Kobold: Evaluating Decentralized Access Control for Remote NSXPC Methods on iOS

Luke Deshotels
North Carolina State University,
Samsung Research America
ladeshot@ncsu.edu

Costin Carabaș
University POLITEHNICA of Bucharest
costin.carabas@cs.pub.ro

Jordan Beichler
North Carolina State University
jabeichl@ncsu.edu

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

William Enck
North Carolina State University
whenck@ncsu.edu

***Abstract*—Apple uses several access control mechanisms to prevent third party applications from directly accessing security sensitive resources, including sandboxing and file access control. However, third party applications may also indirectly access these resources using inter-process communication (IPC) with system daemons. If these daemons fail to properly enforce access control on IPC, *confused deputy* vulnerabilities may result. Identifying such vulnerabilities begins with an enumeration of all IPC services accessible to third party applications. However, the IPC interfaces and their corresponding access control policies are unknown and must be reverse engineered at a large scale. In this paper, we present the Kobold framework to study NSXPC-based system services using a combination of static and dynamic analysis. Using Kobold, we discovered multiple NSXPC services with confused deputy vulnerabilities and daemon crashes. Our findings include the ability to activate the microphone, disable access to all websites, and leak private data stored in iOS File Providers.**

***Keywords*-access control, iOS, iPhone, inter-process communication, fuzzer, attack surface, automation, policy analysis**

I. INTRODUCTION

Apple's iOS App Store offers over 2 million applications [26], and in 2017 was used by half a billion customers per week [27]. To protect users, third party applications run within the confines of a sandbox that limits the number of directly accessible resources. However, applications can also indirectly access sensitive resources through inter-process communication (IPC) with system daemons. For example, an application does not have direct access to a user's calendar, but can use services provided by a calendar managing daemon to view and modify calendar events. If a system daemon does not properly enforce access control, a third-party application may be able to abuse the daemon as a *confused deputy* [20] to perform some action that damages the system or violates the user's privacy.

IPC-based confused deputy vulnerabilities are not new. Woodpecker [17] uses data-flow analysis on pre-loaded Android applications to enumerate dangerous services exposed to other applications. However, several features (e.g., dynamic dispatching for method calls) make data flow analysis less practical for iOS binaries. To the best of our knowledge, there exists no systematic enumeration of iOS remote methods accessible to third party applications. The closest related work is existing IPC fuzzers for iOS [2], [22], [30] that probe for code flaws such as type confusion or dereferencing vulnerabilities, which can be exploited to obtain arbitrary code execution. However, these fuzzers do not attempt to enumerate remote methods or identify confused deputy vulnerabilities. From a policy perspective, SandScout [10] and iOracle [9] detect access control policy flaws in iOS; however, they are limited to the file system.

iOS system daemons frequently protect IPC using static capabilities called *entitlements*, which are immutable key-value pairs bound to an executable's code signature at compile time. To enforce access control policy on IPC, system daemons often use hard-coded conditional checks based on a caller's entitlements. Unfortunately, there is limited understanding of IPC on iOS, despite the growing amount of literature on iOS application [4], [8], [12], [19], [25], [29] and platform security [9], [10]. iOS defines several IPC interface abstractions, many of which exist for legacy reasons. The state-of-the-art interface type is called XPC.[1] This paper focuses on the object-oriented version of XPC called NSXPC (Next Step XPC). Specifically, we seek to answer: *Which security and privacy sensitive NSXPC methods are accessible to third party applications?* The answer represents an attack surface of remote methods that may be exploited by third party apps through IPC.

To answer this question, we address three research challenges. First, the set of entitlements available to third party applications is unknown. We identify two sets of entitlements available to third party applications: a public set accessible to all developers, and a *semi-private* set that Apple

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

provides only to select developers. For example, the Uber application was found to possess a potentially dangerous entitlement normally unavailable to third party applications [6]. Second, the set of NSXPC services accessible to third party applications is unknown. The executables that provide these services are closed source and there is no centralized policy mapping services to their entitlement requirements. Third, which NSXPC services are security or privacy sensitive is unknown. The semantics of these services are not publicly documented, and data flow analysis in iOS is nontrivial.

In this paper, we present the *Kobold*[2] framework for studying NSXPC services in iOS. Kobold leverages two key insights. First, the standardized IPC interfaces (e.g., NSXPC) contain predictable patterns in compiled code that are identifiable via static analysis. Second, error messages returned by unauthorized attempts to access IPC services can provide a model of the iOS IPC access control policy. Using these insights, Kobold provides a pattern-based, static binary program analysis to enumerate NSXPC interfaces and then dynamically uses systematic probing to extract an approximation of the access control policy encoded by conditional checks within a given service. We used Kobold to study iOS 9, 10 and 11 and found multiple NSXPC services with confused deputy vulnerabilities and daemon crashes. The discovered vulnerabilities allow third-party applications to activate the microphone, disable access to all websites, and leak private data stored in iOS File Providers. All issues have been reported to Apple. At the time of writing, Apple has provided two CVEs in response to our disclosure and is working to resolve remaining issues. After fixes are made, we plan to publicly release Kobold as open source code.

This paper makes the following contributions:

- *We present Kobold, the first framework for evaluating NSXPC access control policies implemented in iOS system code.* Kobold enumerates the NSXPC services accessible to third party applications and uses heuristics to determine which services are likely to be exploited.
- *We perform the first measurement of semi-private entitlements.* We analyze approximately six thousand popular third party applications and 100 thousand randomly selected third party applications to determine which semi-private entitlements Apple distributes to an undisclosed subset of third party developers.
- *We identify previously unknown security issues including three categories of confused deputy vulnerabilities and fourteen daemon crashes.* Our findings include crashes for root authority daemons, unprivileged access to Mobile Device Management (MDM) functionality, and microphone activation without user permission.

Kobold does not require a jailbroken device. However, a jailbroken device can provide supplemental data that may assist in identifying vulnerabilities. Furthermore, Kobold is not restricted to a specific version of iOS and can be used to study new versions as they are released.

[2]A spirit from German folklore that haunts mines.

The remainder of the paper proceeds as follows. Section II provides background on iOS IPC and access control. Section III overviews Kobold. Section IV details the implementation of Kobold. Section V presents the results of the semi-private entitlement survey. Section VI quantifies the ports, methods, arguments, and entitlement requirements enumerated by Kobold. Section VII demonstrates Kobold's ability to detect previously unknown policy flaws and crashes. Section VIII discusses limitations. Section IX overviews related work. Section X concludes.

