iOracle: Automated Evaluation of Access Control Policies in iOS

Luke Deshotels
North Carolina State University
ladeshot@ncsu.edu

Răzvan Deaconescu
University POLITEHNICA of Bucharest
razvan.deaconescu@cs.pub.ro

Costin Carabaș
University POLITEHNICA of Bucharest
costin.carabas@stud.acs.upb.ro

Iulia Mandă
University POLITEHNICA of Bucharest
iulia.manda@stud.acs.upb.ro

William Enck
North Carolina State University
whenck@ncsu.edu

Mihai Chiroiu
University POLITEHNICA of Bucharest
mihai.chiroiu@cs.pub.ro

Ninghui Li
Purdue University
ninghui@cs.purdue.edu

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Modern operating systems, such as iOS, use multiple access control policies to define an overall protection system. However, the complexity of these policies and their interactions can hide policy flaws that compromise the security of the protection system. We propose iOracle, a framework that logically models the iOS protection system such that queries can be made to automatically detect policy flaws. iOracle models policies and runtime context extracted from iOS firmware images, developer resources, and jailbroken devices, and iOracle significantly reduces the complexity of queries by modeling policy semantics. We evaluate iOracle by using it to successfully triage executables likely to have policy flaws and comparing our results to the executables exploited in four recent jailbreaks. When applied to iOS 10, iOracle identifies previously unknown policy flaws that allow attackers to modify or bypass access control policies. For compromised system processes, consequences of these policy flaws include sandbox escapes (with respect to read/write file access) and changing the ownership of arbitrary files. By automating the evaluation of iOS access control policies, iOracle provides a practical approach to hardening iOS security by identifying policy flaws before they are exploited.

KEYWORDS

Access Control; Mobile Security; iOS; iPhone; Policy Modeling

ACM Reference Format:
Luke Deshotels, Răzvan Deaconescu, Costin Carabaș, Iulia Mandă, William Enck, Mihai Chiroiu, Ninghui Li, and Ahmad-Reza Sadeghi. 2018. iOracle: Automated Evaluation of Access Control Policies in iOS. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3196494.3196527

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196527

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, 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 [10] 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

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

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 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 2 provides a background on iOS security mechanisms. Section 3 overviews the iOracle framework approach and findings. Section 4 describes the design of iOracle. Section 5 provides a case study of recent jailbreaks and evaluates iOracle's utility in triaging executables with policy flaws. Section 6 evaluates iOracle's ability to discover new policy flaws. Section 7 discusses the limitations of iOracle. Section 8 presents related work. Section 9 concludes. The Appendix quantifies the protection systems for 15 iOS versions.

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

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

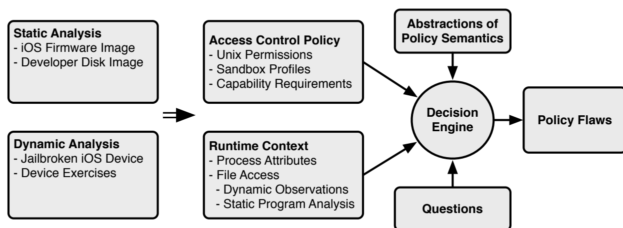

Figure 1: iOracle Overview

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.

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 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 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 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 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 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 the Appendix. Next, we use iOracle to successfully triage executables exploited in the jailbreaks presented in Section 5. In the Appendix 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 Appendix).
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.

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.

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 1.

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. 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 [11] and iRiS [9]. 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 [8] and SandScout [10] 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 4.2.

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 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

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

Table 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

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

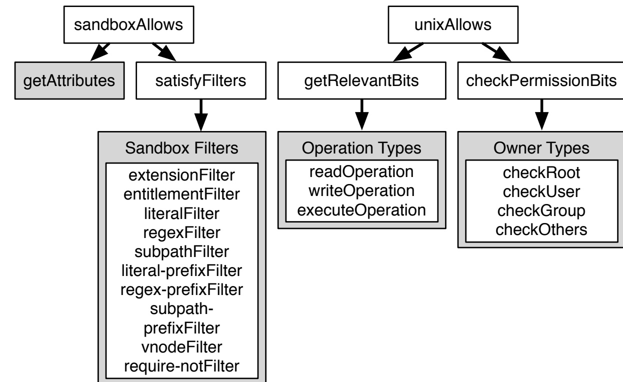

Figure 2: Simplified Hierarchy of Prolog Rules

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 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 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.

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 matching 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 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

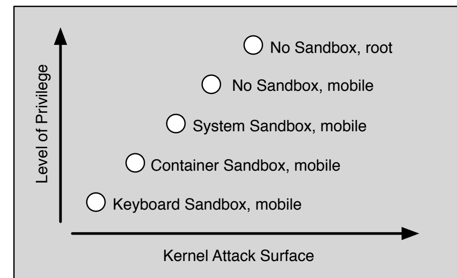

Figure 3: Privilege Levels and Kernel Attack Surface[8]

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: 1) the process user is `root`; 2) the

[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

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 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.

