Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates

Min Zheng[†], Hui Xue[§], Yulong Zhang[§], Tao Wei[§] and John C.S. Lui[†]
[†] The Chinese University of Hong Kong, [§] FireEye Inc.

ABSTRACT

Compared with Android, the conventional wisdom is that iOS is more secure. However, both jailbroken and non-jailbroken iOS devices have number of vulnerabilities. For iOS, apps need to interact with the underlying system using Application Programming Interfaces (APIs). Some of these APIs remain undocumented and Apple forbids apps in App Store from using them. These APIs, also known as "*private APIs*", provide powerful features to developers and yet they may have serious security consequences if misused. Furthermore, apps which use private APIs can bypass the App Store and use the "*Apple's Enterprise/Developer Certificates*" for distribution. This poses a significant threat to the iOS ecosystem. So far, there is no formal study to understand these apps and how private APIs are being encapsulated. We call these iOS apps which distribute to the public using enterprise certificates as "enpublic" apps. In this paper, we present the design and implementation of iAnalytics, which can automatically analyze "enpublic" apps' private API usages and vulnerabilities. Using iAnalytics, we crawled and analyzed 1,408 enpublic iOS apps. We discovered that: 844 (60%) out of the 1408 apps do use private APIs, 14 (1%) apps contain URL scheme vulnerabilities, 901 (64%) enpublic apps transport sensitive information through unencrypted channel or store the information in plaintext on the phone. In addition, we summarized 25 private APIs which are crucial and security sensitive on iOS 6/7/8, and we have filed one CVE (Common Vulnerabilities and Exposures) for iOS devices.

1. INTRODUCTION

As of the end of 2013, Apple has attracted around 800 million iOS users [21] and there are over one million apps in the iOS App Store [12]. According to Apple's "Creating Jobs Through Innovation" report[4], there are more than 275,000 registered iOS developers in the U.S. Despite the popularity, few iOS malware have been discovered [32]. In addition, it was reported [24] that iOS is more secure than Android due to its controlled distribution channel and comprehensive apps review. However, there are still potential risks for iOS systems.

In February 2012, Apple banned all apps from Qihoo [3], a prominent Chinese vendor for anti-virus software, web browser and search engine. This major incident happened because Qihoo used iOS private APIs and encrypted the function calls in its iOS apps, and Apple has a policy that forbids any non-Apple apps in its App Store from using private APIs. Shortly after that, the story developed further when Qihoo received a "double blow" [20] from Apple, which banned every single Qihoo app from iOS app store after an official warning. The reason for this double blow was rather new: Qihoo released their "enterprise" apps to the public but Apple restricts the enterprise apps to be used by employees of that company only, instead of everyone in public.

So what are private APIs and enterprise apps? Why Apple is so vigilant on their usages? In fact, iOS apps on any non-jailbroken device can be classified into two categories based on how they are being distributed. Apps which are distributed through the Apple's App Store are the most common to users and we call them as "normal" apps. The other cateory is the "enterprise" or "developer" apps which are distributed under the *enterprise/developer certificates*. Note that these "enterprise" and "developer" apps are not distributed via the App Store, hence, they are not regulated by the Apple's review process [2]. Often, iOS apps use Apple's Application Programming Interfaces (APIs) to interact with the iOS system for system resources and services. Some of these iOS APIs are "private" in the sense that only apps developed by Apple (Apple apps) can use them and other apps using these private APIs are not allowed on the Apple App Store. These apps using private APIs can only be distributed using the "enterprise" or "developer" certificates which grant a limited number of developers or employee users per enterprise (or company). Public APIs are often used to interact with iOS system for resource allocation or service request. Private APIs, however, are much more powerful comparing to the public ones. To illustrate, consider an iOS 6.0 device. A normal app can call the public Twitter APIs to post a tweet on the user's Twitter homepage (Fig. 1), and the user must consent by clicking the "Post" button. However, by using private APIs, the app can post the tweet without notifying the user [35] at all.

Given that private APIs have powerful functionalities, once abused, private APIs may become formidable weapons for malicious attackers. Although they are undocumented and Apple's review process scrutinizes apps rigorously to search

ASIA CCS'15, April 14–17, 2015, Singapore..

http://dx.doi.org/10.1145/2714576.2714593 .

Figure 1: Screenshot for Legitimate Twitter App

for private APIs, hackers still manage to learn them by analyzing iOS frameworks directly and found sophisticated means [35, 27] which can bypass the review process. Since there is no review process for apps under enterprise/developer certificates, therefore, attackers may take advantage of this venue to distribute malware through the web. In addition, authors in [34] and [29] show that infecting a large number of iOS devices through botnets is feasible. They have demonstrated that a compromised computer can be instructed to install enterprise apps or developer apps on iOS devices. This shows that there are number of ways to infect iOS devices, and App Store's review process is not sufficient to protect iOS devices.

Although Apple only allows the enterprise apps and developer apps to be used by company employees and developers, many vendors do use this method to distribute their apps to the public (e.g., GBA emulator [11]). In addition, hackers do use this channel to distribute malware [36]. It seems that Apple does not have an ideal method to monitor and manage these wild enterprise and developer apps. Although Apple can remotely revoke the enterprise or developer certificates, hackers can resign the app, rename the app's Bundle name or change the system time (e.g., Pangu [19] uses an invalid enterprise certificate for jailbreak) to bypass the revoking process. In addition, if the enterprise apps contain vulnerabilities (e.g., URL scheme flaw and heart bleed), they become valuable targets to attackers as the private APIs that they use make them more powerful as compared with ordinary apps from iOS apps. As more users and enterprises use iOS devices, there is a constant push to create enpublic apps, this is exactly the motivation of this paper because we need to warm the developers (and users) before we see the epidemic spread of this form of malware.

When enterprise apps or developer apps are distributed to the public, we refer to such apps as "enpublic" apps. In order to understand the security landscape of enpublic apps, we propose a security evaluation mechanism for iOS enpublic apps. Researchers and anti-virus companies can use our mechanism to detect and analyze malicious iOS enpublic apps. Customers can also use this mechanism to evaluate the risk of the enpublic apps before installation.

The main contributions of the paper are:

- We present a detailed study of threats using private APIs within enpublic apps, and show the gap between Apple's regulations and the abuse of enterprise and developer certifications. To the best of our knowledge, this paper is the first to investigate the abuse of enterprise and developer certificates.
- We propose both static and dynamic analysis techniques to detect iOS private APIs, URL schemes vulnerability and sensitive information leakage. We summarized 25 private APIs which are crucial and security sensitive on iOS 6/7/8, and we have filed one CVE (Common Vulnerabilities and Exposures) for iOS devices.
- In our evaluation, we discovered that 844 out of the 1408 enpublic apps we studied do use private APIs. 14 apps contain URL scheme vulnerabilities and 901 apps transport sensitive information through unencrypted channel.