II. BACKGROUND

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

A. Mach IPC

IPC on iOS is built upon the Mach microkernel. The primitive components of Mach IPC are mach-messages and mach-ports. A service-providing process can host remote methods by registering a name for a mach-port, and clients can send messages to that port in order to call the remote methods. The mach-port name registration is facilitated by launchd, which also assists clients in connecting to mach-ports. For example, the location daemon, locationd, offers remote methods on the mach-port named "com.apple.locationd.registration". A client process can access these methods by asking launchd to connect it to the "com.apple.locationd.registration" mach-port. If the connection is successful, the client can then send messages to the server via the mach-port. If the messages are well formed, and the client has sufficient capabilities, the server will the execute the methods for the client (e.g., locationd could provide access to the user's coordinates). While launchd plays a low-level role in securely facilitating mach-port connections, flaws in launchd are out of scope for Kobold.

XPC and NSXPC: The process of encoding and decoding mach-messages is complex, error prone, and security sensitive. Abstractions are provided by Apple to make IPC simpler for developers. The state-of-the-art interface types

are XPC and its object oriented variant, NSXPC. In object-oriented IPC, an object and its methods reside in the service-providing process, but the client can access the object as though it existed in the client's address space. Therefore, a service-providing process using NSXPC can register multiple mach-port names that each provide access to remote objects, and each remote object exposes remote methods.

NSSecureCoding: In order to mitigate type confusion attacks [14], remote methods exposed with NSXPC have strict parameter types that must adhere to a protocol called NSSecureCoding.[3] Any attempts to invoke these methods with invalid parameter types are immediately rejected. Therefore, Kobold must perform three tasks: 1) identify the mach-ports associated with NSXPC interfaces; 2) find the names of remote methods provided by remote objects; and 3) obtain the expected argument types of those remote methods.

B. IPC Access Control

Apple uses app vetting and code signing requirements to help protect iOS users from malicious applications. However, code signing and app vetting are not sufficient to stop all attacks, and researchers have demonstrated several attacks that bypass these defenses [19], [29], [33]. To mitigate such attacks, Apple has implemented multiple layers of access control including capability systems and the sandbox.

Entitlements: The capability most relevant to IPC access control is called an entitlement. Entitlements are key-value pairs statically embedded into an executable's code signature. An application's entitlements can only be changed as part of a formal app update, and the entitlements are not made visible to users installing the application. Apple uses entitlements to help determine which privileges are accessible to each application. The most dangerous entitlements (e.g., bypassing code-signing restrictions) are private and reserved for executables created by Apple. Less sensitive entitlements (e.g., inter-app audio) are publicly available to third party developers who can add them to apps by toggling switches in Xcode during development. A third, poorly understood class of semi-private[4] entitlements are not available through Xcode toggles, but can still be found in a number of third party apps on App Store.

Enforcement: Figure 1 illustrates the three locations of NSXPC IPC access control enforcement. At stage one, the sandbox can allow or deny requests to connect to specific mach-port names. The sandbox can prevent the client from making the system call that would cause launchd to make the connection. Apple must allow third party applications to access some IPC functionality (e.g., accessing location data), so it cannot use the sandbox to block access to all mach ports. Sandbox enforcement is also not sufficiently granular to support ports that offer some methods intended for third

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

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

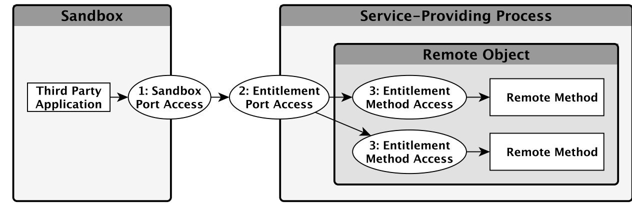

Figure 1. Stages of NSXPC Access Control: 1) Sandbox Access to Port; 2) Entitlement Checks for Port; 3) Entitlement checks for Remote Method

party apps and some methods intended for only system apps. At stage two, the service-providing process can accept or deny attempts to connect to one of its mach-ports based on the client's capabilities. Finally, at stage three, each remote method can accept or deny attempts to invoke them based on the client's capabilities. For stages two and three, service-providing processes can check the entitlements of clients by using the `SecTaskCopyValueForEntitlement` API [23]. This API allows a process to specify an entitlement key and a client (i.e., a token representing the client's id), and the API will return the value associated with that entitlement key for the specified client.

III. Overview

This paper seeks to answer the research question: *Which security and privacy sensitive NSXPC methods are accessible to third party applications?* The answer helps characterize the attack surface of iOS with respect to third party applications. Historically, iOS security has strongly relied on the App Store review process, allowing malicious apps to circumvent protections by obfuscating calls to sensitive system services [29]. Recent years have seen substantial improvements to the iOS platform's access control policies and mechanisms. However, the policies have become complex and difficult to define. Prior work [9], [10] addressed this complexity by systematically studying *file-based access control* in iOS. Kobold compliments this prior work by investigating *IPC access control*.

In order to determine which security and privacy sensitive NSXPC methods are accessible to third-party applications, we must overcome three research challenges.

- *The set of entitlements available to third-party applications is unknown.* While Xcode defines a set of "public" entitlements available to all iOS application developers, reports indicate that there is a set of "semi-private" entitlements that Apple grants to select developers [6].
- *The set of NSXPC services accessible to third party applications is unknown*. NSXPC services are dynamically resolved via service names. There is no documentation or configuration file mapping NSXPC services (i.e., method names) to corresponding daemons, much less IPC entry points within those daemons. Moreover, access control policy for accessing NSXPC services is hard-coded into daemons. Unlike prior-work [9],

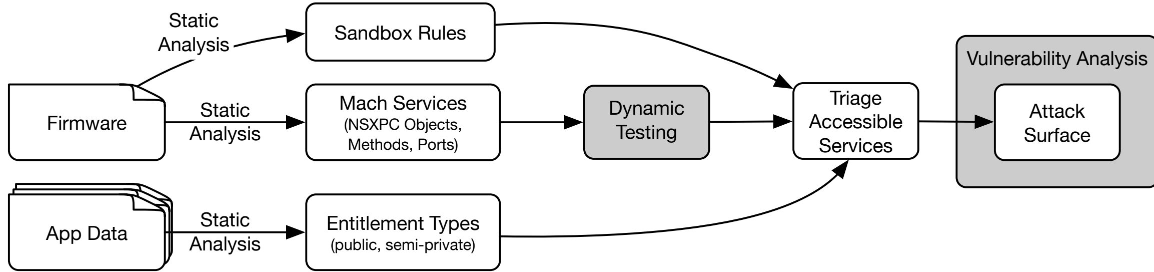

Figure 2. Kobold Overview

[10], we cannot consult a policy specification, encoded in a proprietary format or otherwise. Apple could more easily obtain the method names for IPC services. However, they cannot automatically determine where the entitlement checks will be (e.g., they could occur indirectly in libraries), and they cannot determine which resources are being protected by entitlement checks.

- *Which NSXPC services are security or privacy sensitive is unknown.* There is no public information on service semantics. And code and data flow analysis of NSXPC services is non-trivial due to the complex and closed source implementation of iOS programs.

