering we have confirmed that system programs on iOS do use NSXPC. Therefore, the libraries for NSXPC exist on the iOS device, but Xcode acts as a compile-time obstacle to discourage malicious or accidental abuse of low level functionality. Fortunately for us, Xcode's policy for method restrictions is not tamperproof. Investigating the NSXPC header file[13] in the iOS SDK (Software Development Kit) revealed that the NSXPC API we needed was augmented with the tag __IOS_PROHIBITED. Removing these tags from the header file allowed us to use Xcode to compile applications using NSXPC APIs.

The mach-ports extracted through static analysis of the sandbox policy only represent stage 1 of Figure 6.1. In order to detect mach-ports that are inaccessible for reasons other than the sandbox (e.g., the service is no longer provided or there are hard-coded in the service provider), we use a custom iOS application to attempt to connect to each sandbox accessible mach-port. The NSXPC API to connect to mach-port does not require any remote methods to be specified. If a mach-port connection fails, an "interruptionHandler" error is thrown. If we do not receive such an error, we assume the connection was successful and can proceed to calling remote methods.

Once a connection to a mach-port is made, NSXPC allows a client to call remote methods associated with that mach-port. Many methods contain a special parameter called completions handler that contains zero or more arguments and a block of code to be executed if the remote method completes. Kobold calls all remote methods associated with the protocols extracted and assume that those message which trigger completion handler message are accessible unless those completion handlers return error messages. If an error occurs, a helpful message describing the problem may be available in the error field of the method's completion handler. We speculate that these error messages were only intended for Apple developers since third parties are not expected to use the NSXPC APIs. However, we have found these completion handler errors to provide valuable insights since they may specify the entitlement key and value required for the method being called.

6.4.3 Evaluate Security Sensitivity of NSXPC Services

We use four methods to evaluate the security sensitivity of each remote method: 1) we use method name semantics and entitlement requirement inconsistencies to triage methods worth investigating manually; 2) we manually investigate values returned via completion handler arguments; 3) we ob-

[13] /Applications/Xcode.app/Contents/Developer/Platforms
/iPhoneOS.platform/Developer/SDKs/
iPhoneOS11.3.sdk/System/Library/Frameworks/
Foundation.framework/Headers/NSXPCConnection.h

serve user perceivable changes on the device; 4) we use a jailbroken device to provide supplemental insight into file operations and crash logs.

Apple has not obfuscated the names of the remote methods we extract, and objective-c requires each parameter to be mentioned in the method name. Therefore, the method names contain a significant amount of semantic information related to their functionality. For example, we would manually classify the following method declaration as security sensitive due to it's association with the camera.

```
(void)captureStillImage:(NSString *)arg1
  forCameraIds:(NSArray *)arg2
  withReply:(void (^)(int))arg3;
```

With entitlement checks occurring for each method instead of for each protocol, each method is an opportunity for developers to make access control mistakes. We assume that each method in a protocol has a similar level of security sensitivity. We also assume that methods that require entitlements are security sensitive. Therefore, any method that does not require an entitlement and shares a protocol with a method that does require an entitlement is considered to be security sensitive. For example, if a protocol has 9 methods that require an entitlement and one that does not, we assume that a developer may have forgotten to add an entitlement requirement to the unrestricted method. Our invocation of remote methods primarily uses uninitialized variables, but a part of our manual investigation does include attempts to initialize method variables with valid values (e.g., a valid file handle).

Many methods contain parameters that represent return values, and these values may contain security sensitive data after the method finishes executing. Such values are manually investigated to determine if they contain privacy sensitive information.

A significant amount of system activity can be observed when fuzzing remote methods on a non-jailbroken device. If invalid method arguments cause device features to be disrupted (e.g., internet access, configuration options) a human observer may detect these changes by manually investigating the device state after fuzzing the enumerated remote methods. Affects such as sounds or prompts that occur while running the fuzzing application can also be documented by a human observer. Finally, a collection of crash reports are visible to iOS users through the Settings menu. These reports can be used to detect crashes caused by method invocation on stock or jailbroken devices.

We perform two types on dynamic analysis to observe system activity on a jailbroken device. First, we use filemon to track all file operations (i.e., the process, file source, file destination, and operation type) on the device. Second, we track changes made to crash logs stored in the "/private/var/-mobile/Library/Logs/AppleSupport/" and "/private/var/mobile/Library/Logs/CrashReporter/" directories.

6.5 Entitlement Survey Results

There are 25 app genres listing the top 240 most popular applications for the United States in each genre for a total of 6000 applications. Of the 6000 popular apps, there is overlap between genres (e.g., the same app might be listed under Games and Lifestyle), and only 5873 of the popular applications were unique. Of the 5873 unique applications, 5716 of were free, and we did not collect any paid applications. Of the 5716 free applications, 16 gave error messages stating that they were not currently available in the United States, and we were able to download the other 5700 applications. We speculate that these applications were revoked from the US app store, but are still indexed as popular. Our final sample set consisted of 5700 popular, free applications currently available in the US. 17 of the 5700 applications list Apple as the developer, so we label the remaining 5683 as third party applications. For public and semi-private entitlements we only count third party applications possessing these entitlements.

For each type of entitlement (i.e., public and semi-private) we have created tables listing the entitlement key, entitlement value data type, and the number of applications in our sample that possess an entitlement with the respective key. For example, an entitlement with the bool value type will either be true or false, but the string data type would allow more complex semantics making the value an important part of evaluating the entitlements effects.

6.5.1 Public Entitlements

We label 25 entitlements as public (available by default or through Xcode toggles during development). These entitlements are shown in Table 6.1. While obtaining these entitlements is straightforward for developers, end users are never made aware of these entitlements, and developers may request more public entitlements than necessary.

Three entitlements in Table 6.1 are used by 1 or 0 applications. get-task-allow allows debugging of the application, and is only available to developers for development configurations of apps. Once an application is submitted to the app store, it will not be able to have the get-task-allow entitlement since the debugging capabilities would bypass code security mechanisms. nfc.readersession[14] is related to Near Field Communication (NFC) functionality, which seems unused by any of the applications in our sample. The networking.multipath[15] allows for increased network robustness and the Ebay[16] application is the only application that uses it.

It seems that every application has an application-identifier entitlement. This allows service providing processes and the sandbox profile to condition rules based on a unique identifier of the subject application. It is possible that adhoc checks in iOS could grant privileges to third party apps

[14]com.apple.developer.nfc.readersession.formats

[15]com.apple.developer.networking.multipath

[16]https://itunes.apple.com/us/app/ebay-buy-sell-find-deals/id282614216?mt=8

Table 6.1 Public Entitlements

Entitlement Key	Value Type	Third Party Apps
com.apple.developer.nfc.readersession.formats	arrayOfStrings	0
get-task-allow	bool	0
com.apple.developer.networking.multipath	bool	1
com.apple.developer.networking.HotspotConfiguration	bool	2
com.apple.developer.homekit	bool	3
com.apple.developer.networking.networkextension	arrayOfStrings	31
com.apple.developer.networking.vpn.api	arrayOfStrings	31
inter-app-audio	bool	31
com.apple.external-accessory.wireless-configuration	bool	56
com.apple.developer.siri	bool	68
com.apple.developer.default-data-protection	string	111
com.apple.developer.healthkit	bool	195
com.apple.developer.in-app-payments	arrayOfStrings	252
com.apple.developer.pass-type-identifiers	arrayOfStrings	262
com.apple.developer.ubiquity-container-identifiers	arrayOfStrings	396
com.apple.developer.icloud-services	arrayOfStrings or string	422
com.apple.developer.icloud-container-environment	arrayOfStrings or string	436
com.apple.developer.ubiquity-kvstore-identifier	string	470
com.apple.developer.icloud-container-identifiers	arrayOfStrings	574
com.apple.security.application-groups	arrayOfStrings	1373
com.apple.developer.associated-domains	arrayOfStrings	1376
keychain-access-groups	arrayOfStrings or string	3137
aps-environment	string	3740
com.apple.developer.team-identifier	arrayOfStrings or string	5269
application-identifier	string	5683

based on the application-identifier.

