Launching Generic Attacks on iOS with Approved Third-Party Applications

Jin Han[1], Su Mon Kywe[2], Qiang Yan[2], Feng Bao[1], Robert Deng[2], Debin Gao[2], Yingjiu Li[2], and Jianying Zhou[1]

[1] Institute for Infocomm Research
[2] Singapore Management University

Abstract. iOS is Apple's mobile operating system, which is used on iPhone, iPad and iPod touch. Any third-party applications developed for iOS devices are required to go through Apple's application vetting process and appear on the official iTunes App Store upon approval. When an application is downloaded from the store and installed on an iOS device, it is given a limited set of privileges, which are enforced by iOS application sandbox. Although details of the vetting process and the sandbox are kept as black box by Apple, it was generally believed that these iOS security mechanisms are effective in defending against malwares.

In this paper, we propose a generic attack vector that enables third-party applications to launch attacks on non-jailbroken iOS devices. Following this generic attack mechanism, we are able to construct multiple proof-of-concept attacks, such as cracking device PIN and taking snapshots without user's awareness. Our applications embedded with the attack codes have passed Apple's vetting process and work as intended on non-jailbroken devices. Our proof-of-concept attacks have shown that Apple's vetting process and iOS sandbox have weaknesses which can be exploited by third-party applications. We further provide corresponding mitigation strategies for both vetting and sandbox mechanisms, in order to defend against the proposed attack vector.

1 Introduction

Digital mobile devices, such as smartphones and tablets, have been increasingly used for personal and business purposes in recent years. iOS from Apple is one of the most popular mobile operating systems in terms of the number of users. By Jan 2013, 500 millions of iOS devices had been sold worldwide and Apple's iTunes App Store contained over 800,000 iOS third-party applications, which had been downloaded for more than 40 billion times [1].

Third-party applications are pervasively installed on iOS devices as they provide various functions that significantly extend the usability of the mobile devices. On the other hand, these third-party applications pose potential threats to personal and business data stored on the devices. Thus, Apple adopts various security measures on its iOS platform to protect the device from malicious third-party applications. Among these security measures, Apple's application vetting

M. Jacobson et al. (Eds.): ACNS 2013, LNCS 7954, pp. 272–289, 2013.
© Springer-Verlag Berlin Heidelberg 2013

process and the iOS application sandbox are considered as the fundamental mechanisms that protect users from security and privacy exploits.

Each iOS third-party application is required to go through a vetting process before it is published on the official iTunes App Store, which is the only source of obtaining applications without jailbreaking an iOS device. Although details of the vetting process are kept secret, it is generally regarded as highly effective since no harmful malware on non-jailbroken devices has been reported on iTunes App Store [2,3]. Only graywares, which stealthily collect sensitive user data, were found on iTunes Store. These graywares were immediately removed from the store upon discovery [4].

When an application is downloaded and installed on an iOS device, it is given a limited set of privileges [5], which are enforced by the application sandbox. With the sandbox restrictions, an application cannot access files and folders of other applications. In order to access the required user data or control system hardware (e.g. Bluetooth or WiFi), applications need to call respective iOS APIs which are hooked by the sandbox so that validations of these API invocations are performed dynamically. The sandbox mechanism serves as the last line of defense which restricts malicious applications from accessing privileged system services, abusing user data or exploiting resources of other applications.

Due to the closed-source nature of iOS platform, the implementation details of security mechanisms used by iOS (including vetting process and application sandbox) are not officially documented. As a result, to our best knowledge, there is no systematic security analysis conducted for iOS platform, which has been generally believed as one of the most secure commodity operating systems [6].

In this paper, we make the first attempt in constructing generic attacks on iOS platform. Existing ad hoc attacks usually require root privilege [7,8,9] and thus work only on jailbroken iOS devices. In contrast, our attacks are intended to work on non-jailbroken iOS devices, which are protected by both vetting process and application sandbox. Thus, we propose an attack vector which include two attack stages: 1) In the first stage, malicious applications which are embedded with attack codes need to pass Apple's vetting process in order to appear in the official iTunes App Store; 2) In the second stage, after users have downloaded these applications onto their iOS devices, the attack codes need to bypass the restriction of the iOS sandbox in order to perform malicious functionalities. We realize both attack stages by exploiting the weaknesses of the vetting process and the iOS sandbox. With the proposed generic attack vector, we implement seven proof-of-concept attacks, such as cracking device PIN and taking screenshots without user's awareness, which impose serious threats to the security and privacy of iOS users. Most of our attacks implemented work on both iOS 5 and iOS 6. We implement multiple iOS applications and embed our attack codes into these applications, which are then submitted to the iTunes App Store. These applications with attack codes have passed the vetting process and all our attacks work effectively on non-jailbroken iOS devices[1]. Our proof-of-concept attacks and

[1] Due to privacy concerns, we embedded secret triggers in our applications so that public users will not be affected by the attack codes in these applications.

further validation experiments indicate that the current vetting process and iOS sandbox have vulnerabilities that can be exploited by malicious third-party applications to escalate their privileges and launch serious attacks on non-jailbroken iOS devices.

In order to defend against the proposed attacks, we further discuss several mitigation methods which could enhance both vetting process and iOS application sandbox. Some of these methods utilize existing iOS security features, thus can be conveniently implemented and deployed on the current iOS platform. We have notified Apple all of our findings and shared all our attack codes with Apple's product security team. By the time this paper was accepted, Apple is still in the progress of addressing the security issues we have discovered.

