SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles

Luke Deshotels
North Carolina State University
ladeshot@ncsu.edu

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

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

Lucas Davi
Technische Universität Darmstadt, Germany
lucas.davi@trust.cased.de

William Enck
North Carolina State University
whenck@ncsu.edu

Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
ahmad.sadeghi@trust.cased.de

ABSTRACT

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 this paper, 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 affect non-jailbroken devices running later versions of iOS. We are working with Apple to resolve these attacks, and we expect that SandScout will play a significant role in the development of sandbox profiles for future versions of iOS.

1. INTRODUCTION

The sale of smartphones has out-paced the sale of PCs [15]. The two dominant platforms for these smart phones are Android and iOS [16]. 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 [48, 22, 33, 52, 40, 47]. Second, others emphasize methods to detect malicious behavior either statically [29] or dynamically [28]. Third, new security mechanisms [22, 26, 50] 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

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978336

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 2 provides background information. Section 3 provides an overview of SandScout. Section 4 discusses our design. Section 5 presents our results. Section 6 provides discussion of our limitations. Section 7 presents related work. Section 8 concludes.

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.

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 [7] 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 [2]. Prior work [48, 22, 42, 34] 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 [18] 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 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.

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

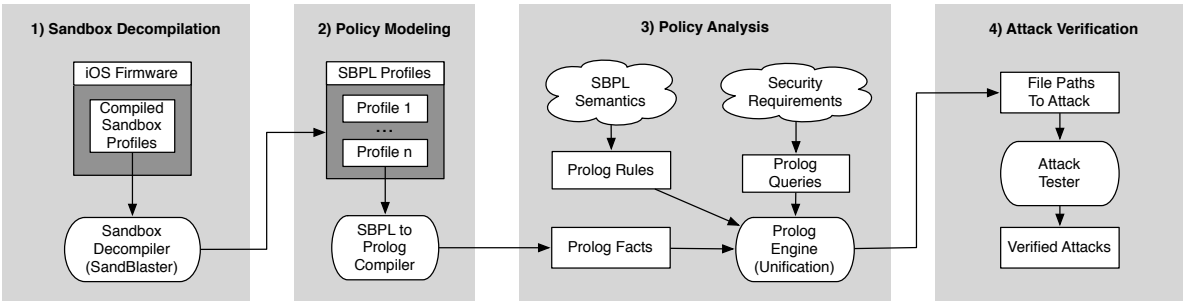

Figure 1: Overview of SandScout.

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 regular 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 4.2 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.

3. OVERVIEW

Apple's application review process is not infallible [48, 22, 42, 34, 23, 51, 13, 1, 11]. 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 [20, 53, 44, 46, 39, 37]. 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 [32, 42], 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 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 [30, 24]. 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 [4]. 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 [3] previously created a sandbox profile decompiler, his tool cannot decompile modern iOS sandbox profiles (i.e., iOS 7, 8, and 9). Esser [31] 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 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. 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.

4.1 Decompiling Sandbox Profiles

As discussed in Section 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 [21] and Esser [31]. Distinctions between SandBlaster and prior work are discussed in Section 7. A full, technical description of SandBlaster is available in our technical report [27]. Here, we limit our description to the key novel contributions of the tool.

We use a combination of our own scripts, existing tools [9, 5, 10, 8, 19], and information shared by reverse engineers [21, 31] 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 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 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 1 to explain each of the four steps in converting the graph in Figure 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 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.

4.2 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 in disjunctive normal form. To address these challenges, we created an SBPL to Prolog compiler using the ply [14] 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

Table 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

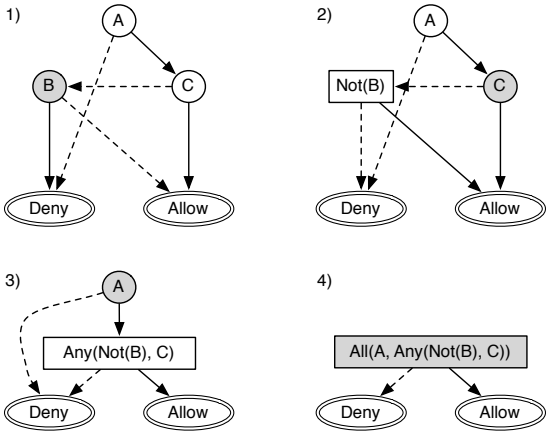

Figure 2: Converting Graph to SBPL.

```
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 3: SBPL Context Free Grammar.

simplified list of SBPL token definitions is provided in Figure 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 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.

4.3 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 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.

4.3.1 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 [12] library package for SWI-Prolog [17] 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.

4.3.2 Example Policy Queries

We now describe the three policy queries that we use to analyze the container sandbox profile in Section 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,[]).
```

Table 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

4.4 Attack Testing Application

The example queries in Section 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 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 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.

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.

