Jekyll on iOS: When Benign Apps Become Evil

Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee,
Georgia Institute of Technology

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Jekyll[*] on iOS: When Benign Apps Become Evil

Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee

School of Computer Science, College of Computing, Georgia Institute of Technology

{*tielei.wang, kangjie.lu, long, pchung, wenke*}*@cc.gatech.edu*

Abstract

Apple adopts the mandatory app review and code signing mechanisms to ensure that only approved apps can run on iOS devices. In this paper, we present a novel attack method that fundamentally defeats both mechanisms. Our method allows attackers to reliably hide malicious behavior that would otherwise get their app rejected by the Apple review process. Once the app passes the review and is installed on an end user's device, it can be instructed to carry out the intended attacks.

The key idea is to make the apps remotely exploitable and subsequently introduce malicious control flows by rearranging signed code. Since the new control flows do not exist during the app review process, such apps, namely *Jekyll apps*, can stay undetected when reviewed and easily obtain Apple's approval.

We implemented a proof-of-concept Jekyll app and successfully published it in App Store. We remotely launched the attacks on a controlled group of devices that installed the app. The result shows that, despite running inside the iOS sandbox, Jekyll app can successfully perform many malicious tasks, such as stealthily posting tweets, taking photos, stealing device identity information, sending email and SMS, attacking other apps, and even exploiting kernel vulnerabilities.

1 Introduction

Apple iOS is one of the most popular and advanced operating systems for mobile devices. By the end of June 2012, Apple had sold 400 million iOS devices [30], such as iPhone, iPad and iPod touch. Despite the tremendous popularity, in the history of iOS, only a handful of malicious apps have been discovered [24]. This is mainly attributed to the advanced security architecture of iOS and the strict regulations of the App Store.

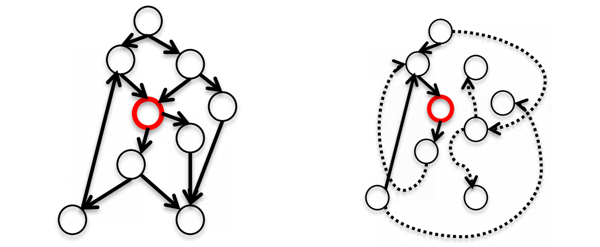
In addition to the standard security features like Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Sandboxing, iOS enforces the mandatory App Review and code signing mechanisms [31]. App Review inspects every app submitted by third parties (in binary form) and only allows it to enter the App Store if it does not violate App Store's regulations [5]. To further prohibit apps distributed through channels other than the App Store (i.e., unsigned apps), the code signing mechanism disallows unsigned code from running on iOS devices. As a result, all third-party apps running on iOS devices (excluding jailbroken devices [48]) have to be approved by Apple and cannot be modified after they have obtained the approval.

According to the official App Review guidelines [5], developers should expect their apps to go through a thorough inspection for all possible term violations. During this process, many reasons can lead to app rejections, such as stealing data from users and using private APIs reserved for system apps. Although the technical details of the review process remain largely unknown, it is widely believed that such a selective and centralized app distribution model has significantly increased the difficulty and cost for malicious or ill-intended apps to reach end users.

In this paper, we present a new attack method against the App Store reviewing process and the code signing mechanism. Using this method, attackers can create malicious or term-violating apps and still be able to publish them on App Store, which in turn open up new attack surfaces on iOS devices. We stress that our attack does not assume any specifics about how Apple reviews apps, but targets theoretical difficulties faced by any known methods to analyze programs. By demonstrating the power of this practical attack, we highlight the shortcomings of the pre-release review approach and call for more runtime monitoring mechanisms to protect iOS users in the future.

The key idea behind our attack is that, instead of sub-

[*] Jekyll is a character with dual personalities from the novel *The Strange Case of Dr. Jekyll and Mr. Hyde*.



CFG exhibited in vetting process. Red node represents the vulnerabilities.

Dynamic CFG in victim's device after exploiting vulnerabilities.

Figure 1: High Level Intuition

mitting an app that explicitly contains malicious functionalities to Apple, the attacker plants remotely exploitable vulnerabilities (i.e., backdoor) in a normal app, decomposes the malicious logic into small code gadgets and hides them under the cover of the legitimate functionalities. After the app passes the App Review and lands on the end user device, the attacker can remotely exploit the planted vulnerabilities and assemble the malicious logic at runtime by chaining the code gadgets together.

Figure 1 shows the high level idea. On the left is the app's original control flow graph (CFG), which is what can be observed during the app review process, without the planted vulnerability being exploited. In comparison, on the right is the effective control flow graph the same app will exhibit during runtime, which differs from the left in the new program paths (represented by the dotted paths) introduced at runtime by the remote attackers (i.e., app developers). Since attackers can construct malicious functionalities through dynamically introducing new execution paths, even if the vetting process could check all possible paths in the left CFG (i.e., 100% path coverage), it cannot discover the malicious logic that is only to be assembled at runtime as per attacker's commands. Apps so constructed bear benign looks and yet are capable of carrying out malicious logic when instructed; we call them Jekyll apps. By carefully designing the vulnerabilities and crafting the gadgets, Jekyll apps can reliably pass app review process and open up a new attack surface on iOS devices when installed. Specifically, an attacker can achieve the following general tasks via Jekyll apps:

First, Jekyll apps offer an approach to stealthily abuse user privacy and device resources, for instance, via private APIs[1], which may provide unrestricted access to certain sensitive resources and are intended for Apple's internal use only. Explicit use of private APIs almost always gets an app rejected by App Store [4]. However, Jekyll apps can dynamically load, locate, and implicitly invoke the private APIs and thus reliably bypass the review checks. Comparing with simple obfuscation techniques (e.g., [7, 23, 25]), our approach hides the usage of private APIs in a way that is more resilient to non-trivial code analysis — without correctly triggering the planted vulnerabilities and arranging the code gadgets, the invocation of private APIs never appears in the code and execution of Jekyll apps.

Second, Jekyll apps open a window for attackers to exploit vulnerabilities in kernel space. Although the sandboxing policy in iOS limits the possibility and impact of exploiting kernel vulnerabilities [22] by third-party apps, certain attacks are still effective against vulnerable device drivers (i.e., IOKit drivers [49]).

Third, Jekyll apps also serve as a trampoline to attack other apps. On iOS, by requesting a URL, an app can launch another app that has registered to handle that URL scheme. However, this simplified IPC (Inter-process communication) mechanism may facilitate inter-app attacks. For instance, once new vulnerabilities have been found in Mobile Safari (the built-in web browser in iOS), an attacker can set up a malicious webpage exploiting such vulnerabilities, use the Jekyll app to direct the Mobile Safari to visit the booby-trapped website, and eventually compromise the browser app. Given the high privileges granted to Mobile Safari, the compromised browser will in turn provide the stepping stone for more powerful attacks, such as untethered jailbreak, as shown by the JailbreakMe attack [1] on old versions of iOS.