The homekit and healthkit entitlements have redundant functionality with Privacy Settings and seem to represent a defense in depth approach to protecting highly sensitive resources. For an application to access the healthkit or homekit, Apple must approve the entitlements during app vetting, and the user must grant access through Privacy Settings.

The inter-app audio entitlement allows multiple applications to read and modify audio signals in a shared memory section. This entitlement is generally used by applications that modify live music or add instrumental sounds to music. However, the Mickey's Magical Math World[17] game possesses the entitlement without any clear justification. Without any clear labelling, users do not have the opportunity to question such unusual entitlements, and over-privilege is likely to go unnoticed.

6.5.2 Semi-Private Entitlements

We found 17 semi-private entitlements, and to the best of our knowledge, only 4 of them have clear documentation from Apple about the process of requesting them to be added to an app.

[17] https://itunes.apple.com/us/app/mickeys-magical-math-world/id925134465?mt=8

Table 6.2 Semi-Private Entitlements

Entitlement Key	Value Type	Third Party Apps
com.apple.accounts.flickr.defaultaccess	bool	1
com.apple.accounts.twitter.defaultaccess	bool	1
com.apple.accounts.vimeo.defaultaccess	bool	1
com.apple.coremedia.allow-mpeg4streaming	bool	1
com.apple.private.allow-explicit-graphics-priority	bool	1
com.apple.developer.healthkit.nikefuel-source	bool	3
com.apple.developer.legacyvoip	bool	3
com.apple.developer.passkit.pass-presentation-suppression	bool	3
com.apple.networking.vpn.configuration	arrayOfStrings	4
com.apple.payment.pass-access	bool	4
com.apple.accounts.facebook.defaultaccess	bool	7
com.apple.developer.payment-pass-provisioning	bool	7
previous-application-identifiers	arrayOfStrings	8
com.apple.developer.playable-content	bool	23
com.apple.developer.networking.HotspotHelper	bool	28
com.apple.developer.video-subscriber-single-sign-on	bool	39
com.apple.smoot.subscriptionservice	bool	50

The process requires sending an email to a specific team within Apple to request access. The four applications publicly documented as semi-private are pass-presentation-suppression,[18] payment-pass-provisioning,[19] previous-application-identifiers,[20] and HotspotHelper.[21] The semi-private entitlements are listed in Table 6.2.

Five of the semi-private entitlements are vendor-specific, listing the app developer's names in the entitlement key. Flickr, Twitter, Vimeo, and several Facebook applications all have vendor-specific, semi-private entitlements with keys referencing default access and account data. Nike has three applications with a vendor-specific entitlement referencing healthkit and Nike Fuel, Nike's proprietary unit of measurement for fitness activity.

Two applications possess unique semi-private entitlements that seem to otherwise be used by system applications. As publicly announced by Strafach, the Uber application has the explicit-graphics-priority[22] entitlement which is used by jailbreak applications to build screen recording applications.[23] This correlation implies that Uber could have recorded the user's screen while the application ran in the background. Uber quickly removed the entitlement after its existence was made public, thus highlighting the importance of transparency for entitlements. Netflix has a similar

[18] com.apple.developer.passkit.pass-presentation-suppression
[19] com.apple.developer.payment-pass-provisioning
[20] https://developer.apple.com/library/content/technotes/tn2319/_index.html
[21] com.apple.developer.networking.HotspotHelper
[22] com.apple.private.allow-explicit-graphics-priority
[23] https://stackoverflow.com/questions/32239969/
iomobileframebuffergetlayerdefaultsurface-not-working-on-ios-9

Table 6.3 Per Method Entitlement Requirements Based on Error Messages

Entitlement Key Requirements Based on Error Message	Number of Methods
No Entitlement Required	123
Unspecified Entitlement Required	8
com.apple.managedconfiguration.profiled-access	1
com.apple.managedconfiguration.profiled.shutdown	1
com.apple.managedconfiguration.mdmd.push	2
com.apple.managedconfiguration.profiled.migration	2
com.apple.managedconfiguration.profiled.usercompliance	4
com.apple.managedconfiguration.profiled.get	5
com.apple.managedconfiguration.profiled.provisioningprofiles	5
com.apple.managedconfiguration.mdmd-access	7
com.apple.managedconfiguration.profiled.configurationprofiles	10
com.apple.private.mobileinstall.allowedSPI	18
com.apple.managedconfiguration.profiled.set	22

entitlement with allow-mpeg4streaming.[24] This entitlement is also possessed by system applications built into iOS, but the best of our knowledge the entitlement is undocumented. While Netflix is not the only video streaming application in our sample (e.g., Hulu and Amazon Prime Video), it is the only third party application in our sample with this entitlement.

6.6 Empirical Study

Figure 6.3 illustrates the number of invocations, unique methods, completion handlers, completion confirmations, and entitlement free methods dynamically tested with Kobold. For a third party application without entitlements, Kobold's static analysis phase extracted 237 sandbox accessible mach-ports and 2019 candidate remote methods to invoke. 1183 unique methods were tested with the mach-ports associated with the daemons each method was extracted from. Note that due to mach-port to protocol mapping ambiguity, many of those methods could be assigned to incorrect ports. 553 of the methods tested contained completion handlers, and 208 of those methods returned completion handler confirmations when invoked. As shown by Table 6.3, of the 208 remote methods with successful completion messages, 123 did not require entitlements, 8 required unspecified entitlements, and 77 required specific entitlements.

Completion Handlers: Completion handlers are blocks of code with arguments that can be passed to a remote method as one of the method's arguments. If the remote method completes, the code block assigned to the completion handler is executed. The completion handler's arguments (e.g., NSError or NSString values) will have been initialized with data from the daemon, and can be used in the scope of the completion handler's code block. Kobold uses this code block to output a completion confirmation that can be detected when inspecting the output of our application. This

[24] com.apple.coremedia.allow-mpeg4streaming

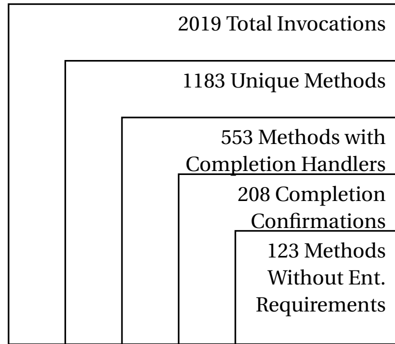

Figure 6.3 NSXPC Method Invocation Quantitative Results

Table 6.4 Entitlement Requirements for Port Access Through Sandbox

Entitlement Key	Entitlement Value	Mach Port	Entitlement Availability
com.apple.smoot. subscriptionservice	bool("true")	com.apple. VideoSubscriberAccount. videosubscriptionsd	Semi-Private
com.apple.smoot. subscriptionservice	bool("true")	com.apple.parsec. subscriptionservice	Semi-Private
com.apple.developer.siri	bool("true")	com.apple.siri. vocabularyupdates	Public

output makes it trivial to determine whether a remote method with a completion handler has run or not. However, as shown in Figure 6.3 approximately half of the unique methods tested did not have completion handlers and could not be labeled as accessible or inaccessible without further analysis.