In summary, this paper makes the following contributions:

- We provide a generic attack vector which exploits the weaknesses of both vetting process and iOS application sandbox. The attack vector consists of two attack stages and can be used to construct serious attacks that work on non-jailbroken iOS devices.
- We implement seven proof-of-concept attacks with the attack vector proposed. We embed these attack codes into multiple applications we implemented and all the applications are able to pass the vetting process and appear on official iTunes Store.
- We suggest several mitigation methods to defend against our attacks. These methods include improvements on both the vetting process and the application sandbox, which can be deployed on the iOS platform conveniently.

2 Background and Threat Model

2.1 iOS Platform Overview

iOS platform follows a closed-source model, where source code of the underlying architecture and implementation details of its security mechanisms are not available to the public. Though it is debatable whether such obscurity provides better security, iOS has been generally believed as one of the most secure commodity operating systems [6]. Unlike other mobile platforms, third-party applications on iOS are given a more restricted set of privileges [5]. In addition, any third-party application developed for iOS must go through Apple's application vetting process before it is published on the official iTunes App Store. While some users and developers favor to have such restrictions for better security, others prefer to have more controls over the device for additional functionalities, such as allowing to install pirated software and allowing applications to change the themes of the device. To attain such extended privileges, an iOS device needs to be jailbroken. Jailbreaking is a process of installing modified kernel patches which allow a user to have root access of the device so that any unsigned third-party applications can run on it. Although jailbreaking is legal [10], it violates Apple's End User License Agreement and voids the warranties of the purchased devices. Jailbreaking is also known to expose to potential security attacks [7,8].

Application Vetting Process. Without jailbreaking a device, the only way of installing a third-party application on iOS is via the official iTunes App Store. Any application that is submitted to iTunes Store needs to be reviewed by Apple before it is published on the store. This review process is known as *Apple's application vetting process*. The vetting covers several aspects, including detection of malware, detection of copyright violations, and quality inspection of submitted applications. Although the vetting process is kept secret by Apple, it is generally regarded as highly effective as no harmful malware has been reported on iTunes Store [3,2]. Only grayware (which stealthily collects user data) had been reported and was removed from the store upon reporting [4,3].

Application Sandbox. iOS utilizes another security measure – application sandbox – to restrict privileges of third-party applications running on a device. The sandbox is implemented as a set of fine-grained access controls, enforced at the kernel level. Under the sandbox restrictions, an application cannot access files and folders of other applications. In order to access user data or control system hardware, applications also need to call respective Application Programming Interfaces (APIs) provided on iOS. These APIs are hooked by the sandbox so that validations of API invocations can be performed dynamically. The sandbox serves as the last line of security defense which limits malicious applications from accessing system services or exploiting resources of other applications.

iOS Frameworks and APIs. To facilitate development of third-party applications, a collection of *frameworks* are provided in Cocoa Touch [11], which include both public frameworks and private frameworks. Public frameworks are application libraries officially provided to third-party developers while private frameworks are intended only for Apple's internal developers. Each framework provides a set of APIs with which applications can access required system resources and services. Similar to frameworks, APIs can also be categorized into public APIs and private APIs.

Public APIs allow third-party applications to access a limited set of user information and control hardware of iOS devices, such as camera, Bluetooth and WiFi. In contrast, private APIs are the APIs that are meant to be used by Apple's internal developers. Private APIs may exist in both public and private frameworks. Though not officially documented, private APIs include various functions which could be used by a third-party application to escalate its restricted privileges. Thus, Apple explicitly forbids third-party developers from using private APIs and rejects applications once the use of private APIs is detected. On the other hand, private APIs can still be used by applications that are designed to run on jailbroken devices. Such applications are available through Cydia [12], which is an unofficial application market built for jailbroken iOS devices.

2.2 Threat Model

In this paper, we are interested in finding out the possible attacks which can be performed by third-party applications on non-jailbroken iOS devices, as illus-

trated in Figure 1. The success of such attacks depends on two major factors: 1) whether the corresponding malicious applications can pass Apple's vetting process and appear in the official iTunes App Store; and 2) whether malicious function calls can bypass the restriction of the iOS sandbox. We embed all our proof-of-concept attack codes in the applications we develop, which have passed Apple's vetting process and have been digitally signed by Apple. Thus, our attacks embedded in these applications are able to work on both jailbroken and non-jailbroken iOS devices.

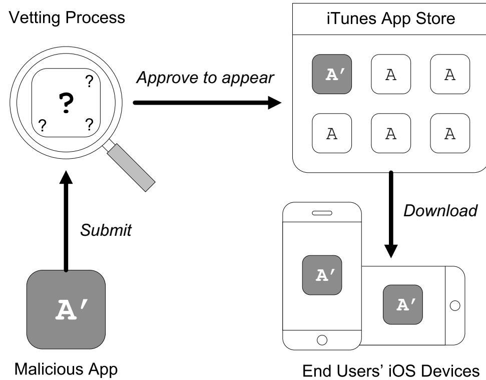

Fig. 1. Threat model

3 Generic Attack Vector

As introduced in Section 2, iOS private APIs exist in both private frameworks and part of public frameworks. When used by third-party applications, private APIs may provide additional privileges to the applications and thus are explicitly forbidden by the vetting process. We choose to utilize private APIs to construct our attacks which perform various malicious functionalities. In this section, we first present two ways of dynamically invoking private APIs which enable the malicious applications to pass the vetting process without being detected. Such dynamic loading mechanisms guarantee the success of the first stage in the proposed attack vector. For the second attack stage, in order to identify useful private APIs that are not restricted by iOS application sandbox, we manually analyze and test each iOS framework. Utilizing the useful private APIs we identified, we manage to implement multiple serious attacks that cover a wide range of privileged functionalities. These attacks can be embedded in any third-party applications, and they work effectively on non-jailbroken iOS devices.

