On the Feasibility of Large-Scale Infections of iOS Devices

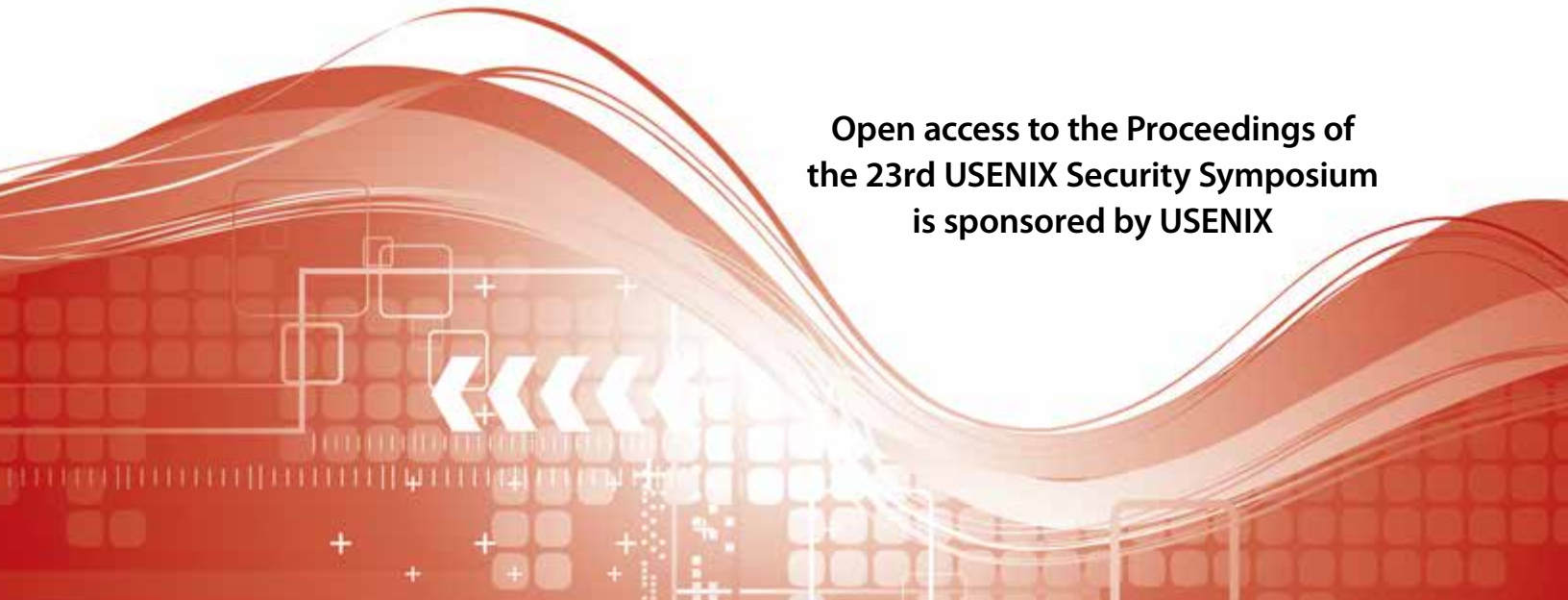
Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang_tielei

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7



Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

On the Feasibility of Large-Scale Infections of iOS Devices

Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee

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

{*tielei.wang, yeongjin.jang, yizheng.chen, pchung, billy, wenke*}*@cc.gatech.edu*

Abstract

While Apple iOS has gained increasing attention from attackers due to its rising popularity, very few large scale infections of iOS devices have been discovered because of iOS' advanced security architecture. In this paper, we show that infecting a large number of iOS devices through botnets is feasible. By exploiting design flaws and weaknesses in the iTunes syncing process, the device provisioning process, and in file storage, we demonstrate that a compromised computer can be instructed to install Apple-signed malicious apps on a connected iOS device, replace existing apps with attacker-signed malicious apps, and steal private data (e.g., Facebook and Gmail app cookies) from an iOS device. By analyzing DNS queries generated from more than half a million anonymized IP addresses in known botnets, we measure that on average, 23% of bot IP addresses demonstrate iOS device existence and Windows iTunes purchases, implying that 23% of bots will eventually have connections with iOS devices, thus making a large scale infection feasible.