5.1 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 [16]. However, as of the iPhone 4S, Apple improved hardware security and boot-time jailbreaks became 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 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.

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

The following survey of known jailbreaks and their gadgets is based on Levin's book chapters [16], 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 the Appendix A.2 (Figure 4 and Figure 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.

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

[10]https://cansecwest.com/slides/2015/CanSecWest2015_Final.pdf

[11]http://proteaswang.blogspot.com/2017/

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, 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.

5.1.2 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/, 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 task port 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

[12]/private/var/mobile/Containers

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

Table 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.

third party applications was sufficiently privileged for the Pangu 7 iOS application to exploit the kernel without elevating its privileges 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 `task port` being sent as a mach-message. The `task port` belongs to an unsandboxed, `root` authorized process called `powerd`, and provides Yalu with debugger control over `powerd`.

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 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`.

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 Appendix.

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.

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.

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 `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 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.

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.

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.

[14] /private/var/mobile/Media

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 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.

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 3, this policy flaw allows a jailbreak to progress from the "No Sandbox, `mobile`" privilege level to the "No Sandbox, `root`" privilege level.

7 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 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).

8 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. [14] and Egele et al. [11] investigate potential privacy leaks in third party iOS applications. iRiS [9] improves privacy leak analysis by integrating static and dynamic analysis techniques to detect dangerous API calls. XARA [30] exploits flaws in iOS inter-process communication to provide a third party app with unauthorized access to sensitive data. Wang et al. [26] 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. [15] investigate methods for third party apps to fingerprint iOS devices. SandScout [10] models all iOS sandbox policies, but its evaluation is limited to the policy for third party applications. Wang et al. [27], and Bucicoiu et al. [2] investigate Return Oriented Programming (ROP) attacks in third party applications. In response to these ROP attacks, Davi et al. [7], Werthmann et al. [29], and Bucicoiu et al. [2] propose new security mechanisms to provide control flow integrity and fine grained access control for third party apps. Han et al. [13] investigate the potential for third party applications to abuse access to Private APIs. Chen et al. [5] 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 [16] and Miller et al. [17] provide detailed descriptions of jailbreaks and security mechanisms. Several security researchers have shared findings after reverse engineering the iOS sandbox mechanism[15, 16, 17] [1, 6]. Finally, Watson [28] 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. [3] use Datalog to model dynamic access control systems (e.g., creating processes) including Windows Vista and Asbestos. SEAL [18], 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. [4] 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. [22] use the

[15] http://www.slideshare.net/i0n1c/ruxcon-2014-stefan-esser-ios8-containers-sandboxes-and-entitlements
[16] http://2013.zeronights.org/includes/docs/Meder_Kydyraliev_-_Mining_Mach_Services_within_OS_X_Sandbox.pdf
[17] http://newosxbook.com/files/HITSB.pdf

NuSMV model checker to automatically construct attack graphs representing networks. MulVAL [19] uses Datalog to create a logic-based attack graph that integrates network configurations with data from reported vulnerabilities. Saha [20] 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 [21] 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. [12] learn from legitimate rooting applications in order to detect Android malware containing rooting exploits. SEAndroid [23] ports SELinux to Android, and EASEAndroid [25] automatically refines SEAndroid policies by using semi-supervised learning. SPOKE [24] models the attack surface of SEAndroid using functional tests.

9 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.

10 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.