2. BACKGROUND

In this section, we briefly provide some background information about iOS developer programs and itms-services installation.

2.1 iOS Developer Programs

iOS developers use development tools like Xcode and iOS simulators to develop apps. To distribute their apps to legal (or non-jailbroken) iOS devices, app developers must join the iOS developer programs[6]. There are three types of iOS developer programs: *standard program*, *enterprise program* and *university program*. Tab. 1 depicts the distribution capability of these three programs. Note that all three programs allow developers to test their apps on devices. They differ in how developers can distribute their apps. Developers under the standard program can release their apps on the App Store or distribute the apps through the ad hoc channel. For the enterprise program, developers can distribute their apps through the ad hoc channel and the in-house channel (we will explain the ad hoc and in-house distribution channels in the next subsection). Apps developed under the university program cannot be distributed in any of these three channels.

Although Apple only allows the enterprise apps and developer apps to be used by company employees and developers, many vendors or malware writers do use this method to distribute their apps or malware to the public [36]. The advantage of using enterprise certificate and developer certificate is that developers can use private APIs to achieve advanced functionalities (e.g., screen recording). In addition, instead of the App Store, developers can maintain the installed apps on their own servers, so that they can update these apps any time without the review process. Furthermore, Apple banned many kinds of apps (e.g., pornographic app) on App Store. By using enterprise certificate or developer certificate, it becomes possible to distribute these kinds of apps.

2.2 Ad Hoc and In-House Distribution

2.2.1 *Ad hoc distribution*

Ad hoc is a Latin phrase which means "for this". Both the standard program and the enterprise program can use the ad hoc channel to distribute iOS apps. These apps are `.ipa` files which contain app runtime resources and an ad hoc provisioning profile. Fig. 2 shows the architecture of the ad hoc

Program	Test apps on iOS devices	App Store	Ad Hoc	In–house
Standard Program	Yes	Yes	Yes	No
Enterprise Program	Yes	No	Yes	Yes
University Program	Yes	No	No	No

Table 1: iOS Developer Programs and their Distribution Capability

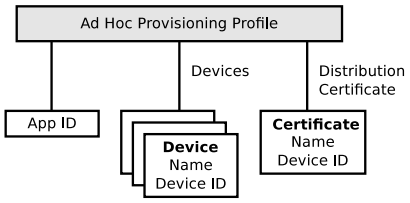

Figure 2: Ad Hoc Provisioning Profile

provisioning profile, each profile contains the ID of the tested app, certificate of the developer, and the unique device IDs (UDIDs) of designated devices. The provisioning profile only allows the iOS app to be installed on these designated devices. In order to register designated devices, developers need to specify the UDIDs of designated devices and register them in the iOS Dev Center [14] before distribution. Note that each development account can register at most 100 devices per membership year. After the registration, developers or testers can install the ad hoc `.ipa` file on the designated devices through the iTunes or via the iOS RPC communication library (e.g., libimobiledevice [17]). Therefore, hackers may register more than one developer account for enpublic app distribution. The advantage of using this method is Apple cannot easily revoke all the certificates.

2.2.2 *In-house distribution*

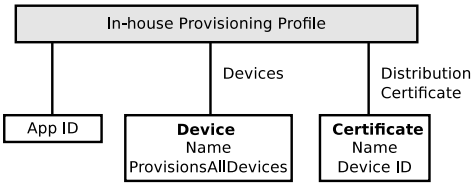

Figure 3: In–house Provisioning Profile

Enterprise program can also use the in-house channel to distribute iOS apps on iOS devices. The enterprise apps contain app runtime resources and an in-house provisioning profile. Fig. 3 shows the architecture of the in-house provisioning profile. The major difference between an ad hoc profile and an in-house profile is that an ad hoc profile serves no more than 100 designated devices, whereas in-house profile may serve *unlimited number* of devices. Besides using the iTunes for distribution, the in-house distribution can use the itms-service as another distribution channel. This service supports an over-the-air installation of custom-developed in-house apps without using the iTunes or the App Store. All it needs is just a web server, an iOS app in `.ipa` format (built for release/production with an enterprise provisioning profile), and an XML manifest file which instructs the device to download and install the apps. In fact, developers can provide a simple web link which triggers the installation process, for instance:

```
<a href=‘‘itms-services://?action=downloadmanifest&
url=http://www.example.com/manifest.plist’’>
Install App</a>
```

During the installation, the device contacts “ocsp.apple.com” to check the status of the distribution certificate used to sign the provisioning profile. In addition, once it is issued, distribution provisioning profiles have 12 months validity before expiration. After that, the profile will be removed and the app cannot be launched. Fig. 4 illustrates the whole installation process. When a user clicks the “Install” button, the iOS system will automatically install the enterprise app and profile on the device.

3. SYSTEM DESIGN AND METHODOLOGY

To detect and analyze enpublic apps, we designed iAnalytics, a security evaluation system for iOS enterprise/developer apps. iAnalytics first collects enpublic apps from the Internet. Then the system can perform private API detection and app vulnerability detection for the downloaded enpublic apps. At the high level, the workflow for iAnalytics is as follows:

- iAnalytics has an `.ipa` crawler for enpublic apps. Because enpublic apps use “itms–service” for installation, the `.ipa` crawler will also search the Internet with the related keyword, i.e., “`itms-services://?action=downloadmanifest`”. When finding a suspicious “itms–service” link, the `.ipa` crawler will parse the `.plist` file and crawl the related enpublic app from the HTTP or HTTPS sever.
- After collecting `.ipa` files, iAnalytics uses two detectors for security evaluation: one for analyzing the private API usage, and the other for app vulnerability analysis.
- For private API detection, iAnalytics first generates a private APIs list from iOS SDK. Then iAnalytics uses a private API detector to detect private API calls according to the private APIs list. Note that because there is no app review process for enpublic apps, the app can use any dynamic loading techniques (e.g., `NSBundle and dlopen()`) to load the private frameworks at runtime. Therefore, in addition to checking the static links of frameworks, iAnalytics also analyzes the dynamic loading behaviors at runtime.
- For app vulnerability detection, iAnalytics uses an app vulnerability detector which focuses on detecting the URL scheme vulnerability and sensitive information

Figure 4: The Process of Itms–service Installation

leakage. URL scheme is a protocol for websites or apps to communicate with other apps. An URL scheme with a faulty logic design may cause serious consequences. For sensitive information leakage, personal information, when transported or stored without being encrypted, can be sniffed or extracted by attackers.

- Finally, iAnalytics summarizes the findings and generates analysis reports of these enpublic apps to security analysts.

Fig. 5 depicts the workflow of our system.