1 Introduction

As one of the most popular mobile platforms, Apple iOS has been successful in preventing the distribution of malicious apps [23, 32]. Although botnets on Android and jailbroken iOS devices have been discovered in the wild [40, 42, 43, 45, 54], large-scale infections of non-jailbroken iOS devices are considered extremely difficult for many reasons.

First, Apple has powerful revocation capabilities, including removing any app from the App Store, remotely disabling apps installed on iOS devices, and revoking any developer certificate. This makes the removal of malicious apps relatively straightforward once Apple notices them.

Second, the mandatory code signing mechanism in iOS ensures that only apps signed by Apple or certified by Apple can be installed and run on iOS devices. This significantly reduces the number of distribution channels of iOS apps, forcing attackers to have their apps signed by a trusted authority.

Third, the Digital Rights Management (DRM) technology in iOS prevents users from sharing apps among arbitrary iOS devices, which has a side effect of limiting the distribution of malicious apps published on the App Store. Although recent studies show that malicious apps can easily bypass Apple's app vetting process and appear in the Apple App Store [26, 36, 51], lacking the ability to self-propagate, these malicious apps can only affect a limited number of iOS users who *accidentally* (e.g., by chance or when tricked by social-engineering tactics) download and run them. Specifically, to run an app signed by Apple, an iOS device has to be authenticated by the Apple ID that purchased the app. For example, suppose we use an Apple ID_A to download a copy of a malicious app from the App Store and later we install this copy on an iOS device that is bound to Apple ID_B. This copy cannot run on the iOS device that is bound to Apple ID_B because of the failure of DRM validation. On iOS 6.0 or later, when launching this app, iOS will pop up a window (Figure 1) to ask the user to re-input an Apple ID and a password. If the user cannot input the correct Apple ID (i.e., Apple ID_A) and the corresponding password, iOS refuses to run the app.

In this paper, we show that despite these advanced security techniques employed by iOS, **infecting a large number of non-jailbroken iOS devices through botnets is feasible**. Even though iOS devices are designed for mobile use, they often need to be connected to personal computers via USB or Wi-Fi for many reasons, such as backup, restore, syncing, upgrade, and charging. We find that the USB and Wi-Fi communication channels between iOS devices and computers are not well protected. Consequently, a compromised computer (i.e., a bot) can be easily instructed to install malicious apps onto its connected iOS devices and gain access to users'

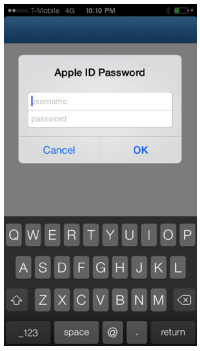


Figure 1: User attempting to run an app downloaded by a different Apple ID on his iOS device needs to first enter the correct Apple ID and password.

private data without their knowledge. In this paper, connected iOS devices refer to those that are plugged into a computer through USB cable or have Wi-Fi syncing enabled. Note that if Wi-Fi syncing is enabled, the iOS device will automatically sync with a paired computer when they are on the same network.

The feasibility of a large scale infection is facilitated by two main problems that we have discovered in our research. The first is a previously unknown design flaw in the iTunes syncing mechanism (including both USB and Wi-Fi based syncing), which makes the iTunes syncing process vulnerable to Man-in-the-Middle (MitM) attacks. By exploiting this flaw, attackers can first download an Apple-signed malicious app (e.g., a Jekyll app [51]) using their Apple ID and then remotely instruct a compromised computer to install the attacker's copy on a connected iOS device, completely bypassing DRM checks. In other words, an attacker can have a malicious app of his own choosing run on a user's iOS device without the user ever seeing an authentication pop-up window.