We address these challenges through a combination of static analysis and dynamic testing, as shown in Figure 2. First, services are enumerated (i.e., identify the port, method name, and arguments). Second, Kobold triages only those services that are accessible to third party applications. Third, we use heuristics to choose accessible, security sensitive services for manual analysis.

Kobold's static analysis helps to enumerate the attack surface, while the dynamic analysis allows an analyst to triage which NSXPC services are likely to contain vulnerabilities. This approach led to the discovery of confused deputy vulnerabilities and daemon crashes that we disclosed to Apple. A simpler approach using only dynamic analysis would likely overlook several services that are rarely called at runtime. Likewise, an approach that focuses only static analysis would risk spending significant time analyzing services that are not actually accessible to third party applications.

Enumeration of services: A common approach to finding IPC vulnerabilities is to dynamically record IPC messages during normal system activity and replay these messages with slight mutations. However, this "record and replay" approach has two disadvantages. First, it will not reveal rarely used services that were not invoked during the "record" phase. Second, it is highly dependent on using a jailbroken device to record the IPC activity. Instead, we apply static analysis to extract object-oriented (i.e., NSXPC) services from iOS firmware, which is available for download from Apple.[5] This analysis is based on the intuition that high level abstractions create patterns in binaries that are amenable to static analysis. Our enumeration approach does not require a jailbroken device and will reveal the ports, method names, and argument types for services regardless of how often those services are used.

Triaging accessible services: Significant manual effort can be saved by triaging the services accessible to third party applications. We use three techniques to perform this triaging. First, we use a model of the iOS sandbox policy for third party apps (i.e., the container policy) to determine which mach ports a third party application has access to. Second, we use an iOS application to dynamically invoke services. A significant number of services provide responses in the form of completion handlers (callbacks). These responses allow us to confirm when a service was successfully accessed. Third, the sandbox model and service responses (e.g., error messages) sometimes indicate that a specific entitlement is required to access a service. In order to determine if the required entitlement can be possessed by a third party application, we performed an automated survey of the Apple App Store and created a list of entitlements observed there.

Vulnerability Analysis: Our initial dynamic testing uses uninitialized values for the variables passed as arguments into services. In many cases, uninitialized values are sufficient to trigger unusual system activity (e.g., crashes, prompts to the user, disabling system resources, audible alerts). Additional tests are performed to map specific service invocations with observed security sensitive operations. We also use the names of methods as a heuristic to prioritize methods for manual investigation. During manual investigation we can initialize variables with valid values, and we can optionally use a jailbroken device to monitor system activity (e.g., file access logs) while invoking the service.

IV. KOBOLD

Kobold is divided into three tasks. First, it performs a survey of the entitlements available to third party applications. Second, it enumerates the NSXPC services accessible to third party applications. Third, we evaluate the security sensitivity of accessible NSXPC services in order to highlight services likely to allow confused deputy attacks.

The first and second tasks are automated. We developed scripts and integrated existing tools to extract application entitlements and enumerate NSXPC services. The third task

[5] https://developer.apple.com/download/

uses fuzzing and manual analysis to investigate NSXPC service methods that are accessible and security sensitive.

A. Identify Semi-Private Entitlements

Since entitlement requirements in the sandbox can determine the set of mach-ports accessible to third party applications, our first step is to enumerate the entitlements that a third party application can possess. Finding the set of public entitlements is trivial. We enable all the capabilities available in Xcode for an iOS app and extract the entitlements from this app's signature. However, identifying semi-private entitlements requires scraping applications from the App Store and surveying their entitlements.

Definition (Semi-Private Entitlement)**.** *An entitlement is semi-private if it is possessed by a third party application on the app store, but not possessed by our experimental Xcode application with all capabilities enabled.*

Kobold's entitlement surveying framework has two stages. First, we automatically download the .ipa (iPhone application archive) files representing iOS applications from the Apple App Store. Second, we extract metadata and entitlement data from each .ipa file and search for entitlements that we have not already labeled as public. Finally, we take care to ignore entitlements from App Store applications that list "Apple" as the developer. Such applications are not installed by default on iPhones, but they can still be granted private entitlements because their code base is owned by Apple.

App Scraper: We developed an app scraper for Kobold, but we do not claim to have developed the first App Store scraper. Kobold uses accessibility options and AppleScript to manipulate iTunes on macOS, while prior work by Orikogbo et al. [25] uses a Windows virtual machine to manipulate iTunes. We collected our app samples in September 2017. Apple has officially removed the iOS app market from the default version of iTunes. However, to reproduce our analysis, it is possible to install[6] an alternative version[7] of iTunes and restore the iOS app market functionality.

Modeling Sandbox Entitlement Checks: Kobold expands upon an existing model of iOS access control called iOracle [9] in two ways: 1) modeling sandbox rules for mach-port access; and 2) enumerating the entitlements available to third party applications. By combining an enumeration of third party entitlements and the model of sandbox rules for mach-port access, Kobold can automatically map third party accessible entitlements to sandbox rules that require those entitlements. This mapping allows us to infer which mach ports are accessible to a sandboxed third party application (even if that app possesses semi-private entitlements).

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

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

B. Enumerate Accessible NSXPC Services

To invoke an NSXPC service, a client must correctly specify two targets: (1) a mach-port name and (2) a remote method associated with the service. Kobold uses two static analysis techniques and one dynamic analysis technique to find these mach-ports and methods in order to enumerate the NSXPC services accessible to third party applications. First, a mapping of mach ports to the executables that host them is extracted from a cache of configuration files extracted from the iOS firmware. Second, protocol headers that contain method names for NSXPC services are extracted statically from daemon binaries, which were extracted from iOS firmware. Third, our internally developed application attempts to invoke combinations of mach ports and method names while recording responses from these invocations.

Mapping Mach Ports to Executables: In general, iOS statically maps mach port names to the executables that host them. In theory, it is possible for services to be set up at runtime, but due to the following reasons it is usually done statically. The static mapping allows launchd to start the appropriate daemon when a service provided by that daemon is requested. A static mapping also prevents processes from pretending to host a mach port in order to steal IPC messages. The mapping of mach-ports to executables can be obtained statically by analyzing a cache of mach-port name registrations stored in `xpcd_cache.dylib`. However, parsing this file is non-trivial. First, Kobold identifies a section in the .dylib binary format that represents a .plist file and extracts that section using jtool.[8] This plist file is then converted from a binary format into xml by using the `plutil` utility. Finally, the xml formatted plist file can be parsed with regular expressions to extract a mapping of service providing executables to the mach-ports they host.