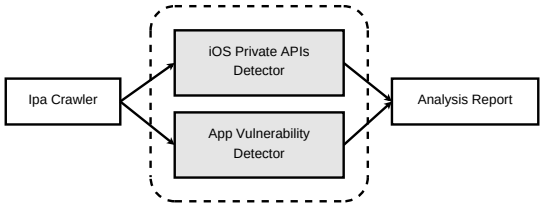

Figure 5: Workflow of iAnalytics

4. IOS PRIVATE APIS DETECTION

In this section, we first give a brief introduction to private APIs, then we show the methodology of obtaining the private APIs list. Finally, we present the implementation of iOS private APIs detector.

4.1 Introduction to Private APIs

iOS Private APIs are undocumented application programming interfaces of the iOS frameworks and they are prohibited by the App Store's review process. These private APIs are methods of framework classes which have no declaration in class header files. Apple emphasized that these private APIs should only be used by the class internally or the iOS system apps [2]. However, when third-party apps use these private APIs, they have enhanced capabilities (e.g., posting tweets, sending emails, and sending SMS without user's consent on iOS [35].) In addition, because there is no application review process in iOS enterprise/developer apps, anyone can distribute their private API apps through this channel.

4.2 Private APIs list

Since the private APIs are the methods of frameworks which have no declaration in the class header files, one can use the following methodology to extract them:

- We first download and install the specific iOS SDK (e.g., iOS 7.1) from Apple's development website[13].
- Then we extract all the frameworks from iOS SDK, for example, UIKit framework can be found at `/Applications/Xcode.app/Contents/Developer/ Platforms/iPhoneSimulator.platform/Developer/ SDKs/iPhoneSimulator7.1.sdk/ System/Library/Frameworks/ UIKit.framework/UIKit`. All of the iOS frameworks are Mach–O file format [18]. By using class–dump[7], which is a command–line tool for extracting the Objective-C information stored in Mach-O files, one can retrieve all the method names and parameters from these Mach–O files. Note that these methods are the API calls, which include both public and private API calls in iOS frameworks.
- Based on Apple's official documents[1], we obtain all the public API calls of a specific SDK version.
- Once we have obtained all private and public APIs call set from Step 2 and public API calls set from Step 3, we can extract the private API calls set of the specific SDK version.

4.3 iOS Private APIs Detector

For iOS private API detection, we use both static and dynamic methodologies to analyze the iOS enpublic apps.

4.3.1 Static Analysis

We built a static analysis tool on top of CCTool[5]. CCTool is an open source project of Apple for analyzing Mach-O format files. This project contains several tools such as otool, nm and strings. However, for private API detection, these tools are not compact and readable. They can only retrieve the basic information of the Mach-O format file

and do not have the logic of detecting private APIs. Therefore, we enhance CC-Tool by adding several new features (e.g., private API detection and dynamic loading behavior detection), then build our static analysis of iOS private API detector on top of these modules.

The static analysis process works in the following steps:

- iOS Private APIs Detector can analyze the framework/library loading and determine which framework/library contains private API calls.
- For suspicious framework/library, iOS private API detector will analyze the method symbols and then compare with the private APIs list. If the detector finds private API calls, it will generate a report for the user.
- If the detector finds dynamic loading behavior (e.g., methods which call `NSBundle` and `dlopen()`) of the app, we use the dynamic analysis of iOS private API Detector.

For example, an enterprise app, "`com.tongbu.tui`", statically loads several sensitive frameworks:

```
/System/Library/PrivateFrameworks/
SpringBoardServices.framework

/System/Library/PrivateFrameworks/
MobileInstallation.framework

/System/Library/Frameworks/IOKit.framework
...
```

`SpringBoardServices` and `MobileInstallation` are private frameworks which contain spring board information and installed package information. As long as an app uses APIs in these two frameworks, they are private APIs. Although `IOKit` framework does not belong to the private frameworks, there is no public API calls in this framework. Therefore, any API calls in this framework also belong to private APIs.

Another example is "`net.qihoo.360msafeenterprise`". This app does not load any private frameworks. However, some public frameworks also have private APIs. For instance: "`net.qihoo.360msafeenterprise`" uses `_CTTelephonyCenterAddObserver` API call of `/System/Library/Frameworks/CoreTelephony.framework` to monitor the incoming telephone calls and SMS messages.

4.3.2 Dynamic Analysis

Because there is no app review process, enpublic apps can use any dynamic loading techniques at runtime. For example, `NSBundle` is a dynamic loading class in Objective-C. It is similar to Java reflection [15] which can dynamically load other frameworks. Another dynamic loading technique is to use private C functions, such as `dlopen()` and `dlsym()`, to dynamically load and execute methods of iOS frameworks. For these apps with dynamic loading behavior, we use `Cydia Substrate`[9] (a hooking framework) to hook the low level library call `_dlopen()` method and `_dlsym()` method on an jailbroken iOS device. Because the low level implementation of `NSBundle` also uses `_dlopen()` method and `_dlsym()` method, there is no need to hook the related methods of `NSBundle`. To illustrate, the logic of `_dlopen()` hooking is shown below:

```
//declare the origional funciton of dlopen
void * (*_dlopenHook)(const char * __path, int __mode);

//implement the replaced dlopen function
void * $dlopenHook(const char * __path, int __mode)
{
NSLog(@"iAnalytics:␣Loading␣%s␣framework", __path);
return _dlopenHook(__path,__mode); //call orig
}
//do hooking
%ctor {
MSHookFunction((void *)MSFindSymbol(NULL,"_dlopen"),
(void *)$dlopenHook, (void **)&_dlopenHook);
}
```

We first obtain a declaration of the original hooked function and implement the function used to replace the original one accordingly. We then use "`MSFindSymbol`" to obtain the address of hooked function and use "`MSHookFunction`" to replace the hooked function. In the implementation of replaced function, the app outputs the loading methods and frameworks to the system log. We collect the related log information from "`/var/log/syslog`". This way, the iAnalytics can detect invocations of private APIs.

5. APP VULNERABILITY DETECTION

Since enpublic apps can use private APIs, they are more powerful, and potentially more damaging, as compared with apps in App Store. In this situation, once plagued by vulnerabilities, enpublic apps become severe threats to iOS users. Attackers may leverage such vulnerabilities to craft dangerous attacks. In this section, we examine two vulnerabilities found in iOS enpublic apps and show how to detect them.

5.1 URL Scheme Vulnerability Detection

A URL scheme vulnerability arises when an app has an unsafe URL scheme logic design and the hacker could send malicious URL requests to the devices. A URL scheme vulnerability may cause memory corruption or malicious actions without user's authorization. In this part, we first briefly explain what URL scheme vulnerability is and present our URL scheme vulnerability detector.