Coupled with botnet infrastructures, this exploit enables large scale delivery of arbitrary Apple-signed apps. This has two serious security implications. First, it challenges the common belief that the Apple App Store is the sole distributor of iOS apps. Instead of relying on tricking a user into downloading apps from the App Store, attackers can now push copies of their app onto a victim's device. Even if an app has been removed from the App Store, attackers can still deliver it to iOS users. Second, this expolit challenges the common belief that the installation of iOS apps must be approved by the user. Attackers can surreptitiously install any app they downloaded onto victim's device.

The second security issue we discovered is that an iOS device can be stealthily provisioned for development through USB connections. This weakness allows a compromised computer to arbitrarily remove installed third-party apps from connected iOS devices and install any app signed by attackers in possession of enterprise or individual developer licenses issued by Apple. This weakness leads to many serious security threats. For example, attackers can first remove certain targeted apps (such as banking apps) from iOS devices and replace them with malicious apps that look and feel the same. As a result, when a victim tries to run a targeted app, they actually launch the malicious app, which can trick the user to re-input usernames and passwords. We originally presented this attack [31] in 2013 and Apple released a patch in iOS 7 that warns the user when connecting iOS devices to a computer *for the first time*. However, this patch does not protect iOS devices from being stealthily provisioned by a compromised computer that the user already trusts.

In addition to injecting apps into iOS devices, attackers can also leverage compromised computers to obtain credentials of iOS users. Specifically, since many iOS developers presume that the iOS sandbox can effectively prevent other apps from accessing files in their apps' home directories, they store credentials in plaintext under their apps' home directories. For example, the Starbucks app was reported to save usernames and passwords in plaintext. Starbucks thought the possibility of a security exploit to steal the plaintext passwords was "very far fetched" [8]. However, once an iOS device is connected to a computer, all these files are accessible by the host computer. Consequently, malware on the computer can easily steal the plaintext credentials through a USB connection. In our work, we found that the Facebook and Gmail apps store users' cookies in plaintext. By stealing and reusing the cookies from connected iOS devices, attackers can gain access to the victims' accounts remotely.

While it is known that a host computer can partially access the file system of a connected iOS device, we point out that it leads to security problems, especially when attackers control a large number of personal computers. Considering that there are many apps dedicated to iOS, this problem may allow attackers to gain credentials that are not always available on PCs.

To quantitatively show that botnets pose a realistic threat to iOS devices, we also conduct a large scale measurement study to estimate how many compromised computers (i.e., bots) could connect with iOS devices. Intuitively, given the immense popularity of iOS devices and compromised Windows machines, we presume that many people are using iOS devices connected to compromised computers. However, to the best of our knowledge, there exists no previous work that can provide large scale measurement results.

By analyzing DNS queries generated from 473,506

Infection Type	Root Cause	Connection Type
Install malicious apps signed by Apple	Man-in-the-Middle attacks against syncing	USB or Wi-Fi
Install malicious apps signed by attackers	Stealthy provisioning of devices	USB
Steal private data (e.g., Facebook and Gmail apps's cookies)	Insecure storage of cookies	USB

Table 1: Infection Summary.

anonymized IP addresses[1] that were involved in known botnets on 10/12/2013 in 13 cities of two large ISPs in the US, we identified 112,233 IP addresses that had App Store purchase traffic issued by iTunes on Windows, as well as network traffic generated by iOS devices. This implies that the iOS users in the 112,233 home networks were purchasing items in the App Store from compromised Windows PCs on the same day. We make the following assumption: if iTunes is installed on a user's personal computer and is also used to purchase some item from the App Store, the user will eventually connect his or her iOS device(s) to it either via USB or Wi-Fi. Based on this assumption, we estimate that iOS devices in the 112,233 IP addresses could be infected via a connected computer. In other words, 23% of bots in our measured botnet could be used to infect iOS devices.