Attack Type	Attack Description	Affected Version
Abuse Device Resources	Sending SMS	iOS 5.x
	Sending Email	iOS 5.x
	Posting Tweet	iOS 5.x & 6.x
	Abusing Camera	iOS 5.x & 6.x
	Dialing	iOS 5.x & 6.x
	Manipulating Bluetooth	iOS 5.x & 6.x
	Stealing Device Info	iOS 5.x & 6.x
Attack Kernel	Rebooting system	iOS 5.x
Attack Other Apps	Crashing Mobile Safari	iOS 5.x & i6.x

Table 1: Attack summary on iPhone

We have implemented a proof-of-concept Jekyll app and submitted it to the App Store. The app successfully passed Apple's review despite the hidden vulnerabilities and code gadgets that can be assembled to carry out malicious logic. Following the ethical hacking practice, we immediately removed the app from App Store once a group of experiment devices of our control had downloaded it. The download statistic provided by Apple later confirmed that the app had never been downloaded by any other users. By exploiting the vulnerabilities and chaining the planted gadgets in the app, we

[1] Private APIs are undocumented and often security-critical APIs on iOS, see Section 2.2 for details.

remotely launched many malicious operations on our experiment devices, as summarized in Table 1. Even on iOS 6.1.2, the latest version of iOS at the time of our experiments, the Jekyll app can abuse the camera device to recode videos, post tweets, steal device identity information such as IMEI (the unique device identifier), manipulate the bluetooth device, attack Mobile Safari, and dial arbitrary number. We made a full disclosure of our attack scheme to Apple in March 2013 and have since been in correspondence with Apple.

In summary, the main contributions of our work are as follows:

- We propose a novel method to generate iOS apps that can pass App Review and synthesize new control flows as instructed remotely during runtime, without violating code signing. We call such malicious apps Jekyll apps. Given that arbitrary control flows can be introduced to such apps at runtime, the code signing mechanism on iOS is totally defenseless against Jekyll apps.

- We are the first to propose a dynamic analysis technique to discover the private APIs used to post tweets, send email, and send SMS without user's consent on iOS. We incorporate these attacks, along with a set of previously known iOS attacks, into a Jekyll app to show its versatility.

- We successfully publish a proof-of-concept Jekyll app in Apple App Store and later launch remote attacks to a controlled group.

- We demonstrate that the security strategy to solely rely on pre-install review, as currently followed by Apple App Store, is ineffective against Jekyll apps and similar attacks. We discuss and advocate runtime security measures as a necessary step in advancing iOS security.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 presents a motivating example and describes the design of our attack scheme. Section 4 demonstrates some of the malicious operations that can be carried out by Jekyll apps. Section 5 gives the implementation details and Section 6 compares our research to related work. Section 7 discusses the potential countermeasures against our attack and Section 8 concludes the paper.

2 Background

2.1 iOS Security

iOS provides a rich set of security features. We briefly introduce the related exploit mitigation mechanisms here. Interested readers are referred to [31, 38] for the overall security architecture of iOS.

DEP and ASLR. Apple introduced the Data Execution Prevention (DEP) mechanism in iOS 2.0 and later the Address Space Layout Randomization (ASLR) mechanism in iOS 4.3 [21]. The DEP mechanism in iOS is based on the NX (eXecute Never) bit supported by the ARM architecture and the kernel prevents third party apps from requesting memory pages that are writeable and executable at the same time. Since data pages such as the stack and heap are marked non-executable and code pages are marked executable but non-writeable, DEP prevents the traditional code injection attacks that need to write payloads into memory and execute them.

ASLR randomizes a process's memory layout. If a third-party app is compiled as a position-independent executable (PIE), the locations of all memory regions in its process's address space, including the main executable, dynamic libraries, stack, and heap, are unpredictable. As an important complementary to DEP, ASLR makes it very difficult for attackers to launch return-to-libc based or return-oriented programming based attacks (see Section 2.3). However, ASLR in iOS only enforces the module level randomization, that is, executable modules are loaded into unpredictable memory regions, but the internal layout of each module remains unchanged. Thus, the ASLR implementation is vulnerable to information leakage vulnerabilities [45]. If an attacker can obtain the absolute address of a function in a module, she is able to infer the memory layout of that entire module.

Privilege Separation and Sandboxing. iOS employs traditional UNIX file permission mechanisms to manage the file system and achieve the basic privilege separation. While all third-party apps run as the non-privileged user `mobile`, only a few most import system processes run as the privileged user `root`. As a result, third-party apps are not able to change system configurations.

To enforce isolation among apps that all run as the same user `mobile`, iOS utilizes the sandboxing mechanism. iOS sandbox is implemented as a policy module in the TrustedBSD mandatory access control framework [8]. Each app contains a plist file in XML format, which declares a set of entitlements for the special capabilities or security permissions in iOS. When an app is launched, iOS determines its sandbox policy according to its entitlements.

Although the built-in apps in iOS, such as Mobile Safari, run as the non-privileged user `mobile`, they may be granted with special privileges via reserved entitlements. For instance, Mobile Safari has an entitlement called `dynamic-codesigning`, which allows Mobile Safari to allocate a writable and executable memory buffer and generate executable code on the fly—a security exception made for Mobile Safari's Just-in-Time

(JIT) JavaScript engine to improve performance.

As for third-party apps, Apple applies a one-size-fits-all sandbox policy called `container`. According to the study in [51], in iOS 4.3, this permissive policy allows third-party apps to read the user's media library, interact with a few IOKit User Clients, communicate with the local Mach RPC servers over the bootstrap port, access the network, etc. On top of the default access granted by the `container` policy, third party apps can also request for two extra entitlements: one for using the iCloud storage and one for subscribing to the push notification service. Finally, even though the `container` policy has undergone significant improvements and is becoming more restrictive over time, as we show in this paper, our Jekyll app, even running in sandbox, still poses a significant threat to the user's privacy and system security.

Also, in contrast to other mobile platforms, such as Android, which use the declarative permissions to regulate each app individually, iOS applies *the default sandbox configuration on most third-party apps*, which consequently share the same broad set of privileges. As of iOS 6, only a few sensitive operations, such as accessing location information and contact book and sending push notifications, have to be explicitly acknowledged by users before they can proceed.

Code signing, App Store, and App Review. Along with the release of iOS 2.0 in 2008, Apple opened the App Store, an application distribution platform for iOS devices. Third-party developers are required to submit their apps to App Store for distribution. Since then, iOS has enforced the mandatory code signing mechanism to ensure only the executables that have been approved and signed by Apple are allowed to run on iOS devices. The study in [37] presents the implementation details of iOS code signing mechanism. In comparison with DEP, code signing mechanism is more strict. In a DEP-enabled system, attackers can compromise a process using ROP attacks and then download a new binary and run it. This does not apply to iOS because iOS will refuse to run the new binary if it is not signed by a trusted authority.

To release an app through App Store, a third-party developer has to participate in Apple's iOS developer program and submit the app to Apple for review. The app is signed and published by Apple only after it passes the review process. In addition to business benefits, the mandatory review process helps Apple prevent malicious apps from entering App Store.

2.2 Public and Private Frameworks

iOS provides the implementation of its system interfaces in special packages called frameworks. A framework is a directory that contains a dynamic shared library and the related resources such as images, localization strings, and header files. Native iOS apps are built on top of these frameworks and written in the Objective-C programming language, a superset of C language.