5.1.1 Introduction to URL Scheme Vulnerability

By using URL schemes, web pages running in Safari or iOS apps can integrate with system apps or third–party apps. For example, the `tel://` scheme can be used to launch the telephone app. For instance, a website may contain the following HTML script and a user browses it using Safari on iOS:

```
<iframe src="tel://123456789"></iframe>
```

When fed with the URL scheme of telephone app, Safari will launch the telephone app with the "123456789" as the parameter. After launching, the telephone app requests for user's authorization to make a call. This is a correct logic design from a security perspective, because a malicious website should not be able to initiate a phone call without notifying the user. However, if the app does not perform the authorization check for the URL scheme, it causes security problems. For example, [25] reported that if the website contains the following HTML script:

```
<iframe src= "skype://123456789?call"> </iframe>
```

The older version of Skype will automatically start a call when the user browses a malicious website using the Safari browser. Therefore, developers need to pay attention to the logic design of URL scheme. Otherwise, attackers may use the vulnerability of URL scheme to implement attacks.

5.1.2 Detection Methodology

We describe our methodology for URL Scheme vulnerability detection in the paragraphs below:

- We parse `Plist` to obtain the URL scheme information from the `Info.plist` file of the `.ipa` package. In the `Info.plist` file, the URL scheme information is stored in the `CFBundleURLTypes` array. By parsing this array, we can extract `CFBundleURLName` and `CFBundleURLSchemes`.
- If an iOS app has the `CFBundleURLSchemes`, it will call the `[application:handleOpenURL]` method in the `application delegate` class. We disassemble the app and then perform a recursive search from the `[application:handleOpenURL]` method. If the app invokes any private API calls, we catch it. Then security analysts can analyze this potential vulnerable app.
- iAnalytics performs fuzzy testing for iOS apps with URL schemes. After obtaining the `CFBundleURLSchemes` from `Info.plist` file, our system generates random strings or customized fuzz templates with corresponding URL schemes. Then we test these URL schemes through `libimobiledevice` on an iOS device with the installed apps. We then record the system log (e.g., crash report) for analysts.

We have performed the analysis above and discovered several apps with URL Scheme vulnerabilities. We will further describe our findings in Sec. 6.

5.2 Sensitive Information Leakage Detection

Apps may transmit or store personal identifiable information like device ID, user name, password, phone number, address and location information to third party companies. It is very dangerous that the sensitive information is transformed or stored by plain text, because attackers can easily obtain it by sniffing the network or through USB connection. In addition, because enpublic apps can obtain more sensitive personal data by using private APIs, they are more dangerous than normal apps. For personal information leakage detection, iAnalytics focuses on HTTP and local data storage.

5.2.1 Sensitive Information Leakage on HTTP

For analyzing HTTP packets, we set the tested device's HTTP proxy to our system. Then iAnalytics uses `libimobiledevice` library to launch the tested apps and sends random user events to the tested apps. If the tested app generates HTTP connection, the HTTP packets will be sent to our system for analysis. After getting the data, iAnalytics searches string patterns of system information (e.g., UDID, MAC address, IMEI and IMSI) and personal information (e.g., telephone number, location information and password) from the HTTP packets. If the system finds anything related to sensitive information, it will report it to analysts.

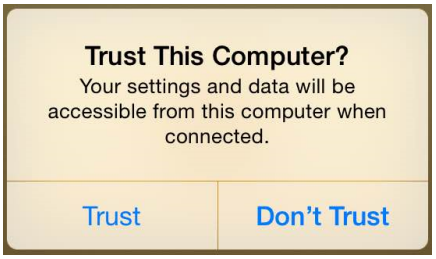

Figure 6: USB connection verification

5.2.2 Sensitive Information Leakage on Local Data Storage

iOS RPC communications library (e.g., libimobiledevice) can be used to obtain apps' data through an USB connection. Although apps in iOS are seperated by sandbox mechinism, the mechinism has no effect for iOS RPC communications library. An PC based application can extract all of apps' data without any permission and user's knowledge. Using iAnalytics, we discovered a bug in the iOS RPC communications: in iOS 7.0, Apple added a new feature to prevent untrusted USB connection (see Fig. 6). However, it only has effect on iTunes software. Other applications can still use iOS RPC communications library to obtain the app data even if the user choose "Don't Trust". Therefore, our system uses `libimobiledevice` library to extract the apps' data from the iOS devices and then search for sensitive information from the apps' data. If the system finds anything related to sensitive information, it will generate a report to a security analyst.

6. EVALUATION

We crawled 1408 enpublic apps from the Internet. We obtained the "Development Region" by parsing apps' `Info.plist` files. The "Development Region" indicates the geographic location of the apps development. Most enpublic apps are from United States, China, England and France, as shown in Tab .2. In this section, we report the statistics of private API usage and app vulnerabilities found in these apps. We also provide case studies of representative enpublic apps.

Country	# of apps
United States	660
China	361
England	223
France	62
Others	102
All	1408

Table 2: Statistics of Development Region

6.1 Private API Statistics and Case Studies

Within the 1,408 enpublic apps we crawled, 844 (60%) use private APIs. For example, some enpublic apps use the private APIs of `CoreTelephony.framework` to monitor phone call and SMS messages. After obtaining the sensitve information, enpublic apps can send them to the hacker's sever in the background. Some iOS 3rd-party market (non-App s-

tore markets) apps use the private APIs of `MobileInstallation.framework` and `SpringBoardServices.framework` to manage and install 3rd-party iOS apps on iOS devices. In addition, enpublic apps are able to remotely install other apps to the devices without user's knowledge. Many enpublic apps use the private APIs of `Message.framework` to get the IMSI (International mobile Subscriber Identity) and IMEI (International Mobile Station Equipment Identity) of the device. Because these IDs are globally unique, after obtaining these IDs, hackers are able to link it to a real-world identity [33] so users' privacy will be compromised. Tab. 3 summarizes the private API usages list found in our crawled apps[1]. Note that we have found 22 new dangerous private APIs which are not mentioned in [35] and [27]. These APIs are crucial and security sensitive on iOS 6/7/8 devices. If users do not pay any attention to them, their personal information can be easily leaked to the hacker. To demonstrate how dangerous the private APIs are, we present several case studies below.

6.1.1 Phone Call & SMS Message Monitoring and Blocking

As mentioned in Sec. 1, Qihoo released their "enterprise" apps to the public. Apple halted Qihoo by revoking the "enterprise" certificates [3]. Qihoo implemented some functions that rely on private APIs, which is prohibited from distribution on App Store. Note that one can still find Qihoo's "enterprise" app from the Internet today. Although the certificate became invalid, we can resign the app with our own certificate, then install it on the iOS devices and analyze it.