Although our attack vector includes two stages, these two stages are not isolated – what private API needs to be utilized decides the way of its dynamic invocation. Thus, in the following, we first use SMS-sending and PIN-cracking

attacks as two examples to explain the underlying mechanisms of the entire attack vector. We then introduce other attacks we implemented utilizing the same attack vector and discuss the implications of these attacks.

3.1 Attacks via Dynamically Loaded Frameworks

When implementing a third-party iOS application that uses private APIs, the normal process is to link the corresponding framework statically (in the application's Xcode [13] project), and import the framework headers in the application's source code. For example, if a developer wants to send SMS programmatically in his application, CoreTelephony.framework needs to be linked, and CTMessageCenter.h needs to be imported in the application code. After preparing those preconditions, the SMS-sending private API can then be called as follows:

```
[[CTMessageCenter sharedMessageCenter]
    sendSMSWithText:@"A testing SMS"
    serviceCenter:nil
    toAddress:@"+19876543210"];
```

In the above code, the static method sharedMessageCenter returns an instance of CTMessageCenter class, and then invokes the private API call "sendSMSWithText:serviceCenter:toAddress:", which performs the SMS-sending functionality on iOS 5. Third-party application can utilize this method to send premium-rate SMS, and the sent SMS will not even appear in the SMS outbox (more precisely, it does not appear in the default iOS Message application[2]). Thus, a user would be totally unaware of such malicious behavior until the user receives his next phone bill.

However, this standard way of invoking private APIs can be easily detected by the vetting process, even though only the executable binary of the compiled application is submitted for vetting. One way of detecting this API call is to simply use string matching (e.g., "grep") on the binary, as the name of the function call appears in the binary's objc_methname segment (and also other segments). Moreover, the framework name and class name also appear in the binary as imported symbols. In this example SMS-sending code, although CoreTelephony is a public framework, CTMessageCenter.h is a private header (i.e., CTMessageCenter is a private class); thus, importing it in the source code can be detected by performing static analysis on the application's binary file. In order to pass Apple's vetting process, the application cannot link the framework statically.

To avoid being detected, the framework has to be loaded dynamically and the required classes and methods need to be located dynamically. In our attacks, we utilize Objective-C runtime classes and methods to achieve this goal. The example SMS attack code that illustrates the dynamic loading mechanism is given as follows:

[2] Another way of sending SMS programmatically on iOS 5 is to utilize MFMessageComposeViewController. However, this method is easy to be noticed as the SMS sent would appear in the default Message application.

```
1: NSBundle *b = [NSBundle bundleWithPath:@"/System/Library
    /Frameworks/CoreTelephony.framework"];
2: [b load];
3: Class c = NSClassFromString(@"CTMessageCenter");
4: id mc = [c performSelector:NSSelectorFromString(@"sharedMessage
   Center")];
5: // call "sendSMSWithText:serviceCenter:toAddress:" dynamically
   by utilizing NSInvocation
...
```

In the above code, the first two lines are used to load the CoreTelephony framework dynamically, without linking this framework in the application's source code. The path of this library is fixed on every iOS device, which is under the /System/Library/Frameworks/ folder. Note that not only public frameworks can be loaded dynamically, private frameworks (which is under /System/Library/PrivateFrameworks/) can also be loaded dynamically using the same method. According to our experiments, Apple's sandbox does not check the parameter of [NSBundle load] to forbid accessing these frameworks under /System/Library folder.

NSClassFromString at the third line is a function which can locate the corresponding class in memory by passing it the class name, which is similar to the "Class.forName()" method in Java reflection. At the fourth line, the sharedMessageCenter method is called via "performSelector:". At last, in order to call a method with more than 2 parameters (which is "sendSMSWithText:serviceCenter:toAddress:" in this case), the NSInvocation class is utilized.

Although the above code dynamically invokes the private API call, it may need certain obfuscation in order to avoid the detection from static analysis during the vetting process[3]. The last step of generating the actual attack code is to obfuscate all the strings appearing in the above example code. There are various ways of obfuscating strings in the source code. One simple technique is to create a constant string which includes all 52 letters (both upper and lower cases), 10 digits and common symbols. Then all the strings appeared in the above code can be generated dynamically at runtime by selecting corresponding positions from this constant string. Some of our applications utilize this method to obfuscate strings in the attack codes, and some others adopt a complex obfuscation mechanism, which involves bitwise operations and certain memory stack operations that are more difficult to be detected.

3.2 Attacks via Private C Functions

Information about private Objective-C classes and methods in the Cocoa Touch frameworks can be obtained from the iOS runtime headers [14], which are generated using runtime introspection tool such as RuntimeBrowser [15]. An example

[3] Actually according to our experiments, obfuscation may not be necessary, as the vetting process does not seem to check all text segments in the binary. In our experiments, we have tried to embed this SMS-sending code in one application which does not utilize obfuscation, and the application passed the vetting process.

of directly utilizing these Objective-C private APIs has been introduced in the previous subsection. However, Objective-C private classes and methods are not the only private APIs we are able to use in third-party applications.