Besides the public frameworks, iOS also contains a set of private frameworks that are not allowed to be used in third-party apps. Even in public frameworks, there are some undocumented APIs (i.e., private APIs) that cannot be used by third-party apps. In fact, these private frameworks and APIs are reserved for the built-in apps and public frameworks. Apple ships all public and private frameworks as part of the iOS Software Development Kit (SDK). Third-party developers can find all these frameworks in their own development environment. It is worth noting that, since iOS 3.x, Apple has combined all frameworks into a single cache file called `dyld_shared_cache` in iOS devices to improve performance [21].

Moreover, the creation of dynamic libraries by third-party developers is not supported by the iOS SDK, which makes the public frameworks the only shared libraries to link in iOS apps. To prevent apps from dynamically loading private frameworks or unofficial libraries, some standard UNIX APIs are also considered as private by Apple, such as `dlopen` and `dlsym` that support runtime loading of libraries. During the app review process, linking to private frameworks or importing private APIs can directly result in app rejections from Apple App Store.

2.3 Code Reuse and ROP Attack

Reusing the code within the original program is an effective way to bypass DEP and code signing mechanism. Solar Designer first suggested return-to-libc [16], which reuses existing functions in a vulnerable program to implement attacks. Shacham et al. proposed the Return-Oriented Programming (ROP) exploitation technique in 2007 [44]. The core idea behind ROP attacks is to utilize a large number of instruction sequences ending with ret-like instructions (e.g., `ret` on x86 and `pop{pc}` on ARM) in the original program or other libraries to perform certain computation. Since attackers can control the data on the stack and ret-like instructions will change the execution flow according to the data on the stack, a crafted stack layout can chain these instruction sequences together. Figure 2 shows a simple ROP example that performs addition and storage operations on the ARM platform. Specifically, constant values `0xdeadbeaf` and `0xffffffff` are loaded to the registers `r1` and `r2` by the first two gadgets, respectively. Next, an addition operation is performed by the third gadget. At last, the addition result (`0xdeadbeae`) is stored on the stack by the fourth gadget.

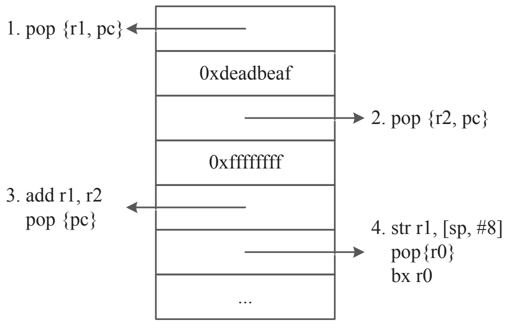


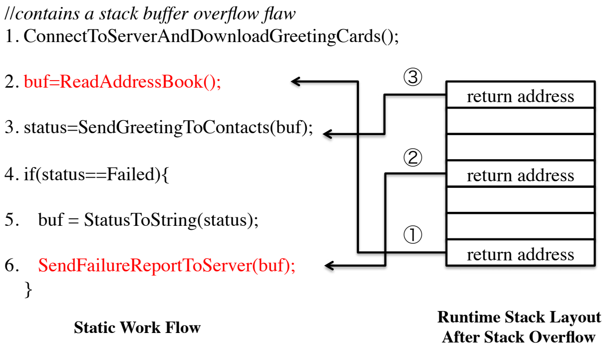
Figure 2: A ROP example

3 Attack Design

Before introducing the design of our attack scheme, we first discuss an example attack, which demonstrates the feasibility of such attacks and helps illustrate the design details in the rest of this section.

3.1 Motivating Example

Suppose the attacker's goal is to steal the user's contacts. To this end, the attacker first creates a normal app, a greeting card app for instance, which can download greeting cards from a remote server and then send them to the user's friends. The pseudo code in Figure 3 presents the workflow of the app, which requires access to user's address book and the network for legitimate reasons. However, direct abuses of these privileges to send the whole address book over the network can be easily detected. In fact, multiple systems (e.g., [17–19, 26]) have been proposed to detect malicious apps by identifying code paths or execution traces where sensitive data is first acquired and then transported out of the device, and we assume the app review process will also be able to detect and reject such apps.



```
//contains a stack buffer overflow flaw
1. ConnectToServerAndDownloadGreetingCards();

2. buf=ReadAddressBook();

3. status=SendGreetingToContacts(buf);

4. if(status==Failed){

5.    buf = StatusToString(status);

6.    SendFailureReportToServer(buf);
  }
```

Figure 3: Running Example

However, our example app (as shown in Figure 3) does not contain any feasible code path to leak the address book after reading it at line 2. As such, our example app appears to be compliant with Apple's privacy policy and can be expected to pass the app review.

To achieve the goal of stealing the user's contact while avoiding the direct approach that will guarantee rejection by App Store, the attacker instead hides vulnerabilities in the `ConnectToServerAndDownloadGreetingCards` function (line 1 in Figure 3). Subsequently, when the app runs on a victim's iOS device and tries to download greeting cards from the server controlled by the attacker, the server exploits the planted vulnerabilities to remotely manipulate the app's stack into the one shown on the right side of Figure 3. The contaminated stack layout will change the original control flows of the app. Instead of sequentially executing the statements from line 2 to line 6, the compromised app first reads the address book into a buffer (line 2 in Figure 3), and then directly invokes the `SendFailureReportToServer` function at line 6 to send the content of the buffer (i.e., address book) to the server. Finally, the app resumes the normal execution by returning the control back to line 3. Note that the attacker will avoid revealing the above behavior to Apple and only exploit the vulnerabilities after the app has passed the app review.

Malicious developers can freely design the vulnerabilities to bootstrap the attacks. For instance, the app can deliberately leak its memory layout information to the remote server so that ASLR is completely ineffective. Based on the memory layout information, attackers can launch attacks by reusing the exiting code inside the app. As a result, DEP and code signing cannot prevent the exploit. Furthermore, by using iOS private APIs, attackers can accomplish more sophisticated attacks, even though the app runs in the sandbox. In other words, once the app gets installed, existing security mechanisms on iOS will be of no defense against the attack.

3.2 Attack Scheme Overview

The high level idea of our attack scheme is very intuitive. The attacker creates a normal app in which he plants vulnerabilities and hides code gadgets along side the normal functionalities. After the app passes Apple's app review and gets installed on victims' devices, the attacker exploits the vulnerabilities and assembles the gadgets in a particular order to perform malicious operations.

For our attack to be successful, the planted vulnerabilities should allow us to defeat the ASLR, DEP, and code signing mechanisms in iOS, and at the same time be hardly detectable. To this end, we design an information leakage vulnerability through which the app deliberately leaks its partial runtime memory information to the remote attacker. Thus, the attacker can infer the locations of the pre-deployed gadgets, making ASLR useless. Next, we plant a buffer overflow vulnerability in the app through which the attacker can smash the stack layout and hijack the app's control flow. The carefully designed stack layout will chain together the gadgets to accomplish malicious tasks.

To avoid the vulnerabilities from being detected in the review process, the communication between the app and the server is encrypted, and all the vulnerabilities have special trigger conditions. Considering the fact that no source code but only the executable is provided to the review process, even if advanced vulnerability detection technologies like fuzz testing and dynamic symbolic execution are employed, it is unlikely for app review process to discover artificially planted and obscured vulnerabilities.