Qihoo's enterprise app is called "MobileSafeEnterprise" and the package name is "net.qihoo.360msafeenterprise". This app uses both static and dynamic approaches to invoke private APIs. First, "MobileSafeEnterprise" can monitor and block incoming phone calls and SMS messages. The methodology is using the private API, `CTTelephonyCenterAddObserver()` method of `coreTelephony.framework` to register a call back method of the `TelephonyCenter`. When the phone receives a phone call, "MobileSafeEnterprise" will get a "kCTCallIdentificationChangeNotification" call back. In this call back, "MobileSafeEnterprise" uses `CTCallCopyAddress()` method to obtain the telephone number of the caller and then use `CTCallDisconnect()` method to hang up the phone call. For incoming SMS messages, "MobileSafeEnterprise" will receive a "kCTMessageReceivedNotification" call back. In this call back, "MobileSafeEnterprise" uses `[[CTMessageCenter sharedMessageCenter] incoming MessageWithId: result]` method to obtain the sender's telephone number and the text of SMS messages.

Note that some private API invocations in "MobileSafeEnterprise" used dynamic loading techniques and our system detects such behaviors successfully. We also find that even in the latest iOS 7.1, developers or attackers can still use private APIs to monitor incoming phone calls and SMS messages. We have notified Apple about this security problem and Apple Inc. has been working on a fix which is to be released in future updates.

6.1.2 Collecting the Information of Installed Apps

For the network traffic monitoring, "MobileSafeEnterprise" uses private APIs of `UIKit.framework` and `SpringBoardServices.framework` to get the names and bundle IDs of running apps. The methodology is using `SBSSpringBoardServerPort()` method of `UIKit.framework` to get the server port of the `SpringBoard`. Then it uses `SBSCopyApplicationDisplayIdentifiers()` method of `SpringBoardServices.framework` to get the array of current running app bundle IDs. By using this private API, the app can record the running time and frequency of other apps. In addition, by analyzing the SDK of iOS 7.X and 8.X, we have found another private API called `allApplications()` in the `MobileCoreServices.framework` . This private API can be used to get all the bundle IDs of installed apps which include both running and unstarted apps. After getting the installed app list, hackers can sell them to analytic or advertisement companies, or hackers can analyze and exploit vulnerabilities of these installed apps so to launch persistent attack on these devices.

6.1.3 User Events Monitoring and Controlling

We encountered another interesting enpublic app called "iControl". This app can monitor and control user events (e.g., touches on the screen, home button press and volume button press) in the background on iOS 5.X and 6.X. For monitoring, it uses the `GSEventRegisterEventCallBack()` method of `GraphicsServices.framework` to register call backs for system wide user events. After that, every user event will invoke this call back method. In this call back, the app can get the user event information by parsing the `IOHIDEventRef` structure. For controlling, the app first uses `SBFrontmostApplicationDisplayIdentifier()` method of `SpringBoardServices.framework` to get the app at the front. After that, the app uses `GSSendEvent()` method of `GraphicsServices.framework` to send the user events to the front app. Attackers can use such mechanisms for malicious purposes. For example, potential attackers can use such information to reconstruct every character the victim inputs (e.g., stealing user's password). In addition, attackers can use this app as a trojan to control the victim's non-jailbroken phone from remote.

Fortunately, Apple fixed these monitoring and controlling private APIs on iOS 7.0 by adding permission controls. However, after analyzing the private APIs of `IOKit.framework` , we have discovered another call back registration method of user events: `IOHIDEventSystemClientRegisterEventCallback()`. After registration, the app can still monitoring the system wide user events in the background on iOS 7.0. We have reported this issue to Apple. Apple applied a CVE (Common Vulnerabilities and Exposures) for us [8] and fixed this flaw on iOS 7.1.

6.1.4 iOS 3rd-party App Installation and 3rd-party Markets

It is different from Cydia which targets only jailbroken devices, we have found several 3rd-party app markets (e.g., SearchForApp and Rabbit Assistant) which target both non-jailbroken and jailbroken iOS devices. These markets are enpublic apps and these apps use similar techniques for app management and installation. For app management, these apps use `MobileInstallationLookup()` method of `MobileInstallation.framework` to get the pList information of installed apps from the iOS system. After that, they use `SBSCopyLocalizedApplicationNameForDisplayIdentifier` method and `SBSCopyIconImagePNGDataForDisplayIdentifier` method

[1]Starting May 1, 2013, the App Store will no longer accept new apps or app updates that access UDIDs. [22]

of `SpringBoardServices.framework` to get the app names and icons. For app installation, it is interesting that these market apps have several ways to install 3rd-party apps on both jailbroken and non-jailbroken iOS devices. For jailbroken iOS devices, since there is no app signature verification, they use `MobileInstallationInstall` method and `MobileInstallationUninstall` method of `MobileInstallation.framework` to install and uninstall other 3rd-party apps. For non-jailbroken iOS devices, these market apps use "`itms-services`" (mentioned in Sec. 2) to install other enpublic apps or pirated apps. Because some companies release their apps in enterprise/developer version for beta testing, these 3rd-party app markets collected them and release them on their markets. Although customers can get the apps for free on non-jailbroken devices, they are taking high security risks (e.g., personal information leakage, remotely monitoring and control) of using these apps. For more discussion about pirated apps, please refer to Sec. 7.

6.1.5 *Phishing Attack and Unlimited Background Running*

We have found another interesting enpublic app called "Buddy Pro". This is an app which can get the location of incoming phone calls. It has several interesting features: First, it can run in the background forever. Second, it can auto start after system rebooting. Thirdly, it can pop up a dialog on the incoming call screen to show the location information. These features are dangerous in that a hacker can use them to launch a phishing attack on the user. Note that running app in the background is very important for phishing apps, because the app needs to stay alive for monitoring sensitive information (e.g., keystrokes or finger movement). However, in iOS, all running tasks are under a strict time limit (around 600 seconds for iOS 6 and 180 seconds for iOS 7/8) for processing.

By analyzing the app using our system, we find two ways for the app to run in the background forever and one way for the app to auto start after rebooting:

- The app can use `AVAudioPlayer` to play a silent music in the background and uses a property called `AudioSessionProperty_OverrideCategoryMixWithOthers`. By using this property, the silent music can be played with other music without being detected. In addition, the app will not appear on the music panel. Hence, the app can run continuously without being detected.
- The app can use two undocumented UIBackgroundModes, `Continuous` and `unboundedTaskCompletion`, to run in the system background forever. Generally speaking these two undocumented UIBackgroundModes can only be used by system apps, so third-party apps cannot bypass the App Store review if they use undocumented UIBackgroundModes. However, authors in [35, 27] showed that it is possible to bypass the review process and it would be dangerous that an app can monitor phone call events in the background forver.
- The app can use a UIBackgroundMode called `VOIP` (Voice -Over -Internet Protocol) to start automatically after the system rebooting. Because an app VOIP needs to maintain a persistent network connection so that it can receive incoming calls and other relevant data. In order to ensure that the VOIP services are always available, the iOS system will relaunch the VOIP app in the background immediately after system boot. Therefore, unless the customer terminates the app manually, the app can always monitor phone call events even after rebooting the device.