When we reverse engineer the binary files of each framework, we find that there are a number of C functions in these frameworks that can be invoked by our application, which do not appear in the iOS runtime headers [14] and cannot be found with RuntimeBrowser [15]. In order to invoke these C functions, we need to dynamically load the framework binary and locate the function at runtime. The following code segment is part of our PIN-cracking code, which illustrates how we realize the dynamic invocation for private C functions.

```
void *b = dlopen("/System/Library/PrivateFrameworks
    /MobileKeyBag.framework/MobileKeyBag", 1);
int (*f)(id, id, id) = dlsym(b, "MKBKeyBagChangeSystemSecret");
...
int r = f(oldpwd, newpwd, pubdict);
...
```

In the above code segment, we use dlopen() to load the binary file of the private framework MobileKeyBag, which returns an opaque handle for this dynamic library. Utilizing this handle and dlsym(), we are then able to locate the address where the given symbol MKBKeyBagChangeSystemSecret is loaded into memory. This address is then casted into a function pointer so that it can be directly invoked later on in our attack code.

Although the above code segment may look simple, it is actually not easy to identify which C functions we should invoke to serve for our attack purpose, especially when only framework binary is given. Even after the C functions are identified and located, it takes further tedious work to figure out the correct parameter types and values to pass to the C functions. And in many cases, even all parameters are correct, these functions may be restricted by iOS sandbox and thus will not function correctly within third-party applications. To speed up the manual reverse engineering process when analyzing the given framework binaries, we build our own static analysis tool (which is based on IDA Pro.[16]) to disassemble the framework binary and obtain assembly instructions that are relatively easy to read.

By manually analyzing the private framework ManagedConfiguration, we find out that the changePasscodeFrom:to:outError: method of MCPasscodeManager is used to reset the password of the iOS device. However, we are not able to directly invoke this Objective-C method because the device needs to be "unlocked" first with current device password (possibly due to sandbox restrictions). Thus, we need to find a way of bypassing such restriction. Digging into the assembly code of the changePasscodeFrom:to:outError: method, we find out that it eventually invokes the MKBKeyBagChangeSystemSecret C function in MobileKeyBag to reset the password, which is allowed to be directly invoked under the sandbox restrictions. Further analysis and experiments are then conducted to figure out the correct parameters used to invoke MKBKeyBagChangeSystemSecret.

Our analysis reveals that the MKBKeyBagChangeSystemSecret function accepts three parameters, all of which have the type of (NSData*). The first parameter is the data of the old password, which can be converted from password string. The second parameter is the data of the new password. The third parameter, however, is an NSDictionary containing the "keyboard type" of the current password, which must be converted into NSData with [NSPropertyListSerialization dataFromPropertyList:format:errorDescription:]. One simple way of obtaining this NSDictionary data is to utilize the private framework ManagedConfiguration. However, in our attack code, to minimize the number of frameworks loaded, we utilize another private C function MKBKeyBagCopySytemSecretBlob[4] in MobileKeyBag to obtain this NSDictionary, which is then passed to MKBKeyBagChangeSystemSecret as the third parameter.

After this MKBKeyBagChangeSystemSecret function is successfully invoked, the rest of the attack code is straight forward – we simply use brute force to crack the password. 4-digit PIN has been widely used to lock iOS devices and has a password space of 10^4. When using our application to crack a device PIN on iPhone 5, it takes 18.2 minutes on the average (of 16 trials on two iPhone 5 devices) to check the whole PIN space (10^4). This gives an average speed of 9.2 PINs per second. To further speed up the cracking, we build a PIN dictionary so that common PINs are checked first. If the given PIN is in birthday format (mmdd/ddmm), it takes about 40 seconds to crack the PIN on average. Note that since our PIN-cracking attack uses the low level C functions, it will not trigger the "wrong password" event on the iOS device which is implemented at higher level (Objective-C functions) in the framework code. Thus, there is no limit on the number of attempts for our brute force attacks when cracking the device PIN. It is the same procedure to crack 4-digit PIN and complex password using our method, but the latter will take much longer time than PIN due to its large password space.

3.3 Other Implemented Attacks and Implications

The *SMS-sending* attack and the *PIN-cracking* attack introduced above explain how the entire attack vector is constructed. The former uses private Objective-C functions (Section 3.1), while the latter uses private C functions (Section 3.2). With the same dynamic invocation mechanisms which are able to bypass the vetting process, other attacks can also be implemented, as long as we can identify sensitive private APIs that are overlooked by the iOS sandbox.

We manually analyze the 180+ public and private iOS frameworks and manage to identify seven sets of sensitive APIs that are not restricted by iOS sandbox. Utilizing these APIs and the dynamic invocation mechanisms, we implement seven attacks, which are listed in Table 1. The corresponding frameworks and

[4] Note that it is not a spelling error in this MKBKeyBagCopySytemSecretBlob function. The key word "System" in this function name is spelled as "Sytem" by Apple's programmers. This detail further shows that in this attack, we utilize a function which Apple programmers may not expect to be used by third-party applications.