Mapping Protocols to Executables: Since NSXPC is an object-oriented interface, both the client and service provider are expected to have a list of method declarations (method names and argument types) called a *protocol*. However, these protocols are not publicly available and must be extracted from the binary executables of service providers found on the iOS firmware image. We use a static analysis tool called class-dump[9] to extract object-oriented features (i.e., protocol method declarations) from iOS daemon executable files. Using class-dump, we search service providers for interface classes associated with either NSXPCConnection or NSXPCListener classes. Class-dump extracts the protocols implemented by these interfaces, and we extract method declarations from those protocols. The extracted protocols are not guaranteed to represent NSXPC services, but we treat them as an over-approximation that can be refined through dynamic testing. For example, the NSXPCListenerDelegate protocol appears often, but it seems to act as a utility service

[8]http://www.newosxbook.com/tools/jtool.html

[9]http://stevenygard.com/projects/class-dump/

supporting connections for other NSXPC services and is not relevant to our analysis.

Mapping Ports to Protocols: At this point, mach-ports have been mapped to executables, and protocols have been mapped to executables. Kobold also removes any mach-ports that the sandbox blocks access to as discussed in Section IV-A. However, an executable could use more than one mach-port and more than one protocol. Therefore, while we have significantly reduced the possible combinations, we still need to disambiguate invalid mach-port to protocol combinations within an executable. Kobold addresses this ambiguity by attempting each combination at run time and using message feedback to determine which mach-port to protocol combinations are valid.

Bypassing Compile Time Policies: The Xcode IDE for iOS forbids developers from calling NSXPC APIs in their code. However, through reverse engineering we have confirmed that system programs on iOS do use NSXPC. Therefore, the libraries for NSXPC exist on the iOS device, but Xcode acts as a compile-time obstacle to discourage malicious or accidental abuse of low level functionality. To clarify, third party applications are expected to call libraries that will indirectly call NSXPC APIs from the third party app's address space. If developers directly invoke NSXPC APIs, they are more likely to call them with invalid or dangerous parameters. Investigating the NSXPC header file[10] in the iOS SDK (Software Development Kit) revealed that the NSXPC API we needed was augmented with the tag `__IOS_PROHIBITED`. Removing these tags from the header file allowed us to use Xcode to compile applications using NSXPC APIs.

Completion Responses: Once a client connection to a mach-port, NSXPC allows it to call associated remote methods. Many methods contain a special parameter called a completion handler that contains zero or more arguments and a block of code to be executed if the remote method completes. Kobold calls all remote methods associated with the protocols extracted and assumes that messages which trigger completion handler responses are accessible unless those completion handlers return error messages specifying entitlement requirements. If an error occurs, a helpful message describing the problem may be available in the error field of the method's completion handler. We speculate that these error messages were only intended for Apple developers since third parties are not expected to use the NSXPC APIs. However, we have found these completion handler errors to provide valuable insights since they may specify the entitlement key and value required for the method being called. An example automatically-generated completion handler is presented in Figure 3.

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

```
1  NSXPCInterface *myIf_9;
2  NSXPCConnection *myConn_9;
3
4  initWithMachServiceName:@"com.apple.commcenter.
       cupolicy.xpc"options:0];
5
6  myConn_9.remoteObjectInterface = myIf_9;
7  [myConn_9 resume];
8  myConn_9.interruptionHandler = ^{NSLog(@"id 9:
       Connection Terminated");};
9  myConn_9.invalidationHandler = ^{NSLog(@"id 9:
       Connection Invalidated");};
10 NSLog(@"id 9: Invocation has a completion handler");
11
12 typedef void (^objectOpBlock_9_2)(NSError * var_9_1);
13 objectOpBlock_9_2 blockH_9_3 = ^(NSError * var_9_1) {
14   NSLog(@"id 9: Completion message");
15   @try {
16     NSLog(@"id 9: COMPLETION HANDLER OUTPUT NSError *
           var_9_1: %@",var_9_1);
17   }
18   @catch (NSException * e) {
19     NSLog(@"Completion Handler Exception: %@", e);
20   }
21 };
22
23 @try {
24   [myConn_9.remoteObjectProxy refreshPlansInfo:
         blockH_9_3];
25 }
26 @catch (NSException * e) {
27   NSLog(@"Invocation Exception: %@", e);
28 }
```

Figure 3. Example Automatically-Generated Completion Handler

C. Security Sensitivity of NSXPC Services

We use four methods to evaluate the security sensitivity of each remote method and filter for candidate attacks: 1) we use method name semantics and entitlement requirement inconsistencies to triage methods for manual investigation; 2) we manually investigate values returned via completion handler arguments; 3) we observe user perceivable changes on the device; 4) we use a jailbroken device to provide supplemental insight into file operations and crash logs.

Method Name Semantics: Apple has not obfuscated the names of the remote methods, and Objective-C requires each parameter to be mentioned in the method name. Therefore, the method names contain a significant amount of semantic information related to their functionality. The method name of each method with a successful completion handler (including the following example) was reviewed manually by an author with experience investigating iOS access control policies. For example, we would manually classify the following method declaration as security sensitive due to the terms "Recording", "Dictation", and "Speech".

```
1  - (oneway void)startRecordingFor
2     PendingDictationWithLanguageCode:(NSString *)
3     arg1 options:(AFDictationOptions *)
4     arg2 speechOptions:(AFSpeechRequestOptions *)
5     arg3 reply:(void (^)(NSXPCListenerEndpoint *))
         arg4;
```

Entitlement Inconsistencies: Each remote method is an opportunity for developers to make access control mistakes. We assume that each method associated with a mach port has a similar level of security sensitivity. We also assume

that methods requiring entitlements are security sensitive. Therefore, any method that does not require an entitlement and shares a port with a method that does require an entitlement is considered to be security sensitive. For example, if a port has 9 methods that require an entitlement and one that does not, we assume that a developer may have forgotten to add an entitlement requirement to the unrestricted method.

Observations Without Jailbreak: A significant amount of system activity can be observed when fuzzing remote methods on a non-jailbroken device. Many methods contain parameters that represent return values, and these values may contain security sensitive data after the method finishes executing. In one step, we initialize method parameters to either simple values such as 0 for numbers or an empty string. Then we use values previously collected via static analysis or dynamic analysis, e.g., names of open files or file names and strings used in programs. If invalid method arguments cause device features to be disrupted (e.g., Internet access, configuration options), a human observer may detect these changes by manually investigating the device state. Effects such as sounds or prompts that occur while running the fuzzing application can also be documented. Finally, crash reports are visible to iOS users through the Settings menu. These reports can be used to detect crashes caused by method invocation on stock or jailbroken devices.

Observations With Jailbreak: We perform two types on dynamic analysis to observe system activity on a jailbroken device. First, we use filemon[11] to track all file operations (i.e., the process, file source, file destination, and operation type) on the device. Second, we monitor crash log files.[12,13]

V. Identified Semi-Private Entitlements

As discussed in Section IV-A, an application's entitlements play a significant role in determining which mach ports and remote methods the app has access to. Public entitlements are trivially identifiable by assigning them to an experimental iOS application created in Xcode with all capabilities toggled on. However, Apple also distributes an unknown set of semi-private entitlements to a subset of third party developers. Therefore, we need to answer the research question: *"What semi-private entitlements can be acquired by third party applications?"*. To answer this question, we performed a survey of the Apple iOS App Store in order to search for third party applications with entitlements that are not in our set of known public entitlements. We conducted the survey in September and October 2017.