Public and Semi-Private Entitlements: We identified three conditional sandbox rules allowing access to mach-ports based on public or semi-private entitlements. These results are represented by Table 6.4. One mach-port was accessible through a public entitlement, and two mach-port was accessible through a semi-private entitlement. Due to the low scale of this finding, we have not investigated those mach-ports or their remote methods. The methods with error messages that specified entitlement requirements only specified private entitlements (those not accessible to third party applications on the App Store). These results imply that public and semi-private entitlements do not play a significant role in access to NSXPC remote methods. However, as discussed in Section 6.8, more thorough analysis techniques could reveal entitlement checks missed by Kobold.

Number of Arguments: The number of arguments in a methods declaration plays a significant role in determining the difficulty of invoking a remote method successfuly and whether or not that method can be exploited. For example, a method with zero arguments is trivial to invoke correctly, but unlikely to be exploitable. At the other extreme, A method containing 10 arguments has a larger attack surface but it may be difficult to find valid values for all arguments. Table 6.5 shows the number of methods with various amounts of arguments (i.e., 0 to 10 arguments). Note

Table 6.5 Methods by Number of Arguments

Number of Arguments	Methods With That Number of Aguments
0	141
1	418
2	365
3	157
4	61
5	24
6	8
7	4
8	1
9	3
10	1

that a completion handler is treated as a single argument with respect to a remote method, but the completion handler may have its own arguments.

Types of Arguments: Table 6.6 lists the data types that appear most frequently in declarations of the methods invoked by Kobold. We categorize these data types into three groups, primitives, documented, and undocumented. Primitive types consist of those low level types that appear in the C programming language (e.g., int, long, double). Documented types (e.g., NSString) are abstractions constructed upon primitive types, and they are documented officially by Apple.[25] Undocumented types (e.g., AFSpeechRequestOptions) are abstraction built upon primitive types, but these data types are not officially documented by Apple. While primitive values can be fuzzed using random values, it is difficult to find acceptable values for more complex types. Apple's documentation may provide hints regarding initialization of documented types, but a thorough analysis of the values expected by NSXPC remote methods may require dynamic analysis, symbolic analysis, or extensive reverse engineering of the remote method.

Intra-Port Entitlement Consistency: Table 6.7 lists the number of methods with successful completion handlers with their respective mach-ports. The methods are also divided into two categories: 1) those that do not require entitlements; and 2) those that do require entitlements. Entitlement requirements are inferred from error messages provided in completion handlers. Mach-ports with methods in both categories are considered to have inconsistent entitlement policies, and have been highlighted in Table 6.7. As demonstrated with MDM functionality in Section 6.7, inconsistent entitlement policies may represent access control flaws where security sensitive methods are accidentally made available to unprivileged clients. While 205 ports where found to be accessible to third party applications through the sandbox, Kobold identifies 13 mach-ports hosting remote methods with successful completion handlers. This discrepancy may be due to mach-ports using

[25]https://developer.apple.com/documentation/foundation?language=objcoverview

interfaces other than NSXPC, entitlement requirements upon port connection, methods that do not use completion handlers, and Kobold's use of uninitialized variable values.

6.7 Findings

Kobold led to the discovery of two types of security relevant findings, confused deputy attacks and daemon crashes. These finding affect iOS 11.4.1 (the latest version at the time of writing), and we plan to disclose them to Apple.

6.7.1 Confused Deputy Attacks

Table 6.8 lists the confused deputy attacks detected by Kobold. These attacks can be grouped into three categories: 1) File Provider information leaks; 2) Microphone activation; 3) Unprotected Mobile Device Management (MDM) services. These attacks have been confirmed with proof of concept applications on a non-jailbroken 6th Generation iPod Touch running iOS 11.4.1 (the latest iOS version at the time of writing).

File Provider State Dump: The File Provider daemon provides an accessible method that replies with state information for the applications with File Provider functionality running on the device (e.g., Google Drive, Microsoft OneDrive). This leaked state information can be abused by a third party application in three ways. First, the leaked information reveals the names of other third party apps that have been installed if those applications use File Provider functionality. Second, the leaked information reveals UUID used in app directory names, which do not change upon rebooting the device. Therefore, these leaked UUID's could be used for device fingerprinting. Finally, the third party app can infer the names of files in File Provider directories. There is a simple side channel in iOS that allows an process to determine whether a file exists or not by attempting to read the file's metadata. If the file exists, a permission denied error may occur, or the metadata may be read. If the file does not exist, an error will specify that the file does not exist. Since the attacker must correctly guess the file path, this side channel is defeated by UUIDs in file paths. However, the File Provider data leak reveals those UUIDs to third party applications allowing them to begin inferring the names of files in File Provider directories. This inference could be accellerated through the use of a dictionary of interesting file names to check for.

Activate Voice Dictation: By invoking methods that start voice dictation sessions, a third party application can briefly activate the microphone without user permission (i.e., the user has not enabled microphone access in their Privacy Settings). This method causes a bell to ring, signalling that the microphone has been activated. Using uninitialized variables, the application does not gain access to the audio recording, and the microphone is only activated for about 1 second.

Inconsistent MDM Access Control Policy: When reviewing the entitlement requirements detected

by Kobold, we observed that the MDM management service had inconsistent entitlement requirements. The “com.apple.managedconfiguration.profiled.public” mach-port provides 71 methods Kobold detects as accessible. Of these 71 methods, the majority require MDM related entitlements, but 21 of the methods have no apparent entitlement requirements. A manual investigation of the MDM methods that did not require entitlements led us to three MDM services that allow a third party application to disable system functionality. These MDM services are effective even if the victim’s device that has not been configured to run under MDM control. First, the text replacement or keyboard shortcuts functionality can be disabled and the menu to configure new shortcuts is disabled in Settings application. Second, the dictation option for voice to text functionality can be disabled and the toggle to enable the feature is removed from the Settings menu. Finally, access to all website on all mobile browsers can be disabled, and users attempting to access websites are challenged to enter an unknown pin code with a recorded number of failed attempts.

6.7.2 Daemon Crashes

Kobold detected crashes on a jailbroken iPhone 5s running iOS 11.1.2 and a stock 6th Generation iPod Touch running iOS 11.4.1 (the latest iOS version at the time of writing). The crashes detected are listed in Table 6.9 which lists ten executables with a total of 14 unique crashes based on stack trace analysis. The locationd and wcd[26] crashes could not be triggered on the stock iPod, but we speculate that this is due to hardware differences since the iPod does not have a GPS sensor and does not support Apple Watch connectivity. Three of the crashed daemons run with root authority. If attackers are able to exploit the causes of these crashes, the root authority daemons would be valuable targets.

Crash Types: We categorize three types of crash based on the way the process was terminated: 1) abort signal; 2) segmentation fault; 3) killed by watchdog. First, seven crashes (three daemons) terminate when the daemon sends an abort signal. This signal could be the result of asserting that a value is null and aborting the process in response. Second, six crashes (six daemons) terminate when daemons attempt to access invalid memory addresses at or near the zero address. We speculate that these unusual memory accesses are caused by Kobold’s default use of uninitialized variables as remote method arguments. Since segmentation faults imply that the daemon is attempting to use corrupted values, we consider segmentation fault crashes more significant than abort signal crashes. All of our root authority daemon crash are due to segmentation faults. Third, the Preferences[27] crash is unique in that the process freezes and is killed by a watchdog process after 10 seconds of inactivity.