Table 1. The seven attacks implemented and their applicability

#	Attack Name	Description	iOS 5	iOS 6	iPhone	iPad*
1	PIN-cracking	Crack and retrieve the PIN of the device.	✓	✓	✓	✓
2	Call-blocking	Block all incoming calls or the calls from specified numbers.	✓	✓	✓	–
3	Snapshot-taking	Continuously take snapshots for current screen (even the app is at background).	✓	✓	–	✓
4	Secret-filming**	Open camera secretly and take photos or videos without the user's awareness.	✓	✓	✓	✓
5	Tweet-posting	Post tweets on Twitter without user's interaction.	✓	✓	✓	✓
6	SMS-sending	Send SMS to specified numbers without the user's awareness.	✓	–	✓	–
7	Email-sending	Send emails using user's system email accounts without the user's awareness.	✓	–	✓	✓

* The call-blocking and SMS-sending attacks do not work on iPad, simply because iPad does not have corresponding functionalities since it is not a phone device.
** This secret-filming attack can be implemented purely with iOS public APIs.

key APIs utilized are listed in Table 2 in the appendix. We embed our attack codes in multiple applications we develop, and all those applications have passed Apple's vetting process and appeared in the official iTunes App Store.

Most of the attacks in Table 1 work on both iOS 5 and iOS 6 (which is the default iOS version on iPhone 5). The last two attacks (*SMS-sending* and *email-sending*) currently only work on iOS 5, but not iOS 6. The APIs of sending SMS and emails on iOS 6 have been substantially changed to prevent such attacks (which will be further analyzed in Section 4).

The severity of most of our attacks would be significantly increased when the attack code is embedded in an application that can keep running at the background. Take the snapshot attack as an example. By calling the private API [UIWindow createScreenIOSurface], an application can capture the current screen content of the device. When continuously running at the background, this application can take snapshots of the device periodically, and send these snapshots back to the developer's server for further analysis[5]. Such *snapshot-taking* attack may reveal user's email content, photos and even bank account information, thus it should be avoided on any mobile devices.

Similar to the *snapshot-taking* attack, the *call-blocking* and *PIN-cracking* attacks also become more serious when they are used in an application that can continuously run at the background, which have been verified in our experi-

[5] The snapshot attack code is embedded into one of our applications which can keep running at background utilizing audio playing feature. This application also passed Apple's vetting process and it sends out snapshots every 5 seconds once triggered.

ments. However, the *secret-filming* attack does not work when in background. The current implementation of the iOS camera service requires that an application utilizing this service be not in the background status. Nevertheless, even if the *secret-filming* attack works only when the application is in the foreground, it is still a serious threat to user privacy. Considering that when a user is playing a game on the iOS device, and the game secretly opens the cameras and takes photos periodically without the user's notice. In our experiments, we have verified that both front and back cameras can be used, and the sound can be muted when taking videos or photos programmatically in our applications.

We emphasize that all these attacks are implemented with secret triggers in the applications that are submitted to iTunes Store. The attacks are only launched on our testing devices after certain sequences of secret buttons have been pressed in the applications. However, note that in the application codes, such triggers are just "if-else" statements. Thus, if the trigger conditions were replaced with an "if-true" condition, these attacks could be launched on any user device with such applications. Therefore, the secret triggers used in our proof-of-concept applications do not affect the conclusions drawn from our experiments.

Besides the seven attacks we have implemented, our attack vector can be used to construct other attacks as long as there are security sensitive functions on iOS that are not restricted by iOS sandbox. As each iOS version will include new functionalities to the platform, each iOS update may introduce new attacks from malicious third-party applications based on our attack vector.

4 Attack Mitigation

Our proof-of-concept attacks have shown that Apple's current vetting and sandbox mechanisms have weaknesses which can be exploited by third-party applications to escalate their privileges and perform serious attacks on iOS users. In this section, we first suggest improvements on the vetting process to mitigate the security threats caused by dynamic invocations. We then propose enhancements on the iOS sandbox to further defend against our attacks utilizing private APIs.

4.1 Improving Application Vetting Process

Static analysis can be used to determine all the API calls which are not invoked with reflection (i.e., dynamic invocations), and it can provide the list of frameworks that are statically linked in the application. Thus, an automated static analysis is able to detect the standard way of invoking private APIs, as what is probably being used by Apple in its current vetting process. In addition, we suggest to improve the existing static analysis to detect suspicious applications based on certain code signatures. For example, one suspicious code signature could be applications containing any `dlopen()` or `[NSBundle load]` invocations whose parameters are not constant strings (which match the cases of our attacks). However, as the static analysis alone is not sufficient to determine whether a suspicious application is indeed a malware or not, manual examination and dynamic analysis should be utilized to examine such suspicious applications.

In many cases, manual examination may not be able to find malicious behaviors of the examined applications, because the malicious functions may not be preformed for every execution. Instead, they can be designed in the way that such functions are only triggered when certain conditions have been satisfied. Examples of such conditions include time triggers or button triggers (as what have been used in our applications). When a malicious application uses such trigger strategy, the manual inspection may not find any suspicious behaviors during the vetting process. Such malicious applications can only be detected by utilizing fuzz testing [16] (or in the extreme case, using symbolic execution [17]), where different inputs are used to satisfy every condition of the application code. Furthermore, in order to determine whether sensitive user data are transferred out of the device, dynamic taint analysis [18] is an effective approach to serve this purpose. However, since it is expensive to apply fuzz testing and dynamic taint analysis on every application, the vetting process may choose to run such examinations only on selected suspicious applications.