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

100k Random App Sample: In addition to our survey of popular applications, we also collected 100 thousand randomly selected applications. This collection was performed automatically by a tool we developed. It took two weeks for it to download the applications and another two days to extract the entitlements used in those applications. However, within this sample we did not detect any new types of semi-private entitlement that had not been observed in our sample of six thousand popular applications. This finding supports the assumption that a sample set of popular applications is sufficient to study semi-private entitlements. Therefore, *quantities listed in this section are with respect to the six thousand popular apps.*

Results: We discovered 17 semi-private entitlements. To the best of our knowledge, only 4 have clear documentation from Apple about the process of requesting them. The process requires sending an email to a specific team within Apple to request access. The four entitlements publicly documented as semi-private are pass-presentation-suppression,[14] payment-pass-provisioning,[15] previous-application-identifiers,[16] and HotspotHelper.[17] The semi-private entitlements are listed in Table I.

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

Sharing Resources with Daemons: Two applications possess unique semi-private entitlements that seem to otherwise be used by system applications. As revealed by Strafach [6], the Uber application has the explicit-graphics-priority[18] entitlement which is used by jailbreak applications to build

[11] http://www.newosxbook.com/src.jl?tree=listings&file=3-filemon.c

[12] /private/var/mobile/Library/Logs/AppleSupport/

[13] /private/var/mobile/Library/Logs/CrashReporter/

[14] com.apple.developer.passkit.pass-presentation-suppression

[15] com.apple.developer.payment-pass-provisioning

[16] https://developer.apple.com/library/content/technotes/tn2319/_index.html

[17] com.apple.developer.networking.HotspotHelper

[18] com.apple.private.allow-explicit-graphics-priority

Table I
SEMI-PRIVATE ENTITLEMENTS USED BY THIRD-PARTY APPS

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

screen recording applications.[19] This correlation implies that Uber could have recorded the user's screen while the application ran in the background. Uber quickly removed the entitlement after its existence was made public, thus highlighting the importance of transparency for entitlements. Further, Netflix has an entitlement with allow-mpeg4streaming.[20] This entitlement is also possessed by system applications built into iOS, but the best of our knowledge the entitlement is undocumented. While Netflix is not the only video streaming application in our sample (e.g., Hulu and Amazon Prime Video), it is the only third party application in our sample with this entitlement.

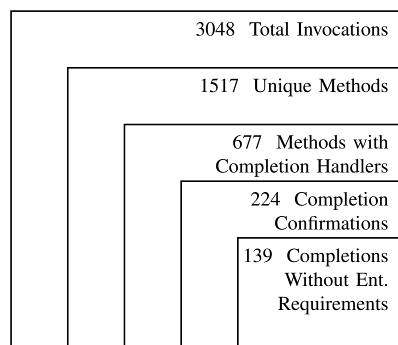

Figure 4. NSXPC Method Invocation Quantitative Results

VI. EMPIRICAL STUDY OF NSXPC ATTACK SURFACE

In addition to searching for confused deputy attacks, we also use Kobold to perform a quantitative analysis of NSXPC methods. This analysis enumerates accessible methods and measures characteristics such as the number and type of arguments required for each method. Entitlement requirements for NSXPC methods are also investigated.

Hierarchical Results: Figure 4 illustrates the number of invocations, unique methods, completion handlers, completion confirmations, and entitlement free methods dynamically tested with Kobold, on an iOS 11.3.2 device, using an application with only default entitlements. Default entitlements (e.g., an app identifier) are embedded into every application's signature and do not require toggles in Xcode. Kobold's static analysis phase extracted 276 sandbox accessible mach-ports and 3048 candidate remote methods to invoke. 1517 unique methods were tested with the mach-ports associated with the daemons each method was extracted from. Note that due to mach-port to protocol mapping ambiguity, many of those methods could be assigned to incorrect ports. 677 of the methods tested contained completion handlers, and 224 of those methods returned completion handler confirmations when invoked. As shown by Table II, of the 224 remote methods with successful completion messages, 139 did not require entitlements, 8 required unspecified entitlements, and 77 required specific entitlements.

Completion Handlers: Completion handlers are blocks of code with arguments that can be passed to a remote method as one of the method's arguments. If the remote method completes, the code block assigned to the completion handler is executed. The completion handler's arguments (e.g., NSError or NSString values) can be initialized with data from the daemon, and used in the scope of the completion handler's code block. Kobold uses this code block to output a completion confirmation that can be detected when inspecting the output of our application. This output allows us to determine whether a remote method with a completion handler has run or not. However, as shown in Figure 4 approximately half of the unique methods tested did not have completion handlers and could not be labeled as accessible or inaccessible without further analysis.

Impact of Entitlements on NSXPC Services: We identified

[19]https://stackoverflow.com/questions/32239969/iomobileframebuffergetlayerdefaultsurface-not-working-on-ios-9

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

Table II
PER METHOD ENTITLEMENT REQUIREMENTS BASED ON ERROR MESSAGES

Entitlement Key Requirements Based on Error Message	**Number of Methods**
No Entitlement Required	139
Unspecified Entitlement Required	8
com.apple.managedconfiguration.profiled-access	1
com.apple.managedconfiguration.profiled.shutdown	1
com.apple.managedconfiguration.mdmd.push	2
com.apple.managedconfiguration.profiled.migration	2
com.apple.managedconfiguration.profiled.usercompliance	4
com.apple.managedconfiguration.profiled.get	5
com.apple.managedconfiguration.profiled.provisioningprofiles	5
com.apple.managedconfiguration.mdmd-access	7
com.apple.managedconfiguration.profiled.configurationprofiles	10
com.apple.private.mobileinstall.allowedSPI	18
com.apple.managedconfiguration.profiled.set	22

Table III
MACH PORT ENTITLEMENT REQUIREMENTS ENFORCED IN SANDBOX

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

Table IV
METHODS BY NUMBER OF ARGUMENTS

Number of Arguments	**Methods With That Number of Arguments**
0	331
1	1078
2	924
3	442
4	167
5	75
6	16
7	8
8	1
9	5
10	1

two conditional sandbox rules allowing access to mach-ports based on public or semi-private entitlements. These results are represented by Table III. One mach-port was accessible through a public entitlement, and one mach-port was accessible through a semi-private entitlement. The methods with error messages that specified entitlement requirements only specified private entitlements (those not accessible to third party applications on the App Store). The port associated with the semi-private entitlement did not map to any NSXPC methods (perhaps it uses another type of IPC interface). The port associated with the Siri entitlement did map to 221 potential remote method invocations. However, our dynamic tests did not cause any of these method invocations to trigger completion handler responses even if the calling application had the Siri entitlement.