Quantifying Crashes: We quantify crashes in two ways, number of daemons and number of unique crash stack traces. The stack traces are included in the crash reports generated by iOS. We developed

[26] a daemon related to the Apple Watch
[27] Also known as Settings.

a script to extract stack traces from these crash reports and compare them to determine how many unique stack traces were present for each daemon. For example, the accessoryd daemon seems to crash for every method we called on the com.apple.iap2d.xpc port. However, a stack trace analysis revealed that each method invocation for the port was triggering the same stack trace, which implies that the same issue is causing the crash despite invoking different methods. The wcd and replayd daemons do generate unique stack traces when crashed by different method invocations. These stack traces imply that multiple bugs exist in wcd and replayd, but a single bug may be causing the crashes for accessoryd.

Crash Causes: For those crashes that are consistently repeatable, we isolate the remote method causing the crash. Methods that trigger crashes were detected using a script on a jailbroken device that killed our method invocation application when a crash report was added to the system log. Then, we manually tested the methods immediately prior to the code line where our app stopped executing. Each of these methods was found to cause crashes in the receiving daemon if called with uninitialized argument values. Note that the willSwitchUser method does not have any arguments, but it still causes the iTunes Store daemon to crash with a segmentation fault if it is invoked by a third party application. This iTunes Store daemon crash represents a bug, but without a field for attacker input, it is unlikely to be exploitable. Five of the crashes were observed to occur when running our method invocation app, but they did not repeat consistently enough for us to assign a specific method to the crash. These crashes are labeled in Table 6.9 as nondeterministic.

Inconsistent Entitlement Enforcement: When investigating the methods crashing replayd, we noticed an inconsistency in one method's entitlement enforcement. A remote method called startRecording[28] is provided by replayd and returns an error message specifying a required entitlement, if the remote method is the only one invoked by our application. However, if our application invokes the other remote methods tested by Kobold before invoking the replayd method, the entitlement requirement error is not returned. Instead, the method triggers a prompt asking the user for permission to record the screen. If the user accepts the prompt, replayd will crash (the crash is likely due to Kobold's use of uninitialized variable values). This finding suggests that state-based conditions (e.g., the set of methods previously invoked) can lead to entitlement enforcement failures.

6.8 Limitations

Kobold has two types of limitations: 1) limitations inherent to working with closed source systems; 2) limitations that could be overcome with additional engineering effort.

Several limitations of Kobold are inherent to the closed source nature of iOS. Since we do not have ground truth for the set of third party accessible NSXPC remote methods we cannot quantify

[28] Method name has been simplified. The full name appears in Table 6.9

the number of methods that Kobold may have failed to detect. While the confused deputy attacks we present in this paper are clear security concerns, we do not have an access control policy specification to compare our findings to. A jailbreak provides logging tools that make dynamic analysis of IPC functionality significantly easier, but not all versions of iOS have been jailbroken, and there is no guarantee of future jailbreaks. While Kobold's methodology can be complemented with a jailbroken device, it can also be applied to a stock device. A jailbroken device provides additional feedback, but IPC reply message analysis, crash logs, and user perceivable system damage can all be detected on a stock device.

Other limitations could be overcome with additional engineering effort that would allow researchers to more accurately enumerate accessible remote methods or gain a deeper understanding of the security consequences of each remote method. While Kobold can statically extract the data types of remote argument methods, it does not automatically determine which values those arguments should be initialized with. Simple variables can be easily assigned various values (e.g., integers or strings), but correctly initializing complex, undocumented class types requires significant reverse engineering or dynamic analysis of the method being invoked during normal runtime operations. Kobold does not detect NSXPC remote methods provided by shared libraries, and it does not detect the other interfaces for remote methods other than NSXPC (e.g., XPC or Mach Interface Generator). A small number (less than ten) of problematic method invocations were intentionally removed from analysis due to Xcode errors preventing compilation.

Kobold relies on error message semantics to infer decentralized entitlement requirements for remote methods, but more sophisticated analysis methods such as symbolic execution or back-tracing may detect additional entitlement requirements missed by Kobold. The entitlement survey performed by Kobold does not include paid applications and only analyzes a sample of the free iOS applications available at one time. A longitudinal study of significantly more applications including paid applications may reveal new semi-private entitlements as well as trends in their distribution over time.

Kobold uses completion handler messages to determine which remote methods were invoked successfully. However, a significant number of the remote methods Kobold extracts do not have completion handlers. Therefore, other forms of confirming remote method invocation such as automated setting of debugger breakpoints in daemon code could reveal more third party accessible methods.

6.9 Related Work

Kobold is related to work in four fields: 1) iOS access control policy analysis; 2) iOS IPC fuzzing; 3) iOS exploitation; 4) iOS application analysis.

Kobold uses tools produced by two prior works in order to identify mach-ports that are accessible

to third party applications through the sandbox. SandScout [Des16] reverse engineers and models iOS sandbox policies, but it does not model the semantics of entitlements requirements. However, iOracle [Des18] builds upon SandScout in several ways including the modeling of Unix Permissions and capabilities such as entitlements and sandbox extensions. Therefore, iOracle allows Kobold to input a set of third party accessible entitlements and automatically determine which mach-ports are accessible through the sandbox for an application with those entitlements.

To the best of our knowledge, Kobold is the first systematic exploration of NSXPC remote methods, but there have been several prior works that fuzz Apple's IPC mechanisms. Han et al. [HC17] fuzzed Apple driver interfaces by dynamically observing system behaviors to infer dependencies between API calls (i.e., calling functions in a certain order or using the return value of a function as an argument to another function). The Pangu team [Wan] presented their approach to fuzzing XPC services in order to exploit data dereference operations. Beer [Bee] fuzzed XPC services in order to identify opportunities for type confusion attacks. Kydyraliev [Kyd] explored Mach Interface Generator (MIG) services by observing messages sent at runtime and replaying those messages with mutations in order to trigger crashes. Kobold differs from these prior works in two ways. First, it seeks confused deputy attacks which must consider both the methods functionality and accessibility. Second it uses static analysis to determine the mach-ports, names, and argument types of remote methods instead of dynamic analysis which may miss methods that were not invoked at runtime.

Kobold assumes that the confused deputy attacks and crashes we discovered can be deployed by a third party app able to pass Apple's app vetting process and infiltrate the app store. This assumption is based on three prior works on modifying iOS app behavior after passing the vetting process. Wang et al.[Wan13] used return oriented programming (ROP) to modify their program's control flow after it had passed the app vetting process and been published to the App Store. XiOS [Buc15] improved upon the work of Wang et al. by reducing the attack's complexity and proposing an attack mitigation in the form of an in-line reference monitor. Finally, Han et al. [Han13b] used obfuscation techniques conceptually similar to Java reflection in order to bypass app vetting and invoke private API calls after publishing to the app store.

The entitlement extraction component of Kobold is a form of app analysis, and there have been several other works in this area. PiOS [Ege11] uses backtracing to determine register values when an Objective-C dispatch call is made. This backtracing allows static analysis to infer which function will be executed in response to the dispatch, which helps detect when applications are invocing private API calls. iRiS [Den15] expands upon PiOS by adding a dynamic analysis component using forced execution to resolve dispatches that were difficult to infer through static analysis. Kobold uses a similar approach as CRiOS [Ori16] for app scraping, and CRiOS analyzes third party applications for network security issues.

6.10 Conclusion

In conclusion, Kobold allowed us to reveal and investigate the relatively unexplored attack surface of NSXPC remote methods available to third party applications. In order to model the capabilities of third party applications, Kobold automatically extracted entitlements from popular third party applications on the App Store and discovered several semi-private entitlements normally unavailable to developers. Invoking the methods we discovered with Kobold revealed several previously unknown access control flaws as well as multiple daemon crashes.

6.11 Acknowledgements

We thank our lab intern, David Wu, for his technical assistance. David helped in isolating causes of daemon crashes and searched for third party apps with Mobile Device Management entitlements.