Finally, the hidden gadgets should be discretely distributed in the app and mingled with the normal functionalities, without explicit control flow or and data flow connections. To do this, we create a number of infeasible branches across the entire code space and hide gadgets under these infeasible branches. In addition, we organize the common operations useful for both legitimate and malicious functionalities into individual functional gadgets.

3.3 Bypassing ASLR via Information Leakage

The ASLR mechanism loads the app executable and other dynamic libraries at different random locations for each run, and this causes some difficulties in the process of chaining up our gadgets. However, since native apps are written in Objective-C, it is very easy to plant information leakage vulnerabilities to bypass ASLR and recover the addresses of our gadgets. In the following, we present two examples of how this can be achieved.

First, we can take advantage of an out-of-bounds memory access vulnerability to read a function pointer, and then send the value back to the remote server. Specifically, we can use a C code snippet similar to Figure 4. In this case, the app assigns the address of a public function to the function pointer in a C structure, and pretends to transmit the user name to the server. However, the server can control the size parameter of the function `memcpy` and is able to accurately trigger an out-of-bounds read. As a result, the address of the public function is leaked. Based on this address, we can infer the memory layout of corresponding executable file.

```
struct userInfo{
    char username[16];
    void* (*printName)(char*);
} user;
...
user.printName = publicFunction.
...
n = attacker_controllable_value; //20
memcpy(buf, user.username, n); //get function ptr
SendToServer(buf);
```

Figure 4: Information Disclosure Vulnerability I

Alternatively, we can take advantage of type confusion vulnerabilities and features of Objective-C objects to leak address information. Most objects in Objective-C programs inherit from a common class called `NSObject`. The first field of these objects points to a `Class` structure that stores information about the object's type, inheritance hierarchy, member methods, etc. These `Class` structures follow the same naming convention (i.e., a common prefix `_objc_class_$_`) and are stored at fixed offsets in the executable files. Using this information, we can also infer the address information of the entire executable file. Figure 5 demonstrates how this method works. First, we create an Objective-C object with the `myObject` pointer pointing to the object. After that, we convert `myObject` into an integer pointer by using explicit type-casting. Finally, by dereferencing the integer pointer, we copy the address value of the `Class` structure into the variable `UID`, and send it to the remote server.

```
//create an object
SomeClass* myObject = [[SomeClass alloc] init];
...
int UID = *(int*)myObject; //type confusion
...
SendToServer(UID);
```

Figure 5: Information Disclosure Vulnerability II

Since many of the malicious operations in Table 1 rely on private APIs, some discussion on how we invoke private APIs in our attack is in order. To this end, we need to be able to dynamically load private frameworks and locate private APIs, and we employ two special APIs, `dlopen()` and `dlsym()`. `dlopen()` is used to load and link a dynamic library specified by filename and return an opaque handle for the library. `dlsym()` is used to get the address of a symbol from a handle returned from `dlopen()`. These two functions are implemented in a library named `libdyld.dylib`. Since there is no evidence to show that the exported APIs in this library can be used by third-party apps, we should avoid directly referencing to any APIs in this library.

Fortunately, we find that both APIs are commonly used by public frameworks due to the need for dynamically loading shared libraries and obtaining the absolute addresses of symbols in the libraries. In particular, in order to support PIE (Position Independent Executable), public frameworks invoke imported APIs through trampoline functions. The trampoline functions here consist of a short sequence of instructions that first load the absolute address of a specific function from an indirect symbol table and then jump to that address. The indirect symbol table is initially set up by the linker at runtime. Therefore, if we can identify the trampolines for dlopen and dlsym in a public framework, our app can use the trampolines to indirectly invoke dlopen and dlsym.

The task of identifying usable trampolines is simple. With the help of a debugger, we set function breakpoints at dlopen and dlsym and run a test app on a physical device. When the debug session hits a breakpoint, we examine the call stack to find out the trampoline function and its relative offset to the beginning of the module. Thanks to the fact that ASLR on iOS work at the granularity of modules, we can always infer the addresses of these trampolines from the address of a public function in the same module leaked by our Jekyll app using the vulnerabilities described before. Finally, we note that trampolines for dlopen and dlsym can be found in many essential frameworks, such as UIKit and CoreGraphics.

3.4 Introducing New Execution Paths via Control-Flow Hijacking

A key design of our attack scheme is to dynamically introduce new execution paths that do not exist in the original app to perform the malicious operations. In order to achieve this, we plant a vulnerability in the Jekyll app, through which we can corrupt data on the stack and overwrite a function return address (or a function pointer). When the function returns, instead of returning to the original call site, the execution will proceed to a program point that is specified by the altered return address on the stack. Although iOS employs the Stack-Smashing Protector method to detect stack-based overflows, we can accurately overwrite the function return address without breaking the stack canary.

```
void vulnerableFoo(int i, int j){
    int buf[16];
    ...
    if(fakeChecks(i)) ;
        buf[i]= j; //overwrite return address
    ...
    return;
}
```

Figure 6: Control Flow Hijacking Vulnerability

Specifically, we use an out-of-bounds write vulnerability as shown in Figure 6 to hijack the control flow. In this case, both i and j are controlled by the attacker. Variable i is used to index a local integer array. Since the offset from the starting address of this local array to the memory slot for the function's return address is fixed, a carefully crafted i can overwrite the return address via an array element assignment without breaking the stack canary [10]. We can also add fake boundary checks on i in the function to prevent the vulnerability from being easily detected. The new return address stored in j points to a gadget that shifts the stack frame to a memory region storing data supplied by the attacker. After that, the new stack layout will chain the gadgets together. By using the existing code in the app, we can defeat DEP and code signing. Since our method for introducing new execution paths is essentially return-oriented-programming, interested readers are referred to [15] and [33] for the details of ROP on the ARM platform.

3.5 Hiding Gadgets

In traditional ROP attack scenarios, attackers have to search for usable gadgets from existing binary or libraries using the Galileo algorithm [44]. However, in our case, the attacker is also the app developer, who can freely construct and hide all necessary gadgets, either at the basic block or function level. This advantage makes our attacks significantly less difficult and more practical to launch than ROP attacks.

For the common functional units (such as converting a char* to NSString and invoking a function pointer), which are useful for both malicious and legit operations of the app, we implement them in individual functions. As a result, we can simply reuse such functions in our attack based on the return-to-libc like exploitation technique. For the special gadgets that are not easily found in existing code, we manually construct them by using ARM inline assembly code [32] and hide them in infeasible branches. In our Jekyll app, we have planted and hidden all gadgets that are required by traditional ROP attacks [15], such as memory operations, data processing (i.e., data moving among registers and arithmetic/logical operations), and indirect function calls.

To create the infeasible branches, we use the opaque constant technique [34]. For instance, in Figure 7 we set a variable to a non-zero constant value derived from a complicated calculation, and perform a fake check on that variable. Since the compiler cannot statically determine that the variable holds a constant value, it will generate code for both branches. As a result, we can reliably embed the gadgets using similar techniques.