Number of Arguments: The number of arguments in a method's declaration plays a significant role in determining the difficulty of invoking a remote method successfully and whether or not that method can be exploited. For example, a method with zero arguments is trivial to invoke correctly, but unlikely to be exploitable. At the other extreme, A method containing 10 arguments has a larger attack surface but it may be difficult to find valid values for all arguments. Table IV shows the number of methods with various amounts of arguments (i.e., 0 to 10 arguments). Note that a completion handler is treated as a single argument with respect to a remote method, but the completion handler may have its own arguments. Note that these values are inferred using all 1517 extracted potential NSXPC methods. Therefore, there may be methods included that were not accessible or were false positives (i.e., methods that are not remotely exposed).

Types of Arguments: Table V lists the data types that appear most frequently in declarations of the methods invoked by Kobold. We categorize these data types into three groups, primitives, documented, and undocumented. Primitive types consist of those low level types that appear in the C programming language (e.g., int, long, double). Documented types (e.g., NSString) are abstractions constructed upon primitive types, and they are documented officially by Apple [11]. Undocumented types (e.g., AFSpeechRequestOptions) are abstractions built upon primitive types, but these data types are not officially documented by Apple. While primitive values can be fuzzed using random values, it is difficult to find acceptable values for more complex types. Apple's documentation may provide hints regarding initialization of documented types, but a thorough analysis of the values expected by NSXPC remote methods may require dynamic analysis, symbolic analysis, or extensive reverse engineering of the remote method.

Intra-Port Entitlement Consistency: Table VI lists the number of methods with successful completion handlers with their respective mach-ports. The methods are also

Table V
TOP 40 DATA TYPES IN EXTRACTED METHODS

Data Type	Occurrences in Method Declarations	Classification
void	1866	primitive
NSString *	488	documented
NSError *	426	documented
_Bool	334	primitive
oneway void	234	primitive
NSDictionary *	185	documented
NSArray *	150	documented
NSData *	98	documented
NSUUID *	95	documented
unsigned long long	75	primitive
long long	66	primitive
NSURL *	66	documented
UIApplication *	47	documented
double	39	primitive
int	38	primitive
NSNumber *	24	documented
unsigned int	21	primitive
NSSet *	18	documented
IDSService *	18	undocumented
NSFileManager *	16	documented
NSURLSession *	15	documented
NSXPCListenerEndpoint *	14	documented
id	12	undocumented
NSDate *	11	documented
IDSAccount *	11	undocumented
DRDragSession *	11	undocumented
CSSpeechController *	11	undocumented
NSURLRequest *	10	documented
AFSpeechRequestOptions *	9	undocumented
unsigned char	8	undocumented
APSConnection *	8	undocumented
NSURLSessionTask *	7	documented
NSFileHandle *	7	documented
NDApplication *	7	undocumented
MCProfileConnection *	7	undocumented
AFAudioPlaybackRequest *	7	undocumented
PBItemCollection *	6	undocumented
IDSMessageContext *	6	undocumented
GKGameSession *	6	documented
GKCloudPlayer *	6	documented

divided into two categories: 1) those that do not require entitlements; and 2) those that do require entitlements. Entitlement requirements are inferred from error messages provided in completion handlers. Mach-ports with methods in both categories are considered to have inconsistent entitlement policies, and have been highlighted in Table VI. As demonstrated with MDM functionality in Section VII, inconsistent entitlement policies may represent access control flaws where security sensitive methods are accidentally made available to unprivileged clients.

Triage strategy: Of all 3048 total invocations shown in Figure 4, 1517 unique methods were called in several testing campaigns, while the phone was monitored for undefined or unexpected behaviour. A new testing campaign with a subset of methods was started once such an event was triggered.

VII. FINDINGS

Kobold led to the discovery of confused deputy vulnerabilities and daemon crashes. These findings were revealed on an iOS 11 jailbroken device and were reproduced on a stock iOS 12 device (the latest major version at the time of writing). We disclosed our findings to Apple in the form of Proof of Concept (PoC) iOS applications.

A. Confused Deputy Vulnerabilities

Table VII lists the confused deputy vulnerabilities detected by Kobold. These vulnerabilities can be grouped into three categories: 1) File Provider information leaks; 2) Microphone activation; 3) Unprotected Mobile Device Management (MDM) services. These vulnerabilities were discovered on a jailbroken iPhone 5s running iOS 11.1.2 and have been confirmed with PoC applications on a non-jailbroken 6th Generation iPod Touch running iOS 12.0.1.

File Provider State Dump: The File Provider daemon provides a method that replies with state information for the applications with File Provider functionality running on the device (e.g., Google Drive, Microsoft OneDrive). This leaked state information can be abused by a third party application in three ways. First, the leaked information reveals the names of other third party apps that have been installed if those applications use File Provider functionality. Second, the leaked information reveals UUIDs used in app directory names, which do not change upon rebooting the device. Therefore, these leaked UUIDs could be used for device fingerprinting. Finally, the third party app can infer the names of files in File Provider directories. There is a simple side channel in iOS that allows a process to determine whether a file exists or not by attempting to read the file's metadata. If the file exists, a permission denied error may occur, or the metadata may be read. If the file does not exist, an error will specify that the file does not exist. Since the attacker must correctly guess the file path, this side channel is defeated by UUIDs in file paths. However, the File Provider data leak reveals those UUIDs to third party applications allowing them to begin inferring the names of files in File Provider directories. This inference could be accelerated through the use of a dictionary of interesting file names to check for. In response to our disclosure, Apple has resolved this issue with CVE-2018-4446.

Activate Voice Dictation: By invoking methods that start voice dictation sessions, a third party application can briefly activate the microphone without user permission (i.e., the user has not enabled microphone access in their Privacy Settings). This method causes a bell to ring, signalling that the microphone has been activated. Using uninitialized variables, the application does not gain access to the audio recording, and the microphone is only activated for about 1 second. In response to our disclosure, Apple has resolved this issue with CVE-2019-8502.

Table VI
METHODS PER MACH PORT. INCONSISTENT ENTITLEMENT REQUIREMENTS HIGHLIGHTED.