This work was supported in part by the Army Research Office (ARO) grants W911NF-16-1-0299, the National Science Foundation (NSF) CAREER grant CNS-1253346. This work has been funded by University Politehnica of Bucharest, through the "Excellence Research Grants" Program, UPB - GEX 2017. Identifier: UPB - GEX2017, Ctr. No. 19/2017. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Table 6.6 Common Data Types in Extracted NSXPC Methods

Data Type	Occurences in Method Declarations
void	1536
NSString *	427
NSError *	381
_Bool	296
oneway void	214
NSDictionary *	161
NSArray *	122
NSData *	87
NSUUID *	84
long long	62
NSURL *	62
unsigned long long	55
int	36
double	31
IDSService *	18
NSNumber *	17
NSFileManager *	16
NSURLSession *	15
unsigned int	14
IDSAccount *	11
CSSpeechController *	11
id	10
NSURLRequest *	10
NSDate *	9
AFSpeechRequestOptions *	9
unsigned char	8
APSConnection *	8
NSXPCListenerEndpoint *	7
NSURLSessionTask *	7
NDApplication *	7
MCProfileConnection *	7
AFAudioPlaybackRequest *	7
NSFileHandle *	6
IDSMessageContext *	6
GKGameSession *	6
GKCloudPlayer *	6
CBPeer *	6
CBCentralManager *	6
SiriCoreLocalSpeechRecognizer *	5
NSURLSessionDownloadTask *	5

Table 6.7 Methods per Mach Port. Inconsistent Entitlement Requirements Highlighted.

Mach Port	Without Entitlement Requirements	With Entitlement Requirements
com.apple.sharingd.nsxpc	1	0
com.apple.assistant.dictation	3	0
com.apple.nsurlstorage-cache	3	0
com.apple.replayd	3	0
com.apple.nsurlsessiond	4	0
com.apple.wcd	4	0
com.apple.coreservices. lsuseractivitymanager.xpc	8	0
com.apple.managedconfiguration. mdmdservice	0	9
com.apple.voiceservices.tts	9	0
com.apple.mobile.installd	1	18
com.apple.FileProvider	28	0
com.apple.nano.nanoregistry. paireddeviceregistry	38	8
com.apple.managedconfiguration. profiled.public	21	50

Table 6.8 Confused Deputy Attacks

Effect	Method	Mach-Port
Leak names of installed apps with File Providers	dumpStateTo:completionHandler:	com.apple.FileProvider
Device fingerprinting	dumpStateTo:completionHandler:	com.apple.FileProvider
Infer file names in File Providers	dumpStateTo:completionHandler:	com.apple.FileProvider
Activate microphone	startRecordingFor PendingDictationWithLanguageCode: options:speechOptions:reply:	com.apple.assistant.dictation
Disable Text Replacement	setKeyboardShortcutsAllowed: completion:	com.apple.managedconfiguration .profiled.public
Disable Dictation	setDictationAllowed:completion:	com.apple.managedconfiguration .profiled.public
Block access to all websites	addBookmark:completion:	com.apple.managedconfiguration .profiled.public

Table 6.9 Daemon Crashes

Executable	Mach-Port	Method	UID	Crash Type
replayd	com.apple.replayd	setupBroadcastWithHostBundleID:broadcastExtensionBundleID: broadcastConfigurationData:userInfo:handler:	mobile	Abort
replayd	com.apple.replayd	startRecordingWindowLayerContextIDs:windowSize: microphoneEnabled:cameraEnabled:broadcast:systemRecording: captureEnabled:listenerEndpoint:withHandler:	mobile	Abort
sharingd	com.apple.sharingd.nsxpc	createCompanionServiceManagerWithIdentifier:clientProxy:reply:	mobile	Abort
wcd	com.apple.wcd	acknowledgeUserInfoResultIndexWithIdentifier:clientPairingID:	mobile	Abort
wcd	com.apple.wcd	acknowledgeUserInfoIndexWithIdentifier:clientPairingID:	mobile	Abort
wcd	com.apple.wcd	acknowledgeFileResultIndexWithIdentifier:clientPairingID:	mobile	Abort
wcd	com.apple.wcd	acknowledgeFileIndexWithIdentifier:clientPairingID:	mobile	Abort
accessoryd	com.apple.iap2d.xpc	stopBLEUpdates:blePairingUUID:	mobile	Segfault
itunesstored	com.apple.itunesstored.xpc	willSwitchUser	mobile	Segfault
aggregated	NONDETERMINISTIC	NONDETERMINISTIC	mobile	Segfault
Preferences	NONDETERMINISTIC	NONDETERMINISTIC	mobile	Killed by watchdog
UserEventAgent	NONDETERMINISTIC	NONDETERMINISTIC	root	Segfault
locationd	NONDETERMINISTIC	NONDETERMINISTIC	root	Segfault
powerlogHelper	NONDETERMINISTIC	NONDETERMINISTIC	root	Segfault

CHAPTER

7

DIRECTIONS FOR IOS ACCESS CONTROL ANALYSIS

This chapter seeks to predict the directions of iOS access control analysis. These predictions are based on this dissertation's findings, trends in iOS access control, and areas amenable to future research.

7.1 Dissertation Summary

Smartphones have become an integral part of modern life, and iOS is running on a significant number of those devices. Users trust their devices with financial, medical, and personal data, and they require their devices to function reliably for work, emergencies, and other daily needs. Access control can mitigate damage caused by malicious or misconfigured software, but designing correct access control policies for complex systems is difficult. Therefore, the research questions introduced in Chapter 1, guided this dissertation to identify flaws in iOS access control policies. This dissertation demonstrated that the access control mechanisms in iOS are inter-dependent and their analysis should consider the composite protection system as a whole.

Chapter 2 provided background knowledge necessary to understand the contributions made by this dissertation. In addition to introducing iOS terminology, Chapter 2 categorized iOS access control mechanisms based on temporal and functional attributes. Chapter 2 also explained the

internal mechanisms of various iOS access control mechanisms as well as enumerating strengths and weaknesses of their designs. Finally, Chapter 2 provides examples of inter-dependence among iOS access control mechanisms (e.g., entitlements that bypass Privacy Settings or entitlement conditioned sandbox rules).

In Chapter 3, we introduced prior work on access control design, introducing terms that we used to describe access control concepts referenced in later chapters. Chapter 3 also highlighted research on access control policy evaluation that inspired the techniques we applied in our methodologies and evaluations. Finally, Chapter 3 discussed the state of iOS security research including works on code analysis, exploitation techniques, and in-line reference monitors.

Chapter 4 presented SandScout, the first systematic study of the iOS container sandbox profile, which confines third-party applications. While SandScout analyzed the sandbox access control mechanism in isolation, it was a necessary first step in modeling the composite protection system. SandScout automatically extracted and reverse engineered sandbox profiles from iOS firmware images and modeled those sandbox rules as queriable Prolog facts. By modeling these profiles, we were able to define security requirements (e.g., no write access to system files) and automatically detect violations of those requirements in policy logic. The security violations we detected resulted in six CVE acknowledgements from Apple. These CVEs represent security patches made to the iOS code base in response to our findings.

Chapter 5, stepped toward building a model of the composite protection system by modeling the relationship between the Apple Sandbox and Unix File Permissions. This composite model was able to answer queries such as "Which low integrity processes have write access to the Media/ directory?". Such queries allowed us to represent security requirements and detect flawed access control policies such as applications that can grant unrestricted privileges to themselves. iOracle performed two types of evaluation, triage of executables exploited by known jailbreaks and discovery of previously unknown policy flaws. For jailbreak triage, iOracle significantly reduced the search space of executables for security analysts to consider while including those executable exploited in known jailbreak attacks. For the discovery of previously unknown policy flaws, iOracle identified several flaws allowing significant privilege escalation with consequences similar to sandbox escapes.