4.2 Enhancement on iOS Sandbox

Dynamic Parameter Inspection. From the perspective of iOS sandbox, a straightforward defense to our attacks that utilize the dynamic loading functions (such as [NSBundle load] and dlopen()) is to forbid third-party applications to invoke these functions. However, it is not practical to completely forbid the invocation of dynamic loading functions, since frameworks, libraries and many other resources need to be dynamically loaded for benign purposes at runtime. Even Apple's official code, including both framework code and application code (which is automatically generated by Xcode), utilizes dynamic loading functions extensively to load resources at runtime. On the other hand, since sensitive APIs can be hooked by utilizing the application sandbox, the parameters of these APIs can be checked at runtime. Thus, it is useful if Apple's sandbox is modified in the way that the parameter values passed to dynamic loading functions are examined, and accessing files under a specific folder is forbidden.

One way of implementing this approach is to forbid the third-party applications to dynamically load any frameworks under "/System/Library/" folder. However, a sophisticated attacker may be able to completely reverse engineer a given framework binary, locate all the code regions in the binary that are needed for launching his attack, and then copy only the needed code regions from the binary and insert into his application code. In this way, he does not need to dynamically load framework binaries in his malicious applications. Therefore, this parameter-inspection approach is not able to completely defend against the proposed attacks, though it can increase the complexity for the adversary to construct these attacks.

Privileged IPC Verification. Another technique of enhancing the sandbox is to dynamically check the privilege of the identity which makes sensitive API calls. For example, a third-party application should not have the privilege to invoke

MKBKeyBagChangeSystemSecret API, which is used in our PIN-cracking attack. Such private APIs should only be invoked by processes or services with the system privilege. However, directly restricting the access to private APIs may not effectively prevent the attacks. By analyzing the implementation of several private APIs (in assembly code), we find that the private APIs eventually use inter-process communication (IPC) methods, which communicate with the system service process, to complete the functionalities of the private APIs. For example, MKBKeyBagChangeSystemSecret API uses perform_command() method to communicate with the system service (with service_bundle_id = "com.apple.mobile.keybagd"). This means that instead of invoking private APIs, an application can also use such IPC method to directly send command to the system service process to perform the same functionality.

In order to defend against such attacks, for each privileged system service, the recipient of the command (which is the service process itself) needs to check the sender of the command to verify whether the sender has the valid privilege to make such IPC. To enable this IPC verification, the system service process needs to maintain a list of privileged IPC commands which are checked dynamically when an IPC is received. Compared to the parameter-inspection approach, privileged IPC verification provides better defense against the *PIN-cracking*, *call-blocking* and *snapshot-taking* attacks as the corresponding privileged functionalities should not be used by any third-party applications. However, this approach alone is not sufficient to mitigate the other four attacks listed in Table 1. For these four attacks, the corresponding functionalities should be provided to applications due to usability reasons, but at the same time, it needs to be ensured that user interactions are involved when these functionalities are performed.

Service Delegation Enhancement. On iOS 6, Apple starts using the XPC Service, which allows processes to communicate with each other asynchronously so that it can be used for privilege separation. Originally on iOS 5, the SMS and email APIs are implemented as "View Controller" classes that are created and used within a third-party application process. Therefore, applications can manipulate these view controller classes to send out SMSes and emails programmatically without users' interaction. However, on iOS 6, the SMS and email functionalities are now delegated to another system process utilizing XPC Service, which is completely out of the process space of third-party applications. Thus, a third-party application on iOS 6 is no longer able to send SMSes or emails programmatically without user's interaction.

Although currently iOS 6 has not implemented the service delegation mechanism for the Twitter service, the *tweet-posting* attack can be prevented using this mechanism, as it follows exactly the same service model as SMS and email. The *secret-filming* attack, however, cannot be easily mitigated using such service delegation. Instead of using a unified user interface, iOS enables third-party applications to create their own customized user interfaces for taking photos or videos. If the same service delegation mechanism is applied, then the camera interface will be identical across different applications as it is provided by system

service. Thus, more precisely, service delegation is able to defend against camera device abuse, but its implementation may greatly impact user experience.

System Notifiers for Sensitive Functionalities. In order to mitigate the threat of secret filming, while preserving the functionality and flexibility of using camera in third-party applications on iOS, one possible solution is to add a half-transparent system notifier on the screen (e.g., at the upper-right corner), whenever the camera device is being used. This notifier can be shown using the XPC mechanism so that the notifier is handled by a system daemon process, which is outside of the control of third-party applications. In this way, whenever the camera is being used (either taking photos or taking videos), the system notifier is shown on the screen to alert the user.

By enhancing the current iOS platform with the 1) privileged IPC verification, 2) comprehensive service delegation, and 3) extended system notifiers, it will be able to defend against all the seven attacks we construct. Note that since iOS is a close-source platform, it is extremely difficult (if not impossible) for us to implement these mitigation methods we proposed, and thus it is one of the limitations in our work. However, we have shared all our mitigation suggestions with Apple so that Apple's product security team may choose some of these methods to fix the sandbox. From the partial knowledge that is revealed by our attacks and the mitigation analysis, it may be inferred that the current iOS sandbox implementation is quite complex and its privilege check is not complete. Due to its complexity and also its trade-off nature against usability, it may not be easy to completely fix the iOS sandbox to prevent future attacks.