Port Name	Without Entitlement Requirements	With Entitlement Requirements
com.apple.DragUI.druid.destination	1	0
com.apple.DragUI.druid.source	1	0
com.apple.FileProvider	28	0
com.apple.accessories.externalaccessory-server	1	0
com.apple.assistant.analytics	1	0
com.apple.assistant.dictation	3	0
com.apple.coreservices.lsuseractivitymanager.xpc	8	0
com.apple.devicecheckd	1	0
com.apple.managedconfiguration.mdmdservice	0	9
com.apple.managedconfiguration.profiled.public	21	50
com.apple.mobile.installd	1	18
com.apple.nano.nanoregistry.paireddeviceregistry	37	8
com.apple.nsurlsessiond	4	0
com.apple.nsurlstorage-cache	3	0
com.apple.parsecd	5	0
com.apple.pasteboard.pasted	8	0
com.apple.replayd	3	0
com.apple.sharingd.nsxpc	1	0
com.apple.voiceservices.tts	9	0
com.apple.wcd	3	0

Table VII
CONFUSED DEPUTY VULNERABILITIES

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

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

B. Daemon Crashes

Kobold detected crashes on a jailbroken iPhone 5s running iOS 11.1.2 and a stock 6th Generation iPod Touch running iOS 12.0.1. The crashes detected are listed in Table VIII which lists ten executables with a total of 14 unique crashes based on stack trace analysis. The locationd and wcd[21] crashes could not be triggered on the stock iPod, but we speculate that this difference is due to hardware differences since the iPod does not have a GPS sensor and does not support Apple Watch connectivity. Three of the crashed daemons run with root authority. If attackers are able to exploit the causes of these crashes, the root authority daemons would be valuable targets.

Crash Types: We categorize three types of crash based on the way the process was terminated: 1) abort signal; 2) segmentation fault; 3) killed by watchdog. First, seven crashes (three daemons) terminate when the daemon sends an abort signal. This signal could be the result of asserting that a value is null and aborting the process in response. Second, six crashes (six daemons) terminate when daemons

[21] a daemon related to the Apple Watch

Table VIII
DAEMON CRASHES

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

attempt to access invalid memory addresses at or near the zero address. We speculate that these unusual memory accesses are caused by Kobold's default use of uninitialized variables as remote method arguments. Since segmentation faults imply that the daemon is attempting to use corrupted values, we consider segmentation fault crashes more significant than abort signal crashes. All of our root authority daemon crashes are due to segmentation faults. Third, the Preferences[22] crash is unique in that the process freezes and is killed by a watchdog process after 10 seconds of inactivity.

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

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

Inconsistent Entitlement Enforcement: When investigating the methods crashing replayd, we noticed an inconsistency in one method's entitlement enforcement. A remote method called startRecording[23] is provided by replayd and returns an error message specifying a required entitlement, if the remote method is the only one invoked by our application. However, if our application invokes a set of twelve other remote methods before invoking the replayd method, the entitlement requirement error is not returned. Instead, the method triggers a prompt asking the user for permission to record the screen. If the user accepts the prompt, replayd will crash (the crash is likely due to Kobold's use of uninitialized variable values). This finding suggests that state-based conditions (e.g., the set of methods previously invoked) can lead to entitlement enforcement failures.

VIII. LIMITATIONS

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

Closed Source System: Several limitations of Kobold are inherent to the closed source nature of iOS. Since we do

[22] Also known as Settings.

[23] Method name has been simplified. The full name appears in Table VIII

not have ground truth for the set of third party accessible NSXPC remote methods, we cannot quantify the number of methods that Kobold may have failed to detect. While the confused deputy vulnerabilities we present in this paper are clear security concerns, we do not have an access control policy specification to compare our findings to. A jailbreak is not required to use Kobold, but a jailbreak would provide logging tools that make dynamic analysis of IPC functionality significantly easier. However, not all versions of iOS have been jailbroken, and there is no guarantee of future jailbreaks. Additionally, our model of the iOS sandbox is not a perfect reversal of the iOS sandbox policies. New sandbox filters were added in iOS 11 that iOracle was unable to reverse engineer. Therefore, there may be mach ports that are accessible to third party applications but not detected by this version of Kobold.

Argument Values: While Kobold can statically extract the data types of remote argument methods, it does not automatically determine which values those arguments should be initialized with. Simple variables can be easily assigned various values (e.g., integers or strings), but correctly initializing complex, undocumented class types requires significant reverse engineering or dynamic analysis of the method being invoked during normal runtime operations.

Scope: Kobold does not detect NSXPC remote methods provided by shared libraries, and it does not detect the other interfaces for remote methods other than NSXPC (e.g., XPC or Mach Interface Generator). The general approach of combining static and dynamic analysis may be applicable to these interfaces, but they are less amenable to static analysis (i.e., class-dump will not detect their methods). A small number (less than ten) of problematic method invocations were intentionally removed from analysis due to Xcode errors preventing compilation. The entitlement survey performed by Kobold does not include paid applications and only analyzes a sample of the free iOS applications available at one time. A longitudinal study of significantly more applications including paid applications may reveal new semi-private entitlements as well as trends in their distribution over time.

Black Box Testing: Kobold relies on error message semantics to infer decentralized entitlement requirements for remote methods. However, more sophisticated analysis methods such as symbolic execution or backtracing may detect additional entitlement requirements missed by Kobold. Kobold uses completion handler messages to determine which remote methods were invoked successfully. However, a significant number of the remote methods Kobold extracts do not have completion handlers. Therefore, a grey box testing form of confirming remote method invocation such as automated setting of debugger breakpoints in daemon code could reveal more third party accessible methods.

IX. Related Work

Kobold is related to work in six fields: 1) iOS access control policy analysis; 2) Android access control policy analysis; 3) iOS IPC analysis; 4) iOS exploitation; 5) iOS application analysis; and 6) fuzzing.

iOS Access Control: Kobold uses tools produced by two prior works in order to identify mach-ports that are accessible to third party applications through the sandbox. SandScout [10] reverse engineers and models iOS sandbox policies, but it does not model the semantics of entitlements requirements. However, iOracle [9] builds upon SandScout in several ways including the modeling of Unix Permissions and capabilities such as entitlements and sandbox extensions. Therefore, iOracle allows Kobold to input a set of third party accessible entitlements and automatically determine which mach-ports are accessible through the sandbox for an application with those entitlements. SandScout and iOracle build upon prior work by reverse engineers who pioneered research into the iOS sandbox [3], [7], [13], [24].