Finally, we will conclude this section with a concrete example of our ROP attack. Figure 8 shows the original source code for dialing attack (see Section 4.2), which loads a framework into process memory, locates a private API called CTCallDial in the framework, and fi-

```
int i = Opaque_constant_calculation();
if(i == 0)
{  //hide a gadget in this branch
    asm volatile(
                "pop {r2}"
                "bx r2"
                );
}
```

Figure 7: Hide an indirect call gadget

nally invokes that function. Accomplishing the equivalent functionality through the ROP technique is very easy, because many function level gadgets are available in our Jekyll app. Specifically, we can find trampolines for `dlopen` and `dlsym` in public frameworks (see Section 3.3), and can also reuse existing code in our Jekyll app to implement the indirect call and the conversion from `char*` to `NSString` (the argument type of the function `CTCallDial` is `NSString`).

```
void* h = dlopen("CoreTelephony", 1);
void (*CTCallDial)(NSString*)=dlsym(h, "CTC-
allDial");
CTCallDial(@"111-222-3333");
```

Figure 8: Attack code for dialing

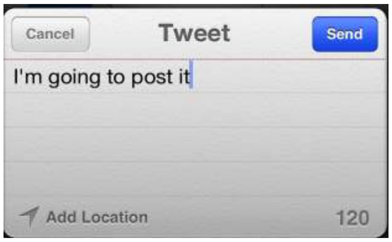
In addition to these function level gadgets, we also utilize a few simple basic block level gadgets that are used to prepare and pass function arguments, recover the stack pointer, and transfer the control back to the normal execution. For example, the first four arguments of a function on iOS are passed through the registers `R0-R3`. Before jumping into the target function, we can use a gadget like `pop{r0,r1,pc}` to set up the function's parameters. Such block level gadgets are ubiquitous in the existing code.

4 Malicious Operations

In this section, we introduce the malicious operations we can perform using Jekyll apps. We present how to post tweets and send email and SMS without the user's knowledge in Section 4.1, describe more private APIs based attacks in Section 4.2, and demonstrate Jekyll app's ability to exploit kernel vulnerabilities and attack other apps in Section 4.3 and Section 4.4.

4.1 Under the Hood: Posting Tweets and Sending Email and SMS

Since iOS 5.0, third-party apps are allowed to send Twitter requests on behalf of the user, by using the public APIs in a framework called `Twitter`. After setting the initial text and other content of a tweet, the public API called by the app will present a tweet view to the user, and let the user decide whether to post it or not, as shown in Figure 9. However, we find that the tweet view in Figure 9 can be bypassed by using private APIs, i.e., our app can post tweets without the user's knowledge. Next, we describe how we discover the private APIs needed for achieving this goal.



I'm going to post it

Figure 9: The default UI for a tweet view

Our intuition is that if we know the event handling function that is responsible for the "Send" button click event, our app can directly invoke that function to post the tweet, without the need to present the tweet view to the user.

To do this, we created a simple app that uses the `Twitter` framework to post tweets, and run the app in the debug model. We developed a dynamic analysis tool based on LLDB, a scriptable debugger in the iOS SDK, to log the function invocation sequence after the "Send" button is clicked. In the following, we will present some details about our tool.

In Objective-C, all object method invocations are dispatched through a generic message handling function called `objc_msgSend`. A method invocation expression in Objective-C like `[object methodFoo:arg0]` will be converted into a C function call expression like

```
objc_msgSend(object, "methodFoo:", arg0).
```

Moreover, iOS follows the ARM standard calling convention. The first four arguments of a function are passed through the registers `R0`-`R3`, and any additional arguments are passed through the stack. For the C function expression above, the arguments will be passed as follows: `R0` stores `object`, `R1` stores the starting address of the method name (i.e.,"`methodFoo:`"), and `R2` stores `arg0`.

Our dynamic analysis tool sets a conditional breakpoint at the `objc_msgSend` function. When the breakpoint is triggered after the user clicks the "Send" button, the tool logs the call stack, gets the target method name through the register `R1`, and retrieves the type information of the target object and other arguments (stored in the registers `R0`, `R2` and `R3`) by inspecting their `Class` structures (see Section 3.3).

According to the information in the log, we can easily identify the relevant Objective-C classes and private APIs for posting tweets. For instance, in iOS 6.x, we find that a tweet is composed through the method "setStatus:" in a class called SLTwitterStatus, and then is posted through the method "sendStatus:completion:" in a class called SLTwitterSession. Our Jekyll app will dynamically load the Twitter framework, create instances from these classes, and invoke private APIs to post tweets without the user's knowledge.

We also extended the idea to find critical private APIs for sending email and SMS. As in the case of posting Tweets, third-party apps are able to set the initial text and other content of an email or SMS, and present the email or SMS view to the user. In iOS 5.x, we successfully implemented the code to send email and SMS without the user's knowledge. Specifically, we find that an email is first composed by a method of the class MessageWriter, and then is sent to a service process via an inter-process communication (IPC) interface CPDistributedMessagingCenter. Eventually, the service process will send the email out. In the case of sending SMS, we find that, the content of an SMS is first converted into an XPC message, and the XPC message is subsequently passed to an XPC service (another kind of IPC interfaces in iOS) named com.apple.chatkit.clientcomposeserver.xpc. By using such private APIs, our Jekyll app is able to compose email and SMS objects, pass them to the corresponding service processes, and automatically send them without the user's knowledge. An independent study simultaneously reported how to send SMS in this manner; interested readers are referred to [20] for details.

However, in iOS 6, Apple introduced a new concept called remote view to enhance the security of email and SMS services. Specifically, a third-party app only passes the initial content of an email or SMS to the corresponding system services. These system service processes will then generate the message view, and let the user make further changes and final decision. Since the message view runs in a separate process, the third-party app is no longer able to invoke the handler function for the "Send" button click event.

4.2 Camera, Bluetooth, Device ID, and Dialing

The iOS developer community has accumulated extensive knowledge of using private APIs and proposed many attacks against jailbroken iOS devices. We integrated some previously known attacks into our Jekyll app. Since these attacks heavily use private APIs, any app that explicitly launches these attacks will most certainly be rejected by Apple. However, our Jekyll app can dynamically load the private frameworks and hide the invocations to private APIs, and successfully passes the App Review.

Next, we briefly introduce the private APIs that we utilized to achieve the following tasks without alerting the users: take photos, switch on/off bluetooth, steal the device identity information, and dial arbitrary numbers. The operations in this subsection work in both iOS 5.x and iOS 6.x.

- Abuse cameras. Our Jekyll app is able to stealthily turn on the camera in iOS devices to record videos without the user's knowledge; this can be achieved by creating and assembling the object instances of a set of classes such as AVCaptureDeviceInput and AVCaptureVideoDataOutput in the AVFoundation framework. Jekyll app can also extract every frame of a video stream and transfer the images back to the server.

- Switch Bluetooth. By using the APIs in a private framework BluetoothManager, our Jekyll app can directly manipulate the Bluetooth device, such as turning it on or off.

- Steal Device Identity. To obtain the device identity information, we take advantage of a private function called CTServerConnectionCopyMobileEquipmentInfo in the CoreTelephony framework. This function can return the device's the International Mobile Station Equipment Identity (IMEI), the International Mobile Subscriber Identity (IMSI), and the Integrated Circuit Card Identity (ICCID).