A broader implication of our study is that it may raise concerns about the security of mobile two-factor-authentication schemes [13]. In such schemes, a mobile phone is used as a second factor of authentication for transactions initiated on a potentially compromised PC. A fundamental assumption made by such schemes is that the "two factors" (i.e., the PC and the phone) are very hard to be reliably and simultaneously compromised (and linked to the same user) by an adversary. This assumption is reasonable if the PC and the phone are exposed to independent attack vectors. However, as shown in this paper, since a large number of users would connect mobile phones to their PCs, the PC itself becomes an attack vector to the phone. As such, the aforementioned assumption becomes dubious. An attacker who already controls the PC can now use it as a stepping stone to inject malware into the phone, and thus can control both factors. As a result, the attacker can easily launch attacks described in [21, 39] to defeat mobile two-factor-authentication schemes[2].

In summary, the main contributions of our work are:

- We discover a design flaw in the iTunes syncing process, and present a Man-in-the-Middle attack that enables attackers to run any app downloaded by their Apple ID on iOS devices that are bound to different Apple IDs, bypassing DRM protections. Based on the MitM attack, we present a way to deliver Apple-signed malicious apps such as Jekyll apps to iOS users.

- We point out the security implications of the stealthy provisioning process and insecure credential storage, and demonstrate realistic attacks, such as replacing installed apps in the iOS device with malicious apps and stealing authentication cookies of the Facebook and Gmail apps.

- We show that a large scale infection of iOS devices is a realistic threat and we are the first to show quantitative measurement results. By measuring iTunes purchases and iOS network traffic generated from IP addresses involved in known botnets, we estimate that on average, 23% of all bot population have connections with iOS devices.

Table 1 summaries the attacks. We have made a full disclosure to Apple and notified Facebook and Google about the insecure storage of cookies in their apps. Apple acknowledged that, based on our report, they have identified several areas of iOS and iTunes that can benefit from security hardening.

The rest of the paper is organized as follows. In Section 2, we demonstrate installation of Apple-signed malicious apps without violating DRM checks. In Section 3, we demonstrate replacing targeted apps with attacker-signed malicious apps. In Section 4, we demonstrate theft of private data (e.g., Facebook and Gmail apps' cookies) from iOS devices. In Section 5, we describe our measurement techniques that indicate large scale infection is feasible. Finally, we provide related work, discussion, and a conclusion.

2 Delivery of Apple-Signed Malicious Apps

This section discusses how a compromised computer can be instructed to install Apple-signed malicious apps on iOS devices. We explore the iOS DRM technology and iTunes syncing process in Section 2.1, present a Man-in-the-Middle attack in Section 2.2, and discuss how to bypass iOS DRM validations in Section 2.3.

[1]The original IP addresses were hashed into anonymized client IDs in our dataset. We performed our measurement over five days of DNS query data, and we used statistics from one day as an example here. We use IP address and client ID interchangeably.

[2]The ultimate defeat of mobile two-factor-authentication schemes will depend on what capabilities the injected mobile malware has. We further discuss it in Section 6.

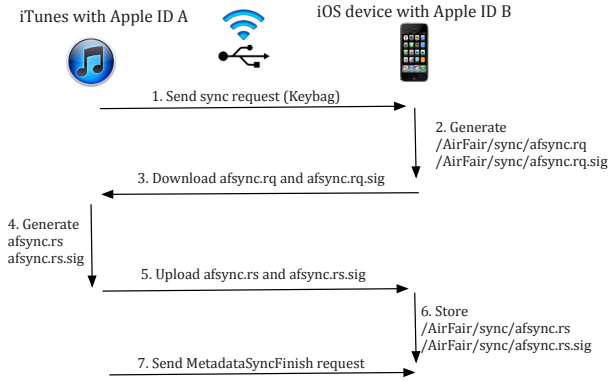


Figure 2: iTunes can sync apps to an iOS device with a different Apple ID.