5 Discussions

On the current iOS platform, when an application plays an audio file (e.g., .mp3), normally a music-playing notifier (i.e., the ▶ symbol) is shown in the status bar on top of the screen. However, this only happens when the application is implemented following the standard programming rules, which require the application code to call [[UIApplication sharedApplication] beginReceivingRemoteControlEvents]. This API call registers the application in the system service so as to receive remote events, such as when a user presses the control buttons on earphone. In the background running application we implement, however, this API is not invoked and our application simply calls the basic audio playing APIs to play a silent music in an infinite loop. As a result, *no notifier is shown on the status bar when our application is running at the background*, thus the iOS user may be totally unaware of the existence of this security threat. In addition to playing audio, there are other means of enabling background running, such as VOIP and tracking locations. Thus, besides the system notifier for the camera functionality (Section 4.2), we suggest to add another system notifier specifically designed to indicate that an application is running at the background. Upon seeing this notifier, a user can force close any background applications that are not being used. This will not only enhance security but also save device battery.

The PIN-cracking attack code introduced in Section 3.2 not only can be used to steal device PIN and send it to an external server, but can also be used to reset the current PIN to another value so that the legitimate user is not able to unlock the device. In iOS settings, there is an option to "erase all data on this device after 10 failed passcode attempts". If this option is enabled on a device and our PIN-cracking code resets the PIN, it could make a user panic if he is unable to unlock the device after several trials of inputting his original password. Again note that our PIN-cracking attack itself will not trigger the "wrong password" event on the iOS device and thus, there is no limit on the number of brute forcing trials for our attack code when cracking the device PIN.

With the attack codes we shared with Apple's product security team, the PIN-cracking vulnerability has been fixed in the newly released iOS 6.1 (January 2013). However, other security issues we discovered are still in the process of being addressed. Note that the conclusions about the vetting process and sandbox given in this paper are inferences based on observations from our experiments, as the details of the vetting process and sandbox are kept as black box by Apple. The ground truth may become available to the public when Apple decides to turn major components of iOS into open source in the future, as what has been done for Mac OS X [19].

6 Related Work

Spyphone [20] is a prototype application, developed for iOS 3.1.2, which illustrates that a wide list of user data can be accessed on iOS by third-party applications. However, Spyphone does not use any private APIs – it only invokes public APIs and reads public files to access user data in order to enable itself to appear in iTunes Store [20], which is completely different from our malicious applications implemented. In addition, the security enforcement of iOS has been significantly improved since then so that a large portion of user data that can be accessed by Spyphone on iOS 3 is forbidden to access since iOS 5.

Malwares, such as iKee [7] and Dutch 5 ransom [8] worms, have been found on iOS. However, these worms only work on jailbroken iOS devices where an SSH server is installed with the default root password unchanged. Other iOS malwares known to the public, such as iSAM created by Damopoulos et al. [9] (which focuses more on malware propagation methods), also exploit vulnerabilities exist only on jailbroken iOS devices, which are different from our work.

Felt et al. [3] conduct a survey on the modern mobile malware in the wild, which encompasses all known iOS, Symbian, and Android malwares that spread between January 2009 and June 2011. They find that (i) all the 4 iOS malwares they identified work only on jailbroken iOS devices, and none were listed in the iTunes App Store; and (ii) only graywares are found on iTunes App Store which are then removed by Apple. These findings are confirmed by Egele et al. [21], in which they develop a static analysis tool, PiOS, to detect privacy leakages in iOS applications. They perform static analysis on more than one thousand third-party iOS applications and find out that only a few applications are graywares which stealthily access user data without user's awareness.

Extensive researches have been conducted on the other popular mobile platform – Android. Privilege escalation attacks on Android are proposed by [22], and the defense mechanisms for such attacks are introduced by Bugiel et al. [23]. Enck et al. [24] performs static analysis of Android applications using the decompiler they developed. Dynamic taint analysis on third-party Android applications is performed by TaintDroid [25]. Comprehensive surveys on mobile security are provided by Becher et al. [26] and Egners et al. [27].

The closest work to our research is the work by Miller [28]. By exploiting the security flaw he found, he managed to get iOS devices to run unsigned codes which are dynamically downloaded by his proof-of-concept malicious application. Miller's attack mechanism provides an alternative for the first stage of our proposed attack vector. However, Apple has removed his application from the iTunes App Store and released a fix for the security flaw. Thus, our dynamic invocation used in the first stage, to our best knowledge, is the only way of bypassing the vetting process. Although our mechanism is not complex, it is a very effective way of allowing malicious applications appear in the official application store. Furthermore, by performing sophisticated analysis on all existing iOS frameworks, we identify seven sets of sensitive APIs which are not restricted by iOS sandbox and thus can be utilized by any malicious applications.

7 Conclusion

The original goal of this work is to answer a simple (but not easy) research question: is there a generic attack vector which enables third-party applications to launch attacks on non-jailbroken iOS devices? Two pre-conditions need to be satisfied in answering this question: (i) the third-party application has to pass the vetting process and appear on the official application store; and (ii) the corresponding attack codes must break through the restrictions of iOS sandbox in order to work on non-jailbroken iOS devices.

In this paper, we constructed effective mechanisms which allow any third-party application to invoke private APIs without being detected by the vetting process. By utilizing such mechanisms and exploiting the vulnerabilities in the application sandbox, we implemented seven proof-of-concept attacks which can cause serious damages to iOS users. Finally, we suggested mitigation mechanisms to enhance the current vetting process and iOS sandbox. Our paper fills the gap in the current mobile security literature where most research efforts are conducted on Android platform. We have shared all our findings with Apple's product security team. In January 2013, Apple released iOS 6.1 and fixed the PIN-cracking vulnerability we discovered in iOS 6.0, while other security issues presented in this paper still remain unsolved.