5.1 Prolog Query Results

We ran each of the Prolog Queries mentioned in Section 4.3.2 on a collection of Prolog facts representing the container profile for iOS 9.0.2. The results of these queries

Table 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

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 3 presents the results of our evaluation.

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.

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.

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.

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[45]. 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.

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 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.

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 [6] 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.

7. RELATED WORK

The initial iOS security research in academic venues focused on privacy threats in third-party applications. PiOS [29] uses static taint analysis to detect privacy leaks. Han et al. [35] 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. [48] 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. [33] and further enhanced by Bucicoiu et al. [22]. Jekyll apps leverage the ability to call APIs in private frameworks. Wang et al. [47] created proof of concept attacks for infecting iOS devices with malicious applications via exploiting the iTunes synching process. iRiS [28] 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. [52] demonstrate a new attack vector for iOS malware by exploiting cross-application interfaces. Finally, Kurtz et al. [40] 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. [26] 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. [50] 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 [22] 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 [21], 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 [25]. He also shared a simplified representation of the iOS 4 container profile. Iozzo [38] 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 [41] and the iOS Hacker's Handbook [42] 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 [32]. However, this guide is significantly outdated and now it only covers a minority of the sandboxing operations available for iOS. Esser [31] 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 [49] 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 [36] models SELinux policy in Prolog to study information flow properties of its MLS enforcement. Similarly, PIDSI [43] 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. [24] 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 [30] 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.

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.

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.

10. REFERENCES

[1] 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.

[2] 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.

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

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

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

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

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

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

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

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

[11] 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.

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

[13] 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.

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

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

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

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

[18] Trustedbsd mandatory access control (mac) framework. http://www.trustedbsd.org/mac.html. Accessed: 2015-11-06.

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

[20] M. Alam, J.-P. Seifert, Q. Li, and X. Zhang. Usage control platformization via trustworthy selinux. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 245–248. ACM, 2008.

[21] D. Blazakis. The apple sandbox. *Arlington, VA, January*, 2011.

[22] M. Bucicoiu, L. Davi, R. Deaconescu, and A.-R. Sadeghi. Xios: Extended application sandboxing on ios. In *ACM Symposium on Information, Computer and Communications Security*, ASIACCS '15, 2015.

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

[24] H. Chen, N. Li, and Z. Mao. Analyzing and comparing the protection quality of security enhanced operating systems. In *NDSS*, pages 11–16, 2009.

[25] D. A. Dai Zovi. Apple ios 4 security evaluation. *Black Hat USA*, 2011.

[26] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[27] R. Deaconescu, L. Deshotels, M. Bucicoiu, W. Enck, L. Davi, and A.-R. Sadeghi. Sandblaster: Reversing the apple sandbox. Technical Report arXiv:1608.04303, Aug 2016.

[28] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56. ACM, 2015.

[29] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[30] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android Software Misuse Before It Happens. Technical Report NAS-TR-0094-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sep 2008.

[31] S. Esser. ios8 containers, sandboxes and entitlements. http://www.slideshare.net/i0n1c/ruxcon-2014-stefan-esser-ios8-containers-sandboxes-and-entitlements. Accessed: 2015-11-6.

[32] 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.

[33] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on ios with approved third-party applications. In *Applied Cryptography and Network Security*, pages 272–289. Springer, 2013.

[34] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on iOS with approved third-party applications. In *Applied Cryptography and Network Security*, ACNS '13, 2013.

[35] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng. Comparing mobile privacy protection through cross-platform applications. 2013.

[36] B. Hicks, S. Rueda, L. S. Clair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Transaction on Information and System Security*, 13(3), 2010.

[37] B. Hicks, S. Rueda, L. St Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for selinux mls policy. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):26, 2010.

[38] V. Iozzo. A sandbox odyssey. https://prezi.com/lxljhvzem6js/a-sandbox-odyssey-infiltrate-2012/. Accessed: 2015-11-7.

[39] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 5–5. USENIX Association, 2003.

[40] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. Freiling. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies*, 2016(1):4–19, 2016.

[41] M. Kydyraliev. 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.

[42] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. John Wiley & Sons, 2012.

[43] S. Rueda, D. H. King, and T. Jaeger. Verifying Compliance of Trusted Programs. In *Proceedings of the USENIX Security Symposium*, 2008.

[44] A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 13–pp. IEEE, 2006.

[45] A. Voida, R. E. Grinter, N. Ducheneaut, W. K. Edwards, and M. W. Newman. Listening in: practices surrounding itunes music sharing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 191–200. ACM, 2005.

[46] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 351–366, 2015.

[47] T. Wang, Y. Jang, Y. Chen, S. Chung, B. Lau, and W. Lee. On the feasibility of large-scale infections of ios devices. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 79–93, 2014.

[48] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Usenix Security*, volume 13, 2013.

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

[50] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: bring your own privacy & security to ios devices. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 13–24. ACM, 2013.

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

[52] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

[53] G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145. ACM, 2004.