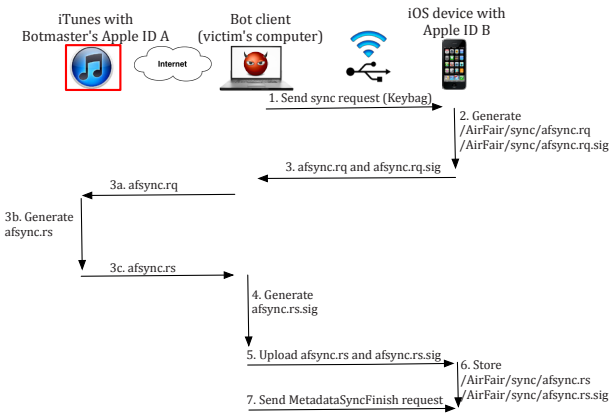


Figure 3: The Man-in-the-Middle against the syncing.

2.1 Fairplay DRM and iTunes Syncing

Apple utilizes a DRM (Digital Rights Management) technology called Fairplay to prevent piracy of iOS apps. All apps in the App Store are encrypted and signed by Apple. To run an app downloaded from the App Store, iOS will 1) verify an app's code signature, 2) perform DRM validation and decrypt the executable file, and 3) run the decrypted code. As a result, a copy of an app purchased by Apple ID_A cannot run on other iOS devices bound to other Apple IDs, because of the failure of DRM validation in step 2.

Although Apple does not disclose any technical details about the Fairplay DRM technology, we found a way to bypass it based on the following key observations.

- **Observation 1: Different Apple IDs will receive the same encrypted executable files for different copies of the same app**. After purchasing an app, an iOS user will receive a file with the `.ipa` extension from the App Store. The `ipa` file is in compressed zip format. We can retrieve the contents of an app package by decompressing the `ipa` file. The following shows the typical structure of an app directory (taking the Twitter app as an example).

```
/iTunesArtwork
/iTunesMetadata.plist
/Payload/Twitter.app/Twitter
...
/Payload/Twitter.app/SC_Info/Twitter.sinf
/Payload/Twitter.app/SC_Info/Twitter.supp
...
```

  Although the whole `ipa` package is unique for each Apple ID, we noticed that the encrypted executable files inside these `ipa` files are the same. Different copies of the same app purchased by different Apple IDs sharing same encrypted executable files implies that **the final decryption of the executables is irrelevant to Apple IDs**[3].

- **Observation 2: Apps purchased by different Apple IDs can run on the same iOS device under certain circumstances**. We found that iTunes can sync apps in its app library to iOS devices through USB or Wi-Fi connections, even if the iOS devices are bound to different Apple IDs. Specifically, when an iOS device with Apple ID_B is connected to iTunes with Apple ID_A, iTunes can still sync apps purchased by Apple ID_A to the iOS device, and authorize the device to run the apps. As a result, apps purchased by both Apple ID_A and Apple ID_B can run on the iOS device.

In particular, we reverse engineered the iTunes authorization process, i.e., how iTunes authorizes an iOS device with a different Apple ID to run its apps. We briefly present the workflow here.

First, iTunes sends a `Keybag` sync request to the iOS device (Step 1 in Figure 2). We refer the readers to [29] for detailed use of "Keybags" in iOS. For example, a keybag named Escrow is used for iTunes syncing and allows iTunes to back up and sync without requiring the user to enter a passcode.

Next, the iOS device generates an authorization request file `/AirFair/sync/afsync.rq` and corresponding signature file `/AirFair/sync/afsync.rq.sig` (Step 2 in Figure 2). Upon retrieving these two files from the iOS device (Step 3), iTunes generates an authorization response file `afsync.rs` and corresponding signature file `afsync.rs.sig` (Step 4).

iTunes then uploads the authorization response and signature files (`afsync.rs` and `afsync.rs.sig`) to the iOS device (Step 5). The iOS device stores the two files

[3] We further explain this in Section A.1

in the directory `/AirFair/sync/` and updates its internal state (Step 6).