Acknowledgments. This work was partially supported by project SecSG-EPD090005RFP(D) funded by Energy Market Authority, Singapore. We also thank the anonymous reviewers for their valuable insights and comments.

References

1. Apple Press Info: App Store Tops 40 Billion Downloads with Almost Half in 2012 (January 2013), http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html
2. Safe and Savvy: How secure is your iPhone (June 2012), http://safeandsavvy.f-secure.com/2012/06/29/how-secure-is-your-iphone/
3. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14 (2011)
4. TrendLabs: Malware for iOS? Not Really (June 2012), http://blog.trendmicro.com/trendlabs-security-intelligence/malware-for-ios-not-really/
5. Han, J., Yan, Q., Gao, D., Zhou, J., Deng, R.H.: Comparing Mobile Privacy Protection through Cross-Platform Applications. In: Proceedings of the Network and Distributed System Security Symposium (February 2013)
6. macgasm.net: IT Professionals Rank iOS As Most Secure Mobile OS (August 2012), http://www.macgasm.net/2012/08/17/it-professionals-rank-ios-as-most-secure-mobile-os/
7. NakedSecurity: First iphone worm discovered - ikee changes wallpaper to rick astley photo (November 2009), http://nakedsecurity.sophos.com/2009/11/08/iphone-worm-discovered-wallpaper-rick-astley-photo/
8. NakedSecurity: Hacked iphones held hostage for 5 euros, http://nakedsecurity.sophos.com/2009/11/03/hacked-iphones-held-hostage-5-euros/
9. Damopoulos, D., Kambourakis, G., Gritzalis, S.: iSAM: An iPhone Stealth Airborne Malware. In: Camenisch, J., Fischer-Hübner, S., Murayama, Y., Portmann, A., Rieder, C. (eds.) SEC 2011. IFIP AICT, vol. 354, pp. 17–28. Springer, Heidelberg (2011)
10. Kravets, D.: ABCNews: Jailbreaking iPhone Legal, U.S. Government Says, http://abcnews.go.com/Technology/story?id=11254253
11. iOS Technology Overview: Cocoa Touch, https://developer.apple.com/technologies/ios/cocoa-touch.html
12. Freeman, J.: Cydia, an alternative to Apple's App Store for jailbroken iOS devices, http://cydia.saurik.com/
13. Apple Developer: Xcode, Apple's integrated development environment for creating apps for Mac and iOS, https://developer.apple.com/xcode/
14. Seriot, N.: iOS 6 runtime headers, https://github.com/nst/iOS-Runtime-Headers
15. Seriot, N.: Objective-C Runtime Browser, for Mac OS X and iOS, https://github.com/nst/RuntimeBrowser/
16. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated Whitebox Fuzz Testing. In: Proceedings of the Network and Distributed System Security Symposium (2008)
17. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 504–515 (2011)
18. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In: Proceedings of the Network and Distributed System Security Symposium (2011)
19. apple.com: Apple Open Source Projects, http://www.apple.com/opensource/
20. Seriot, N.: iPhone Privacy. In: Black Hat DC (2010)

21. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: Proceedings of the Network and Distributed System Security Symposium (2011)
22. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX Security Symposium (2011)
23. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: Annual Network & Distributed System Security Symposium (February 2012)
24. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium (2011)
25. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI (2010)
26. Becher, M., Freiling, F.C., Hoffmann, J., Holz, T., Uellenbeck, S., Wolf, C.: Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In: Proceedings of the IEEE Symposium on Security and Privacy (2011)
27. Egners, A., Marschollek, B., Meyer, U.: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms, Technical Report (2012)
28. Miller, C.: Apple lets malware into App Store (2011), http://nakedsecurity.sophos.com/2011/11/08/apples-app-store-security-compromised/

A Details in Attack Implementations

The frameworks and key APIs utilized in our attacks are given in Table 2.

Table 2. The frameworks and key APIs utilized for the seven attacks implemented

#	Attack	Frameworks	Classes*	Functions
1	PIN-cracking	MobileKeyBag	—	MKBKeyBagChangeSystemSecret MKBKeyBagCopySytemSecretBlob
2	Call-blocking	CoreTelephony	—	CTTelephonyCenterGetDefault CTTelephonyCenterAddObserver CTCallCopyAddress CTCallDisconnect
3	Snapshot-taking	UIKit	UIWindow UIImage	createScreenIOSurface initWithIOSurface:
4	Secret-filming	AVFoundation CoreMedia CoreVideo	AVCaptureDevice AVCaptureDeviceInput AVCaptureVideoDataOutput AVCaptureSession	devices deviceInputWithDevice:error: setSampleBufferDelegate:queue: startRunning
5	Tweet-posting	Twitter	TWTweetComposeViewController	setCompletionHandler: setInitialText: send:
6	SMS-sending	CoreTelephony	CTMessageCenter	sharedMessageCenter sendSMSWithText:serviceCenter:-toAddress:
7	Email-sending	Message AppSupport	MailAccount CPDistributedMessagingCenter	defaultMailAccountForDelivery uniqueId centerNamed: sendMessageAndReceiveReplyName:userInfo:error:

* The symbol of "—" in the Class field indicates that the corresponding attack does not utilize any Objective-C classes, but only utilizes private C functions.