Chapter 6 expanded our knowledge of the composite protection system in the domain of Inter-Process Communication (IPC) by presenting Kobold. Kobold is a framework that allowed us to investigate the relationship between the sandbox and decentralized entitlement checks with respect to IPC functions with NSXPC interfaces. In order to determine the set of entitlements available to third party applications, Kobold performed a survey of entitlement distribution among applications from the App Store, discovering several semi-private entitlements that are normally unavailable to developers. By enumerating the remote methods accessible to third party applications, Kobold provided insight into a relatively unexplored attack surface. Invoking the methods discovered by Kobold revealed several previously unknown access control flaws as well as multiple daemon crashes.

SandScout, iOracle, and Kobold provide significant strides toward a model of the composite protection system of iOS. However, many access control mechanisms remain to be modeled (e.g., Posix ACLs, privacy setting databases, plist configurations). We hope that by publishing our papers and open sourcing our frameworks, we can inspire more iOS access control research and mitigate barriers to entry that have kept many researchers away from this field. Our frameworks are designed with expansion in mind, but researchers skilled and courageous enough to reverse engineer iOS must extract more data about policies and their semantics.

7.2 Trends

There are several iOS trends worth noting that may influence future research in this field: iOS can be expected to run on more devices; machine learning will elevate the security sensitivity of trivial data sources; a dearth of new features could lead vendors to use comparable security as a selling point; virtualization efforts may make iOS app analysis more effective; poor maintenance of research tools and harsh novelty requirements could suppress academic iOS research; and jailbreaks are transitioning to IPC based exploits as sandbox policies mature into a hardened state.

In the near future, the access control mechanisms in iOS may be deployed to several other platforms, each with their own security requirements and user expectations. Several of CVEs we earned with SandScout also applied to Apple's watchOS and tvOS. The same security patch could be used because all three platforms have similar access control systems. A proposed cross-platform development framework called Marzipan will allow iOS applications to run on macOS. In the future, I expect fully functional iOS variants to appear on more household devices (e.g., refrigerators, mirrors, bicycles, etc.). Each of these platforms could run similar versions of iOS while having unique access control requirements (e.g., a refrigerator with odor sensors or automated purchasing capabilities). Finally, Apple's iOS based IoT hub may introduce access control features for managing IoT devices.

Machine learning will provide both the means and motivation to exploit side channels and invade user privacy. As machine learning algorithms prove to be practical effective, the data to train them will become more valuable. Machine learning has already proven capable at using relatively trivial sources of side channel information on iOS to infer sensitive user actions such as app launches and location searches [Zha18]. Sensors that are now considered to have trivial security sensitivity (barometer, brightness sensor, thermometer) might someday be exploited with machine learning to infer user location, behaviors or other private information. As users purchase and use more devices or multi-user capabilities are added to mobile devices, cross device fingerprinting and multi-tenant side channels will become more desirable to attackers.

As access control policies become more complex and vendors seek to compare the security of their systems to those of competitors, automated policy analysis may increase in popularity. User perceivable smartphone features (e.g., screen resolution, cameras) may begin to stabilize

across device releases, and vendors may use formal security comparisons as selling points. Just as VulSAN [Che09] was able to compare the security quality of different access control policies for Linux, Android and iOS security engineers could develop methods and metrics for comparing the security qualities of modern operating systems.

A company named Corellium[1] claims to have successfully virtualized a modern iOS device. If Corellium makes the software accessible to researchers, this breakthrough would eliminate many barriers to entry in the field as well as allowing significant parellellization for dynamic analysis. To the best of our knowledge symbolic and concolic analysis are not currently available for iOS binaries, but they may be developed as future work, making app analysis significantly more effective. Finally, iOS devices are expected to continue being built with more powerful hardware with each device generation. Mobile devices hardware in the future may allow for on-device training for machine learning or brute force attacks that currently require server scale resources.

The nature of jailbreaking is also changing. Recent jailbreaks have exploited IPC vulnerabilities instead of file system vulnerabilities. As iOS file system access control mechanisms harden, security researchers may shift their focus to the more complex and less explored area of IPC. Recent jailbreaks (i.e. LiberiOS[2]) have not received Cydia[3] support (the jailbreaker's app store) and have instead served as research tools rather than allowing new user customizations. There are also efforts to automate and abstract the steps repeated in jailbreaking various iOS versions, such as a jailbreaking toolkit called QiLin[4].

Poor documentation, system changes from Apple, and limited sharing of research tools often leads to iOS researchers reimplementing prior work. In addition to this duplicated effort, academic reviewers demand novelty and are likely to reject efforts focused on updating obsolete, but essential tools. For example, the use of backtracing is an essential static analysis technique to build control flow graphs for iOS binaries, and PiOS addressed this issue, but it was designed for much older architectures and iOS versions. iRiS [Den15] is a framework that improved upon the analysis of iOS applications but first had to reimplement PiOS and then make significant contributions beyond PiOS in order to publish. A part of the iRiS code has been open sourced[5], but it has not been maintained. Researchers hoping to expand upon state of the art iOS app analysis will now need to reimplement PiOS and iRiS before making their own contributions. Therefore, iOS researchers are left in a Sisyphean cycle reimplementing more and more complex tools in order to progress beyond prior work. In order for this field of research to continue and thrive, new researchers must stand on the shoulders of giants, instead of drowning in their wake.

[1]https://corellium.com/
[2]http://newosxbook.com/liberios/
[3]https://en.wikipedia.org/wiki/Cydia
[4]http://newosxbook.com/QiLin/
[5]https://github.com/tyrael9/valgrind-ios

7.3 Future Work

This section suggests areas for future work based on emergent properties in the field and the trends discussed previously.

Automated policy analysis and machine learning may allow for automated planning of exploits. The iOracle framework already represent a limited model of the iOS protection system in Prolog, which is amenable to automated planning. Protection state operations would need to implemented in Prolog as actions that transition the current state to a new one. For example, if a confused deputy can be made to chown a file, then a transition can be made to a new state in which the attacker owns the chowned file. Given a protection system's original state, conditions and consequences for actions, and a goal state, Prolog should be able to automatically construct complex chains of exploits such as jailbreaks.

As researchers gain a better understanding of the internals of iOS access control, proof of concept improvements upon Apple's designs can be proposed. Kobold provided significant insight into the functionality of mach-ports, discovering several abusable IPC services that do not seem necessary for all third party applications. Therefore, more granular, app-specific sandbox profiles could improve upon the generic container sandbox profile. For example, one could determine which mach-ports each third-party application requires at build time or app vetting time and design a custom sandbox to prevent access to other ports. iOracle and SandScout demonstrated that sandbox extensions are impractical to revoke. Perhaps a more revocable dynamic capability could be proposed as a replacement for sandbox extensions.

A defense in depth approach to policy design could prevent an attacker from gaining significant privileges upon defeating one of several access control mechanisms. iOracle discussed how jailbreaks could progress from one level of privilege to the next by exploiting new privileges gained at each step of exploitation. To mitigate these increased privileges, the sandbox, Unix Permissions, and decentralized checks could be designed such that a subject does not gain additional privileges even if it escapes the sandbox or gains ownership of a privileged file. Such policies would be redundant, but more robust against partial failures. These redundant policies could also be designed and tested using automated policy analysis tools.