Finally, iTunes sends a request to the iOS device to finish the syncing process (Step 7). After that, all apps in the iTunes app library can directly run on the iOS device without showing the pop-up window similar to Figure 1, even though iTunes and the iOS device are bound to different Apple IDs.

2.2 Remote Authorization

By reverse engineering the iTunes executables, we identified the functions in iTunes that are used to generate the `afsync.rs` and `afsync.rs.sig` (i.e., Step 4 in Figure 2). Based on these findings, we realized that by launching a Man-in-the-Middle-style attack, iTunes can remotely authorize an iOS device, without requiring physical connections between iTunes and the iOS device.

Figure 3 shows the remote authorization process in which iTunes is running on a remote computer (iTunes with Botmaster's Apple ID) and an iOS device is connected to a local computer (i.e., the bot client) through a USB cable or Wi-Fi connection. The local computer acts as a middleman.

First, the local computer, as instructed by the botmaster, sends a `Keybag` syncing request to a connected iOS device (Step 1 in Figure 3). After receiving the request, the iOS device generates the authorization request and signature file (Step 2), and transfers them to the local computer (Step 3). However, unlike the traditional iTunes authorization process, the local computer does not directly produce the authorization response file. Instead, it sends the authorization request file `afsync.rq` to a remote computer where iTunes is running (Step 3a). Upon receiving `afsync.rq`, the remote computer can force its local iTunes to handle the authorization request file as if it were from a connected iOS device, generate the authorization response file `afsync.rs` (Step 3b), and then send `afsync.rs` back (Step 3c).

Next, the local computer further produces the signature file `afsync.rs.sig` by using local iTunes code (Step 4). Note that the signature file is a keyed hash value of the response file using the connection session ID as the key. The signature file could also be generated by the remote iTunes if the local computer transfers the connection session ID to the remote iTunes in Step 3a. Furthermore, the local computer uploads `afsync.rs` received from the remote iTunes and `afsync.rs.sig` to the connected iOS device (Step 5). Step 6 and Step 7 in Figure 3 are the same as those in Figure 2.

The end result is that the iOS device connected to a local computer obtains authorization to run apps purchased by the iTunes instance running on a remote computer.

2.3 Delivery of Jekyll Apps

Background. Our previous work [51] demonstrated that malicious third-party developers can easily publish malicious apps on the App Store. It also implemented a proof-of-concept malicious app named Jekyll that can carry out a number of malicious tasks on iOS devices, such as posting tweets and dialing any number. Other research also demonstrated malicious keylogger apps on iOS 7.0.6 [55]. However, a key limitation of these apps is that attackers have to passively wait for iOS users to download the apps and thus affect only a limited number of iOS users.

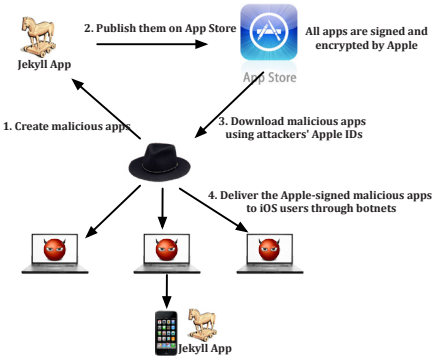


Figure 4: Deliver Jekyll apps to iOS devices.

Delivery Process. Figure 4 illustrates the high level workflow of the attack. First, attackers create Jekyll-like malicious apps using methods proposed in [51] and publish them on the App Store (Step 1 & 2 in Figure 4). Next, attackers download Jekyll-like apps from the App Store using Apple IDs under their control (Step 3). Finally, attackers deliver the downloaded apps to compromised computers and instruct them to install the downloaded apps to connected iOS devices.