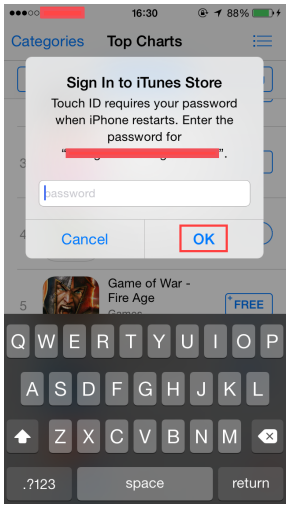

Figure 7: Real AppStore Login Dialog

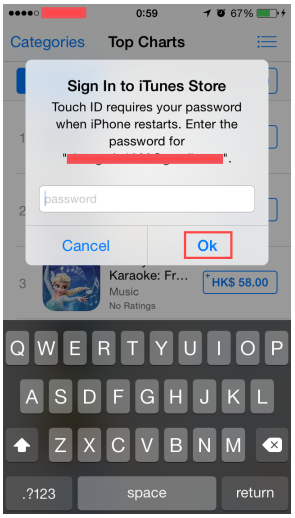

Figure 8: Fake AppStore Login Dialog

In "Buddy Pro" app, we have found two dangerous private API calls : `CFUserNotificationCreate()` and `CFUserNotificationReceiveResponse()` of `CoreFoundation.framework` . These two private APIs can create and pop up interactive dialogs on the foremost screen and then receive user's response. Unlike `UIAlertView` class, the app can only pop up dialogs on its own screen. Using `CFUserNotificationCreate()` and `CFUserNotificationReceiveResponse()`, one app can always pop up dialogs to the foremost screen (includes other apps' screen and home screen) in the background. This is very dangerous, because hackers can easily build a "phishing" app to steal users' accounts. In order to demonstrate, we built an example "phishing" app. This "phishing" app

can pop up a fake login dialog (to differentiate, we changed the letter "K" to lower case) in the AppStore app (Fig. 7 and Fig. 8). In addition, we built such a "phishing" app and disguised it as the Twitter app. If the user opens this "phishing" app, it will run in the background forever and it will auto start after rebooting. In addition, it will monitor the runtime process information of the system. What's more, by using the "CVE-2014-4423" vulnerability [23], the "phishing" could get information about the currently-active iCloud account, including the name of the account. When the user opens the AppStore app, it will pop up a fake login dialog on the foremost screen. After the user enters his password, the "phishing" app will send the user's password to the remote server. Note that the demo used the 7.1 version of iOS system on a non-jailbroken iPhone 5s device.

6.2 App Vulnerabilities Statistics and Case Studies

Within the 1408 enpublic apps, we have found 14 (1%) apps containing URL scheme vulnerabilities. Most of the scheme vulnerabilities crash the corresponding app. However, there are two interesting cases, "PandaSpace" and "iDevice Tool" that may become targets for exploits. For sensitive information leakage, 901 (64%) enpublic apps transform sensitive information through unencrypted HTTP or store the information in plain text on the phone. Tab. 4 shows the statistics of UDID, MAC address, IMEI, IMSI, telephone number, GPS (Global Position System), installed app list and password leakage through HTTP and local data storage. After obtaining the personal information, hackers can sell them to analytic or advertisement companies. In addition, they can implement advanced persistent attack after getting the installed app list or password.

Unencrypted Data	# of apps	% of apps
UDID	453	50.3%
MAC address	183	20.3%
IMEI	97	10.8%
IMSI	28	3.1%
Telephone number	86	9.5%
GPS	52	5.8%
Installed app list	54	6.0%
Password	3	0.3%

Table 4: Statistics of Unencrypted Data Leakage

6.2.1 Remotely Install 3rd-party Apps

"PandaSpace" is a 3rd-party market app. As we mentioned before, these 3rd-party markets are enpublic apps which can manage and install other 3rd-party apps. We found that attackers can remotely exploit the URL scheme vulnerability of "PandaSpace" to install any 3rd-party apps on jailbroken iOS devices. By parsing the `.plist` file, we found "PandaSpace" contains URL schemes. We recursively scan the private API calls in the `handleOpenURL()` method of the app. We found a dangerous API call, `MobileInstallationInstall()`, in the handle URL scheme method. After that, we analyzed the `handleOpenURL()` function of "PandaSpace" manually. We found that when "PandaSpace" receives a valid `.ipa` file download URL, it will download the `.ipa` file from this URL and then install it on the device. Attackers may use this URL scheme to install other apps on victims' iOS devices. For example, attackers can create a malicious website with the following HTML script:

```
<iframe src="PandaSpace://http://bbx.sj.***.com
/iphonesoft/detail.aspx?action=show&f_id=%@">
</iframe>
```

When victims visit this malicious webiste using Safari, "PandaSpace" will download the app and install it on the phone. Although using the URL scheme vulnerability of "PandaSpace" can not directly install 3rd-party apps on a non-jailbroken iOS devices, attackers can use this vulnerability to trick users to install other enpublic apps.

6.2.2 System Log Leakage

"iDevice Tool" is an enpublic app used by developers to debug iOS systems. It uses several private APIs to gain system information (e.g., UDID, mac address and system logs). We found that hackers can use the URL scheme vulnerability of "iDevice Tool" to email all the system information to a specific email address. For example, when the user visits a malicious website with the following content:

```
 <iframe src= "idevicetool://emailLog?email=example
@example.com&filter=TextFilter&appname=AppName
"> </iframe>
```

"iDevice Tool" will send the system log of "AppName" to the email address, "example@example.com".

6.2.3 Location Information Leakage

"Sina weather forecast" is a weather report app. It has both App Store version and enterprise version. In the enterprise version, it uses GPS (Global Positioning System) to get the location of the user and then uses private API to get the system information (e.g., UDID) of the device. After that, it sends these sensitive information to its server in plain text. For example, here is the captured HTTP request:

```
POST http://forecast.sina.cn/app/update.php?device
=iPhone&uid=32aee18d40fcb4c24e5988c76b4e9f0
d84fb8***&os=ios6.1.2&city=CHXX0***&pver=3.249
&pt=2&token=ff0007a31c005916d5e45b360481f1
1e05b40f010405731e94c8&pid=free
```

This HTTP leaks the UDID ("32aee18d40fcb4c24e5988c-76b4e9f0d84fb8***"), iOS version ("6.1.2") and location information (the city number, "CHXX0***"). If the server's information is leaked (e.g., through heart bleed attack), attackers can easily get the user's location through the UDID.