Kobold focused on NSXPC interface type IPC services, and it does not perform significant reverse engineering for each of the methods or semi-private entitlements it enumerates. Therefore, Kobold can be expanded upon in three ways. First, researchers can enumerate and evaluate other IPC interface types such as MIG and XPC. Second, tools can be constructed to perform a deeper analysis of each remote method in order to gain a better understanding of its functionality and expected input values. Third, a systematic analysis of decentralized entitlement checks related to semi-private entitlements would provide insight into the privileges they provide to third party applications.

7.4 Concluding Remarks

From the start of my research in iOS, I knew it would be difficult. In fact, the challenge and the skills I would need to earn were among the reasons I agreed to take on the topic. I saw iOS reverse engineers as some of the most brilliant security researchers of my generation, and I still do. I felt that if I could train myself and break into this field, I would discover vast unexplored attack surfaces with little danger of being scooped by other researchers. However, our first three submissions were rejected, our first two papers had to be abandoned, and our first two years of work did not yield significant results. We ruined not one, but two research devices and accidentally leaked research code to a public git repository. Despite these hardships and failures, we persisted, our team grew, and our efforts paid off. We learned from our failures, modified our approaches, reused our existing data and tools to find our niche.

Systems research is not for the faint of heart. Each experiment, requires significant effort to reach the frontier set by prior work and no guarantee that new findings will be waiting for you as you push forward. However, my advice is to have faith and push beyond those frontiers anyway. Better yet, follow emerging trends and seek accessible yet unexplored new areas. Modern OSs are extremely complex system, and if you can search where no one else has, you should find something interesting. Be patient, give every challenge a strong fight, and do not underestimate the value of playing and following your intuition.

Despite the many flaws we discovered, I have great respect for the security engineers that designed Apple's access control mechanisms and policies. Pierre, Christina, Scott and the rest of the Apple Security team are doing an excellent job, and I look forward to meeting them again soon. Now I am off to start my career in industry working for Samsung, but I have no plans to give up my iPhone any time soon.

BIBLIOGRAPHY

[Ace] *AceDeceiver: First iOS Trojan Exploiting Apple DRM Design Flaws to Infect Any iOS Device*. http://researchcenter.paloaltonetworks.com/2016/03/acedeceiver-first-ios-trojan-exploiting-apple-drm-design-flaws-to-infect-any-ios-device/. Accessed: 2016-05-05.

[Ala08] Alam, M. et al. “Usage control platformization via trustworthy SELinux”. *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM. 2008, pp. 245–248.

[Appa] *App Store kicks off 2018 with record-breaking holiday season*. https://www.apple.com/newsroom/2018/01/app-store-kicks-off-2018-with-record-breaking-holiday-season/. Accessed: 2018-07-24.

[Appb] *App Store shatters records on New Year’s Day*. https://www.apple.com/newsroom/2017/01/app-store-shatters-records-on-new-years-day/. Accessed: 2018-07-24.

[Bee] Beer, I. *Auditing and Exploiting Apple IPC*. https://thecyberwire.com/events/docs/IanBeer_JSS_Slides.pdf. Accessed: 2018-07-24.

[BL73] Bell, D. E. & LaPadula, L. J. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.

[Bib77] Biba, K. J. *Integrity considerations for secure computer systems*. Tech. rep. MITRE CORP BEDFORD MA, 1977.

[Bla11] Blazakis, D. “The apple sandbox”. *Arlington, VA, January* (2011).

[Buc15] Bucicoiu, M. et al. “XiOS: Extended Application Sandboxing on iOS”. *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2015.

[Byf15] Byford, S. *Apple removes malware-infected App Store apps after major security breach*. The Verge. http://www.theverge.com/2015/9/20/9362585/xcodeghost-malware-app-store-security. 15.

[Cha08] Chaudhuri, A. et al. “EON: Modeling and Analyzing Dynamic Access Control Systems with Logic Programs”. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2008.

[Che09] Chen, H. et al. “Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems”. *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2009.

[Che16] Chen, K. et al. "Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS". *Proceedings of the IEEE Symposium on Security and Privacy*. 2016.

[DZ11] Dai Zovi, D. A. "Apple iOS 4 security evaluation". *Black Hat USA* (2011).

[Dav12] Davi, L. et al. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones". *Proceedings of the Network and Distributed Systems Symposium (NDSS)*. 2012.

[Dea16] Deaconescu, R. et al. *SandBlaster: Reversing the Apple Sandbox*. arXiv: 1608.04303. 2016.

[Den15] Deng, Z. et al. "iRiS: Vetting Private API Abuse in iOS Applications". *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2015.

[Des16] Deshotels, L. et al. "SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles". *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2016.

[Des18] Deshotels, L. et al. "iOracle: Automated Evaluation of Access Control Policies in iOS". *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM. 2018, pp. 117–131.

[Ios] *Download*. https://developer.apple.com//ios/download/. Accessed: 2016-04-20.

[Dro] *Dropbox responds to accusations its Mac desktop client hacks OS X security*. https://techcrunch.com/2016/09/09/dropbox-responds-to-accusations-its-mac-desktop-client-hacks-os-x-security/. Accessed: 2016-09-25.

[Dsc] *dsc_extractor.cpp*. https://opensource.apple.com/source/dyld/dyld-195.6/launch-cache/dsc_extractor.cpp. Accessed: 2016-05-19.

[Ege11] Egele, M. et al. "PiOS: Detecting Privacy Leaks in iOS Applications." *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2011.

[Enc08] Enck, W. et al. *Mitigating Android Software Misuse Before It Happens*. Tech. rep. NAS-TR-0094-2008. Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA: Network and Security Research Center, 2008.

[Essa] Esser, S. *Antid0te 2.0 - ASLR in iOS*. http://conference.hackinthebox.org/hitbsecconf2011ams/materials/D1T1%20-%20Stefan%20Esser%20-%20Antid0te%202.0%20-%20ASLR%20in%20iOS.pdf. Accessed: 2016-02-15.

[Essb] Esser, S. *iOS 8 Containers, Sandboxes and Entitlements*. http://www.slideshare.net/i0n1c/ruxcon-2014-stefan-esser-ios8-containers-sandboxes-and-entitlements. Accessed: 2015-11-6.

[Fer] Ferber, I. *Data You Can Trust*. https://developer.apple.com/videos/play/wwdc2018/222. Accessed: 2018-07-24.

[fG!] fG! *Apple's Sandbox Guide v 1.0*. http://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf. Accessed: 2015-02-04.

[Key] *Firmware Keys*. https://www.theiphonewiki.com/wiki/Firmware_Keys. Accessed: 2016-04-19.

[Gas17] Gasparis, I. et al. "Detecting Android Root Exploits by Learning from Root Providers". *Proceedings of the USENIX Security Symposium*. 2017.

[Gra12] Grace, M. et al. "Systematic Detection of Capability Leaks in Stock Android Smartphones." *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2012.

[HC17] Han, H. & Cha, S. K. "IMF: Inferred Model-based Fuzzer". *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2345–2358.

[Han13a] Han, J. et al. "Comparing Mobile Privacy Protection Through Cross-Platform Applications". *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2013.

[Han13b] Han, J. et al. "Launching Generic Attacks on iOS with Approved Third-Party Applications". *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*. 2013.

[Han13c] Han, J. et al. "Launching generic attacks on iOS with approved third-party applications". *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*. 2013.

[Har76] Harrison, M. A. et al. "Protection in operating systems". *Communications of the ACM* **19**.8 (1976), pp. 461–471.

[Hic10a] Hicks, B. et al. "A Logical Specification and Analysis for SELinux MLS Policy". *ACM Transactions on Information and System Security (TISSEC)* **13**.3 (2010), p. 26.