REFERENCES

[1] Dionysus Blazakis. 2011. The Apple Sandbox. *In Blackhat DC*.
[2] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. 2015. XiOS: Extended Application Sandboxing on iOS. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[3] Avik Chaudhuri, Prasad Naldurg, Sriram K Rajamani, Ganesan Ramalingam, and Lakshmisubrahmanyam Velaga. 2008. EON: Modeling and Analyzing Dynamic Access Control Systems with Logic Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[4] Hong Chen, Ninghui Li, and Ziqing Mao. 2009. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
[5] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[6] Dino A Dai Zovi. 2011. Apple iOS 4 security evaluation. *Black Hat USA*.
[7] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the Network and Distributed Systems Symposium (NDSS)*.
[8] Razvan Deaconescu, Luke Deshotels, Mihai Bucicoiu, William Enck, Lucas Davi, and Ahmad-Reza Sadeghi. 2016. SandBlaster: Reversing the Apple Sandbox. (Aug. 2016). https://arxiv.org/abs/1608.04303 arXiv: 1608.04303.
[9] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iRiS: Vetting Private API Abuse in iOS Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[10] Luke Deshotels, Razvan Deaconescu, Mihai Chiroiu, Lucas Davi, William Enck, and Ahmad-Reza Sadeghi. 2016. SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[11] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
[12] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. 2017. Detecting Android Root Exploits by Learning from Root Providers. In *Proceedings of the USENIX Security Symposium*.
[13] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. 2013. Launching Generic Attacks on iOS with Approved Third-Party Applications. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*.
[14] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. 2013. Comparing Mobile Privacy Protection Through Cross-Platform Applications. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
[15] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. 2016. Fingerprinting Mobile Devices Using Personalized Configurations. *Proceedings on Privacy Enhancing Technologies (PoPETS)* 1 (2016).
[16] Jonathan Levin. 2016. *MacOS and iOS Internals, Volume III: Security & Insecurity*. Technologeeks Press.
[17] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. 2012. *iOS Hacker's Handbook*. John Wiley & Sons.
[18] Prasad Naldurg and Raghavendra KR. 2011. SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*.
[19] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. 2005. MulVAL: A Logic-based Network Security Analyzer. In *Proceedings of the USENIX Security Symposium*.
[20] Diptikalyan Saha. 2008. Extending Logical Attack Graphs for Efficient Vulnerability Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[21] Reginald E Sawilla and Xinming Ou. 2008. Identifying Critical Attack Assets in Dependency Attack Graphs. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*.
[22] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. 2002. Automated Generation and Analysis of Attack Graphs. In *Proceedigns of the IEEE Symposium on Security and Privacy*.
[23] Stephen Smalley and Robert Craig. 2013. PSecurity Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
[24] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. 2017. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*.
[25] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M Azab. 2015. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-supervised Learning. In *Proceedings of the USENIX Security Symposium*.
[26] Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee. 2014. On the Feasibility of Large-Scale Infections of iOS Devices. In *Proceedings of the USENIX Security Symposium*.
[27] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the USENIX Security Symposium*.
[28] Robert NM Watson. 2013. A Decade of OS Access-Control Extensibility. *Commun. ACM* 56, 2 (2013), 52–63.
[29] Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2013. PSiOS: Bring Your Own Privacy & Security to iOS Devices. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[30] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. 2015. Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

Table 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

A 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 3. We also compare the data extracted across versions to detect policy or context changes made by Apple in response to jailbreaks.

A.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.

A.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.

A.2.1 Name Resolution Jailbreak Responses. Figure 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.

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 [18], 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.

A.2.2 Capability Based Jailbreak Responses. Figure 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.

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.

B OTHER POLICY FLAWS

In addition to the five policy flaws presented in Section 6, we discovered three other flaws while implementing iOracle.

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

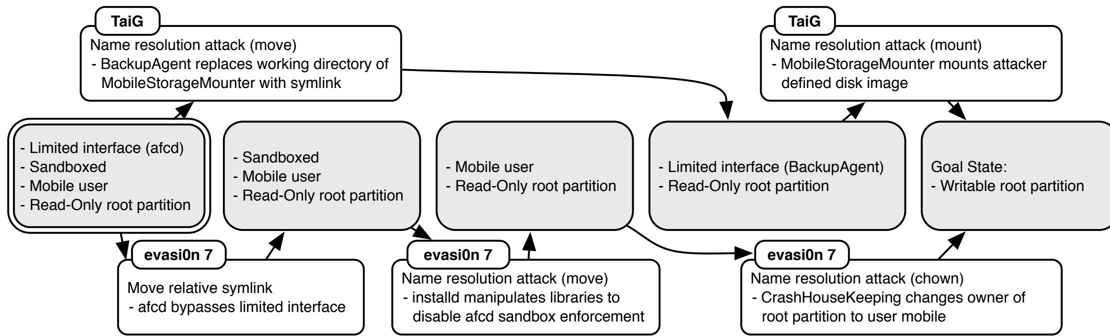

Figure 4: Name Resolution Based Jailbreak Steps

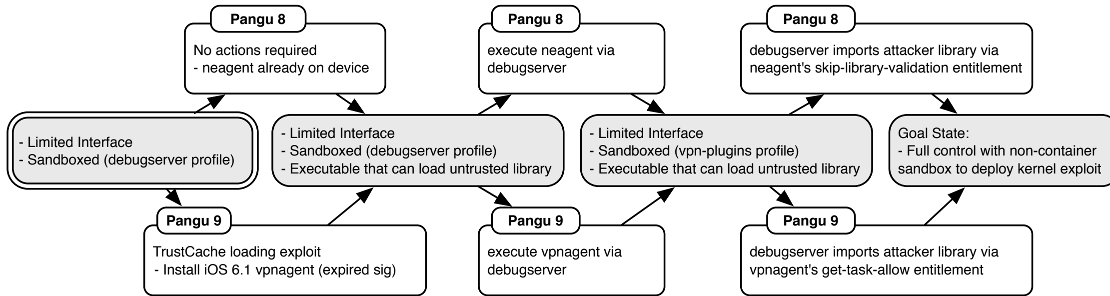

Figure 5: Capability Based Jailbreak Steps

B.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 processes with respect to file integrity.

B.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.

B.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`.[19] 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

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

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 [10]. Therefore, our proof of concept does not apply to iOS 10. However, it can affect iOS 9.3.5 (the latest version supported by 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.