- Dial. By invoking the private API CTCallDial in the CoreTelephony framework, our Jekyll app can dial arbitrary numbers. Note that, this API supports to dial not only phone numbers, but also GSM service codes [3] as well as carrier-specific numbers. For instance, by dialing *21**number*#, Jekyll app can forward all calls to the victim's phone to another phone specified by *number*.

4.3 Exploiting Kernel Vulnerabilities

Since they run directly on iOS, native apps are able to directly interact with the iOS kernel and its extensions, making the exploitation of kernel vulnerabilities possible. Even though the sandbox policy limits third-party apps to only communicate with a restricted set of device drivers, and thus significantly reduces the attack surface for kernel exploitation, security researchers still managed to find vulnerabilities in this small set of device divers [49].

In our Jekyll app, we hide the gadgets that can enable us to communicate with the accessible device drivers. Specifically, Jekyll app can dynamically load a framework called `IOKit`, in which Jekyll app further locates the required APIs such as `IOServiceMatching`, `IOServiceOpen` and `IOConnectCallMethod` to create and manipulate connections to device drivers. Therefore, our Jekyll app provides a way for attackers to exploit kernel vulnerabilities. We demonstrate this by exploiting a kernel NULL pointer dereference vulnerability in iOS 5.x, disclosed in [49]. The exploitation of this vulnerability causes the iOS devices to reboot.

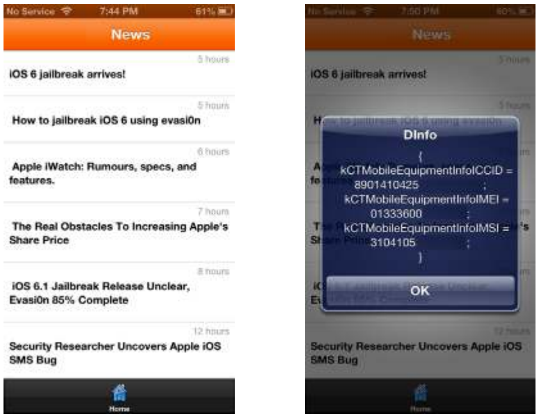
4.4 Trampoline Attack

Due to the sandboxing mechanism, iOS apps are restricted from accessing files stored by other apps. However, iOS provides a form of inter-process communication (IPC) among apps using URL scheme handlers. If an app registers to handle a URL type, other apps can launch and pass messages to this app by opening a URL scheme of that type. The `http`, `mailto`, `tel`, and `sms` URL schemes are supported by built-in apps in iOS. For example, an app opening a `http` URL will cause the built-in web browser Mobile Safari to launch and load the webpage. Since attackers can fully control the content in a URL request, our Jekyll app has the ability to attack other apps that have vulnerabilities when handling malformed URL requests.

In our proof-of-concept Jekyll app, we demonstrated an attack against Mobile Safari; in particular, we prepared a web page containing malicious JavaScript code that can trigger an unpatched vulnerability in Mobile Safari. Through our Jekyll app, we can force the victim's Mobile Safari to access this web page. Finally, Mobile Safari will crash when loading the webpage due to a memory error. JailbreakMe [1], a well-known jailbreak tool, completes the untethered jailbreak through exploiting a vulnerability in Mobile Safari and then exploiting a kernel vulnerability. If new vulnerabilities in Mobile Safari are disclosed by other researchers in the future, we can simply take advantage of these new vulnerabilities to launch similar powerful attacks.

5 Jekyll App Implementation

We have implemented a proof-of-concept Jekyll app based on an open source news client called News:yc [2]. The original News:yc app fetches news from a server, and allows the user to share selected news items through email. We modified News:yc in several places. First, we configured it to connect to a server controlled by us. Second, we planted vulnerabilities and code gadgets in the app. These vulnerabilities are triggerable by special news contents, and the code gadgets support all the malicious operations listed in Table 1. Third, we modified the app to use a secure protocol that provides authenticated and encrypted communication, so that the app client only accepts data from our server. In addition, the server was configured to deliver exploits only to the clients from specific IP addresses, which ensures that only our testing devices can receive the exploits. Figure 10.a shows the snapshot of the app.



iOS 6 jailbreak arrives!

How to jailbreak iOS 6 using evasi0n

Apple iWatch: Rumours, specs, and features.

The Real Obstacles To Increasing Apple's Share Price

iOS 6.1 Jailbreak Release Unclear, Evasi0n 85% Complete

Security Researcher Uncovers Apple iOS SMS Bug

DInfo

```
{
kCTMobileEquipmentInfoICCID =
8901410425          ;
kCTMobileEquipmentInfoIMEI =
01333600          ;
kCTMobileEquipmentInfoIMSI =
3104105          ;
}
```

a. The main UI of the app

b. After an attack, device identity is popped up for illustration purposes

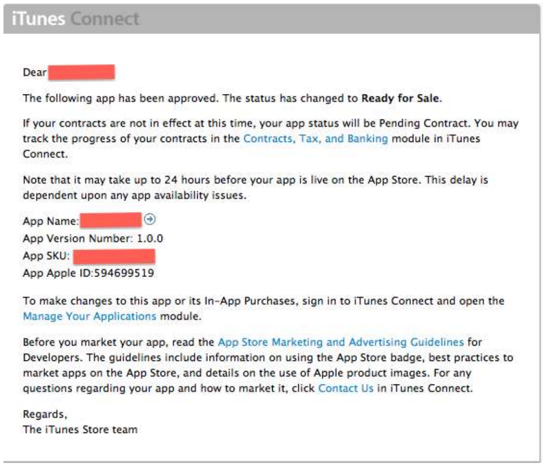
Figure 10: Snapshots of the app

We submitted the app to Apple and got Apple's approval after 7 days. Figure 11 shows the approval notification from Apple. Once the app was on App Store, we immediately downloaded it into our testing devices and removed it from App Store. We have data to show that only our testing devices installed the app. The server has also been stopped after we finished the testing.

The testing results are summarized in Table 1. By exploiting the vulnerabilities and chaining the planted gadgets, we can send email and SMS and trigger a kernel vulnerability on iOS 5.x, and post tweets, record videos, steal the device identity, manipulate bluetooth, dial arbitrary number, and attack Mobile Safari on both iOS 5.x and iOS 6.x. We show the attack of stealing device identity in Figure 10.b. We have made a full disclosure of our attack to Apple.

6 Related Work

Jailbreak, which obtains the root privilege and permanently disables the code signing mechanism, represents the majority of efforts to attack iOS [38]. Since jailbreak usually relies on a combination of vulnerabilities found in the iOS kernel, the boot loaders, and even the firmware, Apple and hackers have long played a cat-and-mouse game. However, due to Apple's increasing efforts



Dear [redacted]

The following app has been approved. The status has changed to **Ready for Sale**.

If your contracts are not in effect at this time, your app status will be Pending Contract. You may track the progress of your contracts in the Contracts, Tax, and Banking module in iTunes Connect.

Note that it may take up to 24 hours before your app is live on the App Store. This delay is dependent upon any app availability issues.