Since Jekyll [51] has been removed from the App Store, we reused the copy of the app downloaded by our testing accounts. By using the remote authorization technique we described in Section 2.2, we successfully made our copy of Jekyll run on iOS devices bound to different Apple IDs without triggering DRM violations. The attack demonstrates that even if an app has been removed from the App Store, attackers can still distribute their own copies to iOS users. Although Apple has absolute control of the App Store, attackers can leverage MitM to build a covert distribution channel of iOS apps.

3 Delivery of Attacker-Signed Apps

In this section, we present how the USB interface can be exploited to install arbitrary apps that are signed by attackers. The iOS mandatory code signing mechanism ensures that only apps signed by trusted authorities can

run on iOS devices. Apple allows developers to install apps into iOS devices through a process called device provisioning, which delegates code signing to iOS developers. This was originally intended for developers to either test their apps on devices or for enterprises to do in-house app distribution. Unfortunately, we found that this process can be abused to install malicious apps into iOS devices.

3.1 Provisioning Process

Preparation. A provisioning profile is a digital certificate that establishes a chain of trust. It describes a list of iOS devices that are tied to an Apple ID, using the Unique Device Identifier (UDID) of each device. After sending the UDID of an iOS device to Apple, a provisioning profile is produced for installation on the device. Once installed, apps created and signed by the Apple ID that created the provisioning profile can be successfully executed in a provisioned iOS device. Although a device's UDID is considered sensitive information, it is straightforward for compromised machines to obtain the UDID of a connected iOS device because an iOS device exposes its UDID through the USB device descriptor header field.

Installing Provisioning Profiles. Conventionally, the provisioning process is transparently done when one is using Xcode (Apple's IDE). However, we found that the installation of provisioning profiles can also be done by directly sending requests to a service running on iOS devices called `com.apple.misagent`, launched via the `lockdownd` service [33]. Specifically, by crafting requests of the following key value pairs encapsulated in plist format [1]: <`MessageType`, `Install`>, <`Profile`, the provisioning profile to be installed>, and <`ProfileType`, `Provisioning`>, we can stealthily install a provisioning profile in a USB-connected iOS device [33].

Installing and Removing Apps. The removal of an app is done by issuing an `Uninstall` command and app-id to a service on the device called `com.apple.mobile.installation_proxy`. The installation of an app is done by issuing an `Install` command and app-id to the same service, with the addition of the path to where the app package resides on the computer [33]. Note that using the `com.apple.mobile.installation_proxy` service to install or remove apps also works for non-provisioned iOS devices.

3.2 Attack Examples

As we will present in Section 5.7, we discovered that 4,593 (4%) of 112,233 potential victims queried mobile banking domains in a day (10/12/2013), implying that these devices are likely to have mobile banking apps installed. We implemented a proof-of-concept program that can check whether a plugged-in iOS device has banking apps installed and replaces them with malicious apps that look and feel the same, but trick the user to re-input usernames and passwords.

4 Stealing Credentials

In this section, we demonstrate that in addition to injecting apps into iOS devices, attackers can also leverage bots to steal credentials of iOS users.

4.1 Background

Cookie as Credential. Due to the common architecture of backend servers, the most frequently used credential types in the iOS app model are HTTP cookies [17] and OAuth Tokens [25, 27]. We focus on cookies, as they generally provide a full set of privileges for a particular app, whereas OAuth credentials are utilized to provide limited permissions.

Many iOS apps (such as Facebook and Gmail) use cookies as their authentication tokens. After first login to these apps, a cookie is generated by the server, transmitted to the device, and stored locally. Later, this cookie is presented to the server for accessing its contents via APIs (mostly `RESTful` HTTP).

The locations of the stored cookies are determined by the library that processes HTTP requests. The most commonly used library is called `NSURLConnection` [15], which is provided by the default iOS SDK. In addition, many third-party HTTP libraries [49, 50] are built on top of `NSURLConnection`. As a result, these libraries store the cookies in the same fixed locations as `NSURLConnection`. After analyzing the `NSURLConnection` library, we found that all cookies are stored inside an app's home folder by default.