6.2.4 Username and Password Leakage

"eHi Taxi" is an iOS app providing services to call a taxi. It provides an App Store version and also an enterprise version. However, both versions have the same vulnerability that they transfer the user's phone number, name and password in plain text through HTTP, as shown in the captured HTTP post:

```
POST: http://myehilogin.1hai.cn/Customer/Login/Log-
inMobile
TextView: LoginName=5109318***&LoginPassword=pass***
```

Hence, attackers can easily sniff and harvest these private information. After getting user's account, hackers can call taxis using victim's money. In addition, the victim may use the same account for other apps. Therefore, hackers may launch advanced persistent attack to these devices. Note that this vulnerability is not specific to enpublic apps.

7. DISCUSSION

In this section, we first discuss about pirated apps on iOS. Then we discuss the limitation of our system and possible improvements.

7.1 Pirated Apps on Non–jailbroken iOS Devices

As we mentioned in Sec. 2, iOS apps have four distribution ways: test apps on iOS devices, App Store, ad hoc and in–house channels. The first distribution channel is for debugging iOS apps only, developers need to use XCode and USB connection to install iOS apps. Customers usually install iOS apps from the second channel, the Apple Store. Given that Apple has the review process for apps on App Store, we assume apps on Apple Store are genuine ones. However, pirated apps can use ad hoc and in–house channels to distribute to non–jailbroken iOS devices. For example, a malware developer can extract a paid iOS app from a jailbroken iOS device and package it into an `.ipa` file. He or she can then use the developer key from iOS developer program or iOS enterprise program to resign the `.ipa` file. After that customers may install the pirated apps through iTunes or itms–services without paying for them. In addition, because hackers have already reverse engineered the iTunes protocol, so that they can install the `.ipa` files on non–jailbroken iOS devices without iTunes [17] [16]. Although ad hoc distribution has a 100 device install limit, hackers can rename the Bundle name of the app, iTunes will then treat the renamed app as a new app with another 100 device limitation. Therefore, hackers can use this method to attract customers to download free pirated apps which are not free in Apple Store from their third–party markets, and then use advertisement to gain money. In this case, both iOS developers for the original app and Apple become victims.

7.2 Limitation and Future Work

Because enterprise iOS apps observe no regulation on using dynamic loading techniques, it is difficult for traditional static analysis methodologies to get the payload behavior before execution. Therefore, our system uses dynamic analysis to handle the dynamic loading behavior. However, dynamic analysis may not have a good code coverage since an app may have hundreds or thousands execution paths. Dynamic analysis can only explore a single execution path at a time and it is hard for the dynamic analysis system to trigger the expected result without the right input. Currently, we use simple behavior trigger techniques (e.g., launching the app, making a telephone call and locking the screen), so iAnalytics cannot guarantee complete code coverage of all dynamic loading behaviors. A possible improvement is to perform symbolic execution to compute all of the feasible paths and we are considering this in our future work.

For unencrypted sensitive information leakage, we only focus on the HTTP protocol and plain text data. However, apps may use other protocols to transfer unencrypted data or simple encrypted data through network sockets. In addition, iOS system has several internal communication techniques (e.g., shared keychain access and custom URL scheme). If the app does not encrypt the data or it has a faulty logic design, it is possible for other apps to sniff or hijack the exchanged data. For capturing sockets and the data of internal communication, the system needs to provide hooking service on the related methods, triggering the behavior, having simple decryption engine and determining the transferred data structure. We plan to address such extension in our future work.

8. RELATED WORK

There have been number of works which aim to bypass the code signing and app review process of Apple App Store. A common method is to Jailbreak [31][10][28] the system so to obtain the root privilege and permanently disable the code signing mechanism. In addition, hackers distribute iOS malware on jailbroken devices [32]. Although jailbreaking is feasible [26], it is getting more and more difficult because Apple actively fixes known vulnerabilities and iOS becomes more mature. Despite the increasing difficulty of exploiting iOS, our findings show that it is possible to distribute apps without jailbreaking iOS. However, it is worth noting that our approach can still take advantage of the vulnerabilities utilized by other jailbreaking methods to compromise iOS.

C. Miller [30] discovered that iOS code signing mechanism could be bypassed, allowing attackers to allocate a writeable and executable memory buffer. A malicious app can exploit this vulnerability to generate and execute attack code at runtime. However, Apple has fixed this issue and blocked apps using such kind of methods to dynamically load and launch malicious payload. T.Wang [35] puts forward another novel approach to evade the app review process by making the apps remotely exploitable and introduing malicious control flows by rearranging signed code. Because these control flows are dynamically generated when the attackers try to exploit those apps, Apple will not discover them during the app review process. However, this vulnerability has been fixed by Apple as well. Authors in [34] and [29] show that infecting a large number of iOS devices through botnets is possible. By exploiting the design flaws of iTunes and the device provisioning process, they demonstrate that a compromised computer can be instructed to install "enpublic" apps on iOS devices. These works show that iOS devices can be infected and App Store's review process is not adequate for protecting iOS devices. Our work further shows that iOS vulnerabilities exist.

The work closest to ours is by J. Han etc. [27]. They propose to launch attacks on non-jailbroken iOS devices from third-party application by exploiting private APIs. Compared with their work, our work is more systematic and we show there is another channel to distribute malicious apps except the App Store. Our focus is not just to exploit private APIs, but also illustrate other vulnerabilities. Moreover, we performed systematic analysis on large number of iOS apps, and in particular, we applied our analysis upon iOS 7/8, which has not been studied before.

9. CONCLUSION

In this paper, we present the security landscape of iOS enpublic apps and their usage of private APIs. In order to understand their security impact, we designed and im-

plemented a mechanism which evaluate the overall security status of enpublic apps by combining static semantic checks and runtime detection technologies. Our results show that 844 (60%) out of the 1408 enpublic apps do use private APIs. 14 (1%) apps contain URL scheme vulnerabilities, 901 (64%) enpublic apps transport sensitive information through unencrypted channel or store private information in plain text on the devices. In addition, we summarized 25 private APIs on iOS 6/7/8 which have security vulnerabilities and we have filed one CVE for iOS devices.

10. ACKNOWLEDGMENTS

We would like to thank our shepherd, Jin Han, and the anonymous reviewers for their valuable comments. We also thank Raymond Wei, Dawn Song, and Zheng Bu for their valuable help on writing this paper.

11. REFERENCES

[1] API Reference of iOS Frameworks, 2014. https://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Reference.

[2] App store review guidelines. https://developer.apple.com/appstore/resources/approval/guidelines.html.

[3] Apple Bans Qihoo Apps From iTunes App Store, February, 2012. http://www.techinasia.com/apple-bans-qihoo-apps/.

[4] Apple, Creating Jobs Through Innovation, 2012. http://www.apple.com/about/job-creation/.

[5] CCTool. http://www.opensource.apple.com/source/cctools.

[6] Choosing an iOS Developer Program, 2014. https://developer.apple.com/programs/start/ios/.