Android Access Control: Several research papers discuss Access Control in Android. Kratos [28], AceDroid [1], and ACMiner [15] discover inconsistencies in security policy enforcement. ARF [16] identifies re-delegation vulnerabilities in Android system services. Finally, Invetter [36] focuses on the widespread yet undocumented input validation problem.

iOS IPC Analysis: To the best of our knowledge, Kobold is the first systematic exploration of NSXPC remote methods, but there has been prior work that investigates Apple's IPC mechanisms. Han et al. [18] fuzzed Apple driver interfaces by dynamically observing system behaviors to infer dependencies between API calls (i.e., calling functions in a certain order or using the return value of a function as an argument to another function). The Pangu team [30] presented their approach to fuzzing XPC services in order to exploit data dereference operations. Beer [2] fuzzed XPC services in order to identify opportunities for type confusion attacks. Kydyraliev [22] explored Mach Interface Generator (MIG) services by observing messages sent at runtime and replaying those messages with mutations in order to trigger crashes. Kobold differs from these prior works in two ways. First, it seeks confused deputy vulnerabilities which must consider both the method's functionality and accessibility. Second it uses static analysis to determine the mach-ports, names, and argument types of remote methods instead of dynamic analysis which may miss methods that were not invoked at runtime.

iOS Attacks: Kobold assumes that the confused deputy attacks and crashes we discovered can be deployed by a third party app able to pass Apple's app vetting process and infiltrate the app store. This assumption is based on three prior works on modifying iOS app behavior after passing the vetting process. Wang et al. [29] used return oriented programming (ROP) to modify their program's

control flow after it had passed the app vetting process and been published to the App Store. XiOS [4] improved upon the work of Wang et al. by reducing the attack's complexity and proposing an attack mitigation in the form of an in-line reference monitor. Finally, Han et al. [19] used obfuscation techniques conceptually similar to Java reflection in order to bypass app vetting and invoke private API calls after publishing to the app store.

iOS App Analysis: Kobold's entitlement extraction is a form of app analysis. PiOS [12] uses backtracing to determine register values for Objective-C dispatch calls, allowing static analysis to infer which function will be executed. This anlaysis helps detect when applications invoke private API calls. iRiS [8] adds a dynamic analysis component using forced execution to resolve dispatches that were difficult to infer through static analysis. Kobold uses a similar approach as CRiOS [25] for app scraping, and CRiOS analyzes third party applications for network security issues. Chen et al. [5] use the intuition that malicious libraries detected in Android applications may have similarly malicious counterparts on the iOS platform. They analyze malicious Android libraries, and use the findings from Android to help detect the counterparts of those libraries on the iOS platform, which is less amenable to analysis. iCredFinder [32] analyzes iOS applications that use popular Software Development Kits (SDKs) and automatically searches for misused credentials associated with those SDKs.

Fuzzing: As future work, Kobold can be expanded by applying state of the art fuzzing techniques. Mutational fuzzing techniques such as those used by American Fuzzy Lop (AFL) [35] could help generate input values for simple data types (e.g., strings and integers). Furthermore, there is already an experimental port [31] of AFL to the iOS platform. Hybrid fuzzing techniques such as QSYM [34] combine fuzzing with concolic analysis to efficiently choose input values. Finally, any evaluation of a fuzzing system for iOS should adhere to the guidelines set by Klees et al. [21] in their evaluation of common flaws in fuzzing research.

X. Conclusion

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

Acknowledgments

We thank David Wu and Iulia Mandă for their assistance.

This work was supported in part by the Army Research Office (ARO) grant W911NF-16-1-0299, the National Science Foundation (NSF) CAREER grant CNS-1253346, and a grant of Romanian Ministry of Research and Innovation, CCCDI - UEFISCDI, project number PN-III-P1-1.2-PCCDI-2017-0272 / 17PCCDI-2018, within PNCDI III. 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] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection," in *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS)*, Feb. 2018.

[2] I. Beer, "Auditing and Exploiting Apple IPC," https://thecyberwire.com/events/docs/IanBeer_JSS_Slides.pdf, 2015, accessed: 2018-07-24.

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

[4] M. Bucicoiu, L. Davi, R. Deaconescu, and A.-R. Sadeghi, "XiOS: Extended Application Sandboxing on iOS," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[5] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "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*, 2016.

[6] K. Conger, "Researchers: Uber's iOS App Had Secret Permissions That Allowed It to Copy Your Phone Screen," https://gizmodo.com/researchers-uber-s-ios-app-had-secret-permissions-that-1819177235, 2017, accessed: 2018-07-24.

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

[8] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iRiS: Vetting Private API Abuse in iOS Applications," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[9] L. Deshotels, R. Deaconescu, C. Carabas, I. Manda, W. Enck, M. Chiroiu, N. Li, and A.-R. Sadeghi, "iOracle: Automated Evaluation of Access Control Policies in iOS," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[10] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi, "SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[11] A. Documentation, "Foundation," https://developer.apple.com/-documentation/foundation?language=objc#overview, 2019, accessed: 2018-11-13.

[12] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[13] S. Esser, "ios 8 containers, sandboxes and entitlements," http://www.slideshare.net/i0n1c/ruxcon-2014-stefan-esser-ios8-containers-sandboxes-and-entitlements, 2014, accessed: 2015-11-6.

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

[15] S. A. Gorski III, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. Bartel, "ACMiner: Extraction and Analysis of Authorization Checks in Androids Middleware," in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2019.

[16] S. A. Gorski III and W. Enck, "ARF: Identifying Re-Delegation Vulnerabilities in Android System Services," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, May 2019.

[17] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones." in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2012.

[18] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[19] 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 *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2013.

[20] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[21] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[22] 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, 2013, accessed: 2015-11-6.

[23] J. Levin, "A (long) evening with mobile_obliterator and a look into ios entitlements," http://newosxbook.com/articles/EveningWithMobileObliterator.html, 2013, accessed: 2015-11-9.

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

[25] D. Orikogbo, M. Büchler, and M. Egele, "CRiOS: Toward Large-Scale iOS Application Analysis," in *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2016.

[26] A. P. Release, "App Store shatters records on New Year's Day," https://www.apple.com/newsroom/2017/01/app-store-shatters-records-on- new-years-day/, 2017, accessed: 2018-07-24.

[27] ——. (2018) App Store kicks off 2018 with record-breaking holiday season. https://www.apple.com/newsroom/2018/01/app-store-kicks-off-2018-with-record-breaking-holiday-season/. Accessed: 2018-07-24.

[28] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. Mao, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[29] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When Benign Apps Become Evil," in *Proceedings of the USENIX Security Symposium*, 2013.

[30] T. Wang, H. Xu, and X. Chen, "Review and Exploit Neglected Attack Surface in iOS 8," https://www.blackhat.com/docs/us-15/materials/us-15-Wang-Review-And-Exploit-Neglected-Attack-Surface-In-iOS-8.pdf, 2015, accessed: 2018-07-24.

[31] W. Wang and Z. Wang, "Make iOS App more Robust and Security through Fuzzing," https://ruxcon.org.au/assets/2016/slides/Make_iOS_App_more_Robust-_and_Security_through_Fuzzing-1476442078.pdf, 2016, accessed: 2018-11-11.

[32] H. Wen, J. Li, Y. Zhang, and D. Gu, "An Empirical Study of SDK Credential Misuse in iOS Apps," in *Proceedings of the Asia-Pacific Software Engineering Conference*, 2018.

[33] 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 ACM Conference on Computer and Communications Security (CCS)*, 2015.

[34] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the USENIX Security Symposium*, 2018.

[35] M. Zalewski, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, Year Unspecified, accessed: 2018-11-11.

[36] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating Insecure Input Validations in Android Services," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.