[Hic10b] Hicks, B. et al. "A Logical Specification and Analysis for SELinux MLS Policy". *ACM Transaction on Information and System Security* **13**.3 (2010).

[Ioz] Iozzo, V. *A Sandbox Odyssey*. https://prezi.com/lxljhvzem6js/a-sandbox-odyssey-infiltrate-2012/. Accessed: 2015-11-7.

[Appc] *iTunes Preview*. https://itunes.apple.com/us/genre/ios/id36?mt=8. Accessed: 2016-05-04.

[Jae08] Jaeger, T. “Operating system security”. *Synthesis Lectures on Information Security, Privacy and Trust* **1**.1 (2008), pp. 1–218.

[Jae02] Jaeger, T. et al. “Managing access control policies using access control spaces”. *Proceedings of the seventh ACM symposium on Access control models and technologies*. ACM. 2002, pp. 3–12.

[Jae03] Jaeger, T. et al. “Analyzing Integrity Protection in the SELinux Example Policy”. *Proceedings of the USENIX Security Symposium*. 2003.

[Jok] *Joker*. http://newosxbook.com/tools/joker.html. Accessed: 2016-05-19.

[Kur16] Kurtz, A. et al. “Fingerprinting Mobile Devices Using Personalized Configurations”. *Proceedings on Privacy Enhancing Technologies (PoPETS)* 1 (2016).

[Kyd] Kydyraliev, M. *Mining Mach Services within OS X Sandbox*. http://2013.zeronights.org/includes/docs/Meder_Kydyraliev_-_Mining_Mach_Services_within_OS_X_Sandbox.pdf. Accessed: 2015-11-6.

[Lam74] Lampson, B. W. “Protection”. *ACM SIGOPS Operating Systems Review* **8**.1 (1974), pp. 18–24.

[Dmg] *Lekensteyn/dmg2img*. https://github.com/Lekensteyn/dmg2img. Accessed: 2016-05-19.

[Lev] Levin, J. *A (long) evening with Mobile_Obliterator and a look into iOS entitlements*. http://newosxbook.com/articles/EveningWithMobileObliterator.html. Accessed: 2015-11-9.

[Lev16] Levin, J. *MacOS and iOS Internals, Volume III: Security & Insecurity*. Technologeeks Press, 2016.

[Loo] Lookout. *Technical Analysis of Pegasus Spyware*. Accessed: 2017-08-03.

[Lzs] *lzssdec.cpp*. http://nah6.com/~itsme/cvs-xdadevtools/iphone/tools/lzssdec.cpp. Accessed: 2016-05-19.

[Mil12] Miller, C. et al. *iOS Hacker’s Handbook*. John Wiley & Sons, 2012.

[Sna] *Multiple iOS apps found to be harvesting Snapchat user credentials*. http://9to5mac.com/2016/03/08/ios-apps-snapchat-harvest-credentials/. Accessed: 2016-05-05.

[NK11] Naldurg, P. & KR, R. “SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models”. *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 2011.

[Ori16] Orikogbo, D. et al. “CRiOS: Toward Large-Scale iOS Application Analysis”. *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM. 2016, pp. 33–42.

[Ou05] Ou, X. et al. “MulVAL: A Logic-based Network Security Analyzer”. *Proceedings of the USENIX Security Symposium*. 2005.

[Pro] *Package "regex"*. http://www.swi-prolog.org/pack/list?p=regex. Accessed: 2016-05-19.

[Pir] *Pirated App Store client for iOS found on Apple’s App Store*. https://www.helpnetsecurity.com/2016/02/22/pirated-app-store-client-ios-found-apples-app-store/. Accessed: 2016-05-05.

[Ply] *PLY (Python Lex-Yacc)*. http://www.dabeaz.com/ply/. Accessed: 2016-05-17.

[Sma] “PSecurity Enhanced (SE) Android: Bringing Flexible MAC to Android”.

[Ube] *Researchers: Uber’s iOS App Had Secret Permissions That Allowed It to Copy Your Phone Screen*. https://gizmodo.com/researchers-uber-s-ios-app-had-secret-permissions-that-1819177235. Accessed: 2018-07-24.

[Rue08] Rueda, S. et al. “Verifying Compliance of Trusted Programs”. *Proceedings of the USENIX Security Symposium*. 2008.

[Sah08] Saha, D. “Extending Logical Attack Graphs for Efficient Vulnerability Analysis”. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2008.

[Sas06] Sasturkar, A. et al. “Policy Analysis for Administrative Role Based Access Control”. *Proceedings of the IEEE Computer Security Foundations Workshop*. 2006.

[SO08] Sawilla, R. E. & Ou, X. “Identifying Critical Attack Assets in Dependency Attack Graphs”. *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. 2008.

[She02] Sheyner, O. et al. “Automated Generation and Analysis of Attack Graphs”. *Proceedigns of the IEEE Symposium on Security and Privacy*. 2002.

[Sal] *Smart phones overtake client PCs in 2011*. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011. Accessed: 2016-05-18.

[Mar] *Smartphone OS Market Share, 2015 Q2*. http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed: 2016-05-18.

[Swi] *SWI Prolog*. http://www.swi-prolog.org/. Accessed: 2016-05-19.

[Bla] *The Apple Sandbox*. https://media.blackhat.com/bh-dc-11/Blazakis/BlackHat_DC_2011_Blazakis_Apple%20Sandbox-Slides.pdf. Accessed: 2016-02-15.

[Tru] *TrustedBSD Mandatory Access Control (MAC) Framework*. http://www.trustedbsd.org/mac.html. Accessed: 2015-11-06.

[Vfd] *VFDecrypt*. https://www.theiphonewiki.com/wiki/VFDecrypt. Accessed: 2016-05-19.

[Voi05] Voida, A. et al. "Listening in: practices surrounding iTunes music sharing". *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2005, pp. 191–200.

[Wan15] Wang, R. et al. "EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-supervised Learning". *Proceedings of the USENIX Security Symposium*. 2015.

[Wan17] Wang, R. et al. "SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android". *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 2017.

[Wan] Wang, T. et al. *Review and Exploit Neglected Attack Surface in iOS 8*. https://www.blackhat.com/docs/us-15/materials/us-15-Wang-Review-And-Exploit-Neglected-Attack-Surface-In-iOS-8.pdf. Accessed: 2018-07-24.

[Wan13] Wang, T. et al. "Jekyll on iOS: When Benign Apps Become Evil". *Proceedings of the USENIX Security Symposium*. 2013.

[Wan14] Wang, T. et al. "On the Feasibility of Large-Scale Infections of iOS Devices". *Proceedings of the USENIX Security Symposium*. 2014.

[Wat01] Watson, R. N. M. "TrustedBSD: Adding Trusted Operating System Features to FreeBSD". *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 2001.

[Wat13] Watson, R. N. "A Decade of OS Access-Control Extensibility". *Communications of the ACM* **56**.2 (2013), pp. 52–63.

[Wer13] Werthmann, T. et al. "PSiOS: Bring Your Own Privacy & Security to iOS Devices". *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2013.

[Xia] Xiao, C. *YiSpecter*. http://researchcenter.paloaltonetworks.com/2015/10/yispecter-first-ios-malware-attacks-non-jailbroken-ios-devices-by-abusing-private-apis/. Accessed: 2015-10-21.

[Xin15] Xing, L. et al. "Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS". *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2015.

[ZM04] Zanin, G. & Mancini, L. V. “Towards a Formal Model for Security Policies Specification and Validation in the SELinux System”. *Proceedings of the ninth ACM symposium on Access control models and technologies*. ACM. 2004, pp. 136–145.

[Zha18] Zhang, X. et al. “OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS”. *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018). Internet Society*. 2018.