App Name: [redacted]
App Version Number: 1.0.0
App SKU: [redacted]
App Apple ID:594699519

To make changes to this app or its In-App Purchases, sign in to iTunes Connect and open the Manage Your Applications module.

Before you market your app, read the App Store Marketing and Advertising Guidelines for Developers. The guidelines include information on using the App Store badge, best practices to market apps on the App Store, and details on the use of Apple product images. For any questions regarding your app and how to market it, click Contact Us in iTunes Connect.

Regards,
The iTunes Store team

Figure 11: The approval notification from Apple

to secure iOS and keep fixing known bugs, it is becoming extremely difficult to find exploitable vulnerabilities in newer versions of iOS. Our attack does not try to achieve a jailbreak on iOS devices, instead, it takes advantage of the intrinsic incapability of the App Review process and the design flaws of iOS to deliver various types of malicious operations remotely, which cannot be trivially addressed via software updates. Note that, it is possible for Jekyll apps to take advantage of the vulnerabilities used by jailbreak tools to compromise iOS devices.

C. Miller [37] recently discovered a vulnerability in the iOS code signing mechanism, which allows attackers to allocate a writeable and executable memory buffer. He demonstrated that, by exploiting this vulnerability, a malicious app can safely pass the app review process if it generates malicious code only at runtime. However, Apple had instantly fixed the issue, and therefore, effectively blocked apps that use similar methods to load or construct malicious code during runtime.

In contrast, Jekyll apps do not hinge on specific implementation flaws in iOS. They present an incomplete view of their logic (i.e., control flows) to app reviewers, and obtain the signatures on the code gadgets that remote attackers can freely assemble at runtime by exploiting the planted vulnerabilities to carry out new (malicious) logic. In addition, the lack of runtime security monitoring on iOS makes it very hard to detect and prevent Jekyll apps. Considering that ROP attacks can achieve Turing-completeness [9] and automatic ROP shellcode generation is also possible [29, 43], the attack scheme in this paper significantly generalizes the threat in [37].

Return-Oriented Programming (ROP) [44], without introducing new instructions, carries out new logic that is not embodied in the original code. ROP and its variants [11, 29, 33, 36] allow attackers to create new control flows of a program at runtime via code gadget rearrangements, obviating the need for code injections that are prevented by DEP and code signing. Jekyll apps also employ code gadget rearrangements to alter runtime control flows—an idea inspired by ROP. However, our attack differs from ROP in both the assumption and the goal. Traditional ROP attack targets at programs that are out of the attacker's control and its power is often limited by the availability of useful code gadgets.

In comparison, Jekyll apps are created and later exploited by the same person, who has the ultimate control of the gadget availability. On the other hand, traditional ROP attackers have no concern about hiding potential code gadgets and their inter-dependencies, whereas we do so that Jekyll app can bypass existing and possible detections. Currently, we need to manually construct the ROP exploits that are responsible for chaining gadgets together. However, previous studies [29, 43] have demonstrated the possibility of automatically generating ROP shellcode on the x86 platform. We leave the automatic ROP shellcode generation for Jekyll apps as future work. In addition, M. Prati [40] proposed a way to hide ROP gadgets in open source projects with a purpose to evade the code audit of the projects. This implies that even Apple could audit the source code of third-party apps in the future, detecting the hidden gadgets is still quite challenging.

Jekyll apps also share a common characteristic with trojan and backdoor programs [13], that is, the malice or vulnerabilities of attacker's choice can be freely planted into the program, which later cooperates with the attacker when installed on a victim's device. In fact, Jekyll app can be deemed as an advanced backdoor app that stays unsuspicious and policy-abiding when analyzed during the app review process, but turns into malicious at runtime only when new control flows are created per attacker's command.

Thus far Apple's strict app publishing policies and review process [5] have helped keep malicious apps out of iOS devices [41]. Automated static analysis methods, such as [17, 26], were also proposed to assist the review process in vetting iOS apps. However, as we have demonstrated with our design and evaluation of Jekyll apps, malicious apps can easily bypass human reviewers and automatic tools if their malicious logic is constructed only at runtime. This demonstrates the limitations of Apple's current strategy that solely relies on app reviewing to find malicious apps and disallows any form of security monitoring mechanism on iOS devices.

7 Discussion

In this section, we discuss a number of possible countermeasures against Jekyll apps and analyze the effectiveness as well as the feasibility of these countermeasures.

7.1 Possible Detection at App Review Stage

Two possible directions that the app reviewers may pursue to detect Jekyll apps are: (i) discover the vulnerabilities we plant; (ii) identify the code gadgets we hide.

We emphasis that discovering software vulnerabilities using static analysis alone is fundamentally an undecidable problem [35], even without considering the powerful adversary in our attack who can arbitrarily obscure the presence of the vulnerabilities. Dynamic analysis based vulnerability detection approaches can also be easily defeated by using complicated trigger conditions and encrypted input data. We argue that the task of making all apps in App Store vulnerability-free is not only theoretically and practically difficult, but also quite infeasible to Apple from an economic perspective because such attempts will significantly complicate the review tasks, and therefore, prolong the app review and approval process that is already deemed low in throughput by third-party app developers.

To simplify the engineering efforts, our current implementation of Jekyll app directly includes some code gadgets in an isolated fashion (i.e., unreachable from program entry points), essentially leaving them as dead code that may be detectable and in turn removed during app review process. However, given our freedom to craft the app, it is totally possible to collect all gadgets from the code that implements the legitimate functionalities of the app, without the need to hide extra gadgets as dead code.

In summary, even though the hidden vulnerabilities and gadgets might take unusual forms comparing with regular code, accurately detecting Jekyll apps (e.g., based on statistical analysis) is still an open challenge. Thus, detecting Jekyll apps in App Review process via vulnerability discovery or gadgets identification is not a feasible solution.

7.2 Possible Mitigation through Improved or New Runtime Security

Generally, improving the existing security mechanisms or introducing more advanced runtime monitoring mechanisms can limit Jekyll apps' capability to perform malicious operations. However, completely defeating Jekyll apps is not easy.

- A natural idea to limit Jekyll apps is to technically prevent third-party apps from loading private frameworks or directly invoking private APIs. However, Jekyll apps do not have to dynamically load private frameworks. As we discussed, since many public frameworks rely on these private frameworks, Jekyll apps can reasonably link to these public frameworks so that certain private frameworks will also be loaded into the process space by the system linker. A more strict execution environment like Native Client [50] can help prevent the apps from directly invoking private APIs by loading private frameworks into a separate space and hooking all invocations. However, since iOS public and private frameworks are tightly coupled, applying such a mechanism to iOS is quite challenging.

- Fine-grained ASLR such as [27, 39, 46] can greatly reduce the number of gadgets that we can locate during runtime even with the help of the planted information leakage vulnerabilities. Although expanding the scale and refining the granularity of the information leakage can help obtain a detailed view of the memory layout, Jekyll apps may lose the stealthiness due to the increased exposure of the vulnerabilities and increased runtime overhead.