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

[8] CVE-2014-1276 IOKit HID Event, 2014. http://support.apple.com/en-us/HT202935.

[9] Cydia Substrate. http://www.cydiasubstrate.com.

[10] Evad3rs, evasi0n jailbreaking tool, 2013. http://evasi0n.com/.

[11] How Apple's Enterprise Distribution Program was abused to enable the installation of a GameBoy emulator, 2014. http://www.imore.com/how-gameboy-emulator-finding-its-way-non-jailbroken-devices.

[12] How Many Apps Are in the iPhone App Store. http://ipod.about.com/od/iphonesoftwareterms/qt/apps-in-app-store.htm.

[13] iOS Dev Center. https://developer.apple.com/devcenter/ios/index.action.

[14] iOS Dev Center, 2014. https://developer.apple.com/devcenter/ios/index.action.

[15] Java Reflection. http://docs.oracle.com/javase/tutorial/reflect/.

[16] Kuai Yong iOS device management, 2014. http://www.kuaiyong.com/eg_web/index.html.

[17] Libimobiledevice: A cross-platform software protocol library and tools to communicate with iOS devices natively, 2014. http://www.libimobiledevice.org/.

[18] OS X ABI Mach-O File Format Reference. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html.

[19] Pangu Jailbreak, 2014. http://pangu.io/.

[20] Qihoo Double Blow as iOS Apps Banned by Apple, China Warns of Anti-Competitive Practices, January, 2013. http://www.techinasia.com/qihoo-apps-banned-apple-app-store/.

[21] Tim Cook to shareholders: iPhone 5s/c outpace predecessors, Apple bought 23 companies in 16 months. http://appleinsider.com/articles/14/02/28/tim-cook-at-shareholder-meeting-iphone-5s-5c-outpace-predecessors-apple-bought-23-companies-in-16-months.

[22] Using Identifiers in Your Apps, 2013. https://developer.apple.com/news/index.php?id=3212013a.

[23] Vulnerability Summary for CVE-2014-4423, 2014. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-4423.

[24] When Malware Goes Mobile. http://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile/why-ios-is-safer-than-android.aspx.

[25] D. Chell. *iOS Application (In)Security*. 2012.

[26] D. Goldman. Jailbreaking iphone apps is now legal. *CNN Money. Retrieved*, pages 09–11, 2010.

[27] 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.

[28] Y. Jang, T. Wang, B. Lee, , and B. Lau. Exploiting unpatched ios vulnerabilities for fun and profit. In *Proceedings of the Black Hat USA Briefings, Las Vegas, NV, August 2014*.

[29] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal. Injecting malware into ios devices via malicious chargers. In *Proceedings of the Black Hat USA Briefings, Las Vegas, NV, August 2013*.

[30] C. Miller. Inside ios code signing. In *Proceedings of Symposium on SyScan*, 2011.

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

[32] F. A. Porter, F. Matthew, C. Erika, H. Steve, and W. David. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM SPSM*. ACM, 2011.

[33] E. Smith. iphone applications & privacy issues: An analysis of application transmission of iphone unique device identifiers (udids). 2010.

[34] W. Tielei, J. Yeongjin, C. Yizheng, C. Simon, L. Billy, and L. Wenke. On the feasibility of large-scale infections of ios devices. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 79–93. USENIX Association, 2014.

[35] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: when benign apps become evil. In *Presented as part of the 22nd USENIX Security Symposium*, pages 559–572, 2013.

[36] C. Xiao. Wirelurker: A new era in ios and os x malware. 2014.

Method	Framework	Usage	Available on iOS 6.X	Available on iOS 7.X	Available on iOS 8.0
[[UIDevice currentDevice] U–niqueIdentifier]	UIKit	Get the UDID of the device.	Yes	No	No
CTSIMSupportCopyMobile–SubscriberIdentity()	coreTelephony	Get the IMSI of the device.	Yes	No	No
CTSettingCopyMyPhoneN–umber()	coreTelephony	Get the telephone number of the device.	Yes	No	No
CTTelephonyCenterAddOb–server()	coreTelephony	Register call back of SMS mes–sages and incoming phone calls.	Yes	Yes	Yes
CTCallCopyAddress()	coreTelephony	Get the telephone number of the phone call.	Yes	Yes	Yes
CTCallDisconnect()	coreTelephony	Hang up the phone call.	Yes	No	No
[[CTMessageCenter shar–edMessageCenter] incom–ingMessageWithId: result]	coreTelephony	Get the text of the incoming SMS message.	Yes	Yes	Yes
[[NetworkController shared–Instance] IMEI]	Message	Get the IMEI of the device.	Yes	No	No
SBSCopyApplicationDispla–yIdentifiers()	SpringBoardServices	Get the array of current running app bundle IDs.	Yes	No	No
SBFrontmostApplicationDi–splayIdentifier()	SpringBoardServices	Get the front most app port.	Yes	No	No
SBSCopyLocalizedApplicati–onNameForDisplayIdentifier()	SpringBoardServices	Get the app name from the bundle ID.	Yes	Yes	Yes
SBSCopyIconImagePNG–DataForDisplayIdentifier()	SpringBoardServices	Get the app icon from the bundle ID.	Yes	Yes	Yes
SBSLaunchApplicationWit–hIdentifier()	SpringBoardServices	Launch the app using bundle ID.	Yes	No	No
MobileInstallationLookup()	MobileInstallation	Get the pList information of installed iOS apps.	Yes	Yes	Yes
MobileInstallationInstall()	MobileInstallation	Install `.ipa` file on jailbroken iOS devices.	Yes	Yes	Yes
MobileInstallationUninstall()	MobileInstallation	Uninstall app on jailbroken iOS devices.	Yes	Yes	Yes
GSEventRegisterEventCall–Back()	GraphicsServices	Register call back for system wide user events.	Yes	No	No
GSSendEvent()	GraphicsServices	Send user events to the app port.	Yes	No	No
IOHIDEventSystemClientR–egisterEventCallback()	IOKit	Register call back for system wide user events.	Yes	Yes	No
CFUserNotificationCreate()	CoreFoundation	Pop up dialogs to the foremost screen.	Yes	Yes	Yes
CFUserNotificationReceive–Response()	CoreFoundation	Receive the user input from the dialog.	Yes	Yes	Yes
allApplications()	MobileCoreServices	Get the bundle ID list of installed iOS apps.	No	Yes	Yes
publicURLSchemes()	MobileCoreServices	Get the URL schemes list of installed iOS apps.	Yes	Yes	Yes
continuous	UIBackgroundMode	Run in the background forever.	Yes	Yes	Yes
unboundedTaskCompletion	UIBackgroundMode	Run in the background forever.	Yes	Yes	Yes
VOIP (not a private API)	UIBackgroundMode	Auto start after rebooting.	Yes	Yes	Yes

Table 3: Statistics of Private API Usage