- A fine-grained permission model, sandbox profile, or user-driven access control policy [28,42] can also help limit the damages done by Jekyll apps. However, simply using Android-like permission system will not be an unsurmountable obstacle to Jekyll apps. As long as a Jekyll app can reasonably require all permissions, it can still carry out certain attacks successfully. A user-driven access control model [28, 42] also cannot stop Jekyll apps from abusing the access already granted and attacking other apps or the kernel. Take the greeting card app in Section 3.1 as an example. After the user allows the greeting card app to access the address book, it is very hard to prevent the app from leaking the information.

- Since Jekyll apps heavily reply on control flow hijacking vulnerabilities, advanced exploit prevention techniques such as CFI [6] may effectively limit Jekyll apps. CFI ensures that runtime control-flow transfers conform with the rules that are derived from the static analysis of the program and the constraints inferred from the execution context. MoCFI [14] and PSiOS [47] brought the same idea to iOS with a caveat that they require jailbroken devices. Despite its high performance overhead and low adoption rate in practice, CFI is generally deemed effective against conventional ROP attacks, which partially inspired the design of Jekyll apps. In principle, if properly implemented and deployed on iOS, CFI can significantly increase the complexity of designing Jekyll apps and force attackers to trade code flexibility for success. Although skilled attackers presumably can either employ very systematic non-control data attacks [12] to perform malicious operations or use function-level gadgets to bypass

CFI, given their freedom to craft the gadgets in our attack, they may have to sacrifice the stealthiness of Jekyll apps to some extent due to the increased distinguishability caused by such techniques.

- Type-safe programming languages like Java are immune to low-level memory errors such as buffer overflows. Thus, if we can enforce that third-party apps be developed in type-safe programming languages, we can prevent the problems of planted control flow hijacking or information leakage vulnerabilities in the apps.

In summary, we advocate the official support for runtime security monitoring mechanisms on iOS. Our design of Jekyll apps intends to motivate such mechanisms, which can protect iOS against advanced attacks and ensure that the app review practice and regulations receive their maximum efficacy.

8 Conclusion

In this paper, we presented a novel attack scheme that can be used by malicious iOS developers to evade the mandatory app review process. The key idea is to dynamically introduce new execution paths that do not exist in the app code as reviewed by Apple. Specifically, attackers can carefully plant a few artificial vulnerabilities in a benign app, and then embed the malicious logic by decomposing it into disconnected code gadgets and hiding the gadgets throughout the app code space. Such a seemingly benign app can pass the app review because it neither violates any rules imposed by Apple nor contains functional malice. However, when a victim downloads and runs the app, attackers can remotely exploit the planted vulnerabilities and in turn assemble the gadgets to accomplish various malicious tasks.

We demonstrated the versatility of our attack via a broad range of malicious operations. We also discussed our newly discovered private APIs in iOS that can be abused to send email and SMS and post tweets without the user's consent.

Our proof-of-concept malicious app was successfully published on App Store and tested on a controlled group of users. Even running inside the iOS sandbox, the app can stealthily post tweets, take photos, gather device identity information, send email and SMS, attack other apps, and even exploit kernel vulnerabilities.

Acknowledgements

We thank our shepherd Benjamin Livshits and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation under grants no. CNS-1017265 and no. CNS-0831300, and the Office of Naval Research under grant no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

References

[1] JailbreakMe. http://www.jailbreakme.com/.

[2] News:yc, the open source news client for iOS. https://github.com/Xuzz/newsyc.

[3] Unstructured supplementary service data. http://en.wikipedia.org/wiki/Unstructured_Supplementary_Service_Data.

[4] Apple's worldwide developers conference keynote address, June 2010. http://www.apple.com/apple-events/wwdc-2010/.

[5] Apple's app store review guidelines, 2013. https://developer.apple.com/appstore/resources/approval/guidelines.html.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, 2005.

[7] A. Bednarz. Cut the drama: Private apis, the app store & you. 2009. http://goo.gl/4eVr4.

[8] D. Blazakis. The apple sandbox. In *Blackhat DC*, Jan 2011.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, Alexandria, VA, USA, 2008.

[10] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, 56(5), 2000.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, Oct 4-8, 2010.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 12–12, 2005.

[13] S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A framework to eliminate backdoors from response-computable authentication. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[14] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[15] L. Davi, A. Dmitrienkoy, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on arm. Technical Report HGI-TR-2010-002, System Security Lab, Ruhr University Bochum, Germany, 2010.

[16] S. designer. *Bugtraq*, Aug, 1997. return-to-libc attack.

[17] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, February 2011.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[19] J. Engler, S. Law, J. Dubik, and D. Vo. ios application security assessment and automation: Introducing sira. In *Black Hat USA*, LAS VEGAS, 2012.

[20] K. Ermakov. Your flashlight can send sms. http://blog.ptsecurity.com/2012/10/your-flashlight-can-send-sms-one-more.html, Oct 2012.

[21] S. Esser. Antid0te 2.0 -ASLR in iOS. In *Hack In The Box(HITB)*. Amsterdam, May 2011.

[22] S. Esser. ios kernel exploitation. In *Black Hat USA*, LAS VEGAS, 2011.

[23] D. ETHERINGTON. iphone app contains secret game boy advance emulator, get it before it's gone. March 2013. `http://goo.gl/OGyc0`.

[24] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 3–14, 2011.

[25] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. H. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on ios with approved third-party applications. In *11th International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Banff, Alberta, Canada, June 2013.

[26] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[27] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 571–585, San Francisco, CA, USA, May 2012.

[28] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *The Web 2.0 Security & Privacy Workshop (W2SP)*, 2010.

[29] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, Aug, 2009.

[30] iOS Market Statistics, 2012. `http://goo.gl/LSK7I/`.

[31] iOS Security, May 2012. `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`.

[32] H. Kipp. Arm gcc inline assembler cookbook. 2007. `http://www.ethernut.de/en/documents/arm-inline-asm.html`.

[33] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009.

[34] C. Kruegel, E. Kirda, and A. Moser. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, Florida, USA, Dec, 2007.

[35] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001.

[36] K. Lu, D. Zou, W. Wen, and D. Gao. Packed, printable, and polymorphic return-oriented programming. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, California, USA, September 2011.

[37] C. Miller. Inside ios code signing. In *Symposium on Security for Asia Network (SyScan)*, Taipei, Nov 2011.

[38] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. Wiley, 1 edition edition, May 2012.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 601–615, San Francisco, CA, USA, May 2012.

[40] M. PRATI. *ROP gadgets hiding techniques in Open Source Projects*. PhD thesis, University of Bologna, 2012.

[41] P. Roberts. Accountability, not code quality, makes ios safer than android. April 2012. `http://goo.gl/ZaXhj`.

[42] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2012.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of USENIX Security*, San Francisco, CA, USA, 2011.

[44] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, Oct. 29-Nov. 2,2007.

[45] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington DC, USA, 2004.

[46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, Raleigh, NC, USA, Oct, 2012.

[47] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: Bring your own privacy & security to ios devices. In *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, May 2013.

[48] Wikipedia. iOS jailbreaking. 2013. `http://en.wikipedia.org/wiki/IOS_jailbreaking`.

[49] H. Xu and X. Chen. Find your own ios kernel bug. In *Power of Community (POC)*, Seoul, Korea, 2012.

[50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[51] D. A. D. Zovi. ios 4 security evaluation. In *Blackhat USA*, Las Vegas, NV, Aug 2011.