Due to sandbox-based isolation, accessing files inside directories of other apps is prohibited in iOS. Since all cookies are stored separately per app, unless attackers can bypass the sandbox, a cookie can only be accessed by the corresponding app itself. Thus, storing cookies in each app directory is widely used by third-party developers and is considered to be hard to exploit [8].

4.2 Threat from the PC

This assumed difficulty of exploitation is incorrect when USB-based attacks are considered in the threat model. From a USB connection, a host computer can connect to an iOS device not only through the iTunes sync protocol, but also via the Apple File Connection (AFC) protocol [46]. AFC is designed to access media

files (e.g., images taken from the camera, recorded audio, or music files) through the USB cable. Furthermore, there exists a service maintained by `lockdownd` called `com.apple.mobile.house_arrest`, which provides access to iOS app directories through the AFC protocol. As a result, a computer has full USB-based access to the contents of `/private/var/mobile/Applications/*`.

Attacks. Credentials can be accessed via the `Cookie.binarycookies` file within an app's directory using AFC. Both the httpOnly cookie that is protected from JavaScript access and secure cookies that are only transferred through HTTPS connections can be recovered. Applying these cookies to the same apps in different devices allows attackers to log-in to the services of their victims.

Real Examples. As a proof of concept, we implemented a tool that can retrieve the cookies of Facebook and Gmail apps from a USB-connected iOS device, and transfer them to another computer.

By using these stolen cookies, we successfully logged in as the victim via the web services for both Facebook and Gmail. Although we only demonstrated the attacks against Facebook and Gmail, we believe that our finding affects a number of third-party apps that store cookies in the way similar to the Facebook and Gmail apps.

5 Measurement

In this section, we describe the methodology and datasets we use to determine a lower bound of the coexistence of iOS devices, App Store purchases made from Windows iTunes, and compromised Windows machines in home networks, with a goal to quantitatively show that a large number of users are likely to connect iOS devices to infected personal computers.

5.1 Overview

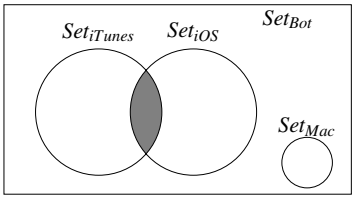


Figure 5: $(Set_{bot} - Set_{Mac}) \cap Set_{iOS} \cap Set_{iTunes}$ is the estimation of iOS devices that can be connected with bots.

Due to NAT, there are usually multiple machines (such as Windows machines, mobile phones, and Mac machines) behind a single client ID (i.e., an anonymized IP address). In our measurement, we used different fingerprints to determine the client IDs (CIDs) that produce network traffic generated by iOS devices, App Store purchases from Windows iTunes, and compromised Windows machines.

Identified as \ Truth	CID has Mac	CID has no Mac
CID has a Mac	True Positive	False Positive
CID has no Mac	False Negative	True Negative

Table 2: Definition of false positive and false negative.

First, we quantified the population of compromised Windows machines (i.e., bots) using a labeled C&C domains dataset (see Section 5.3). Since we only used C&C domains from Windows malware families, we are confident that all CIDs (Set_{bot} in Figure 5) have Windows machines. Next, within Set_{bot}, we further measured how many CIDs contain network traffic generated from iOS devices. Since there is an overlap of DNS queries from Mac OS X and queries from iOS, we excluded CIDs with Mac OS X traffic (Set_{Mac} in Figure 5). We used a set of iOS fingerprint domain names to identify a lower bound of CIDs containing iOS devices (Set_{iOS} in Figure 5). Then, we estimated how many iOS devices in Set_{iOS} are likely to be connected to a Windows machine. Since there is no unique DNS query generated when an iOS device is connected to a Windows machine either via USB or Wi-Fi, we assumed that if users have purchased an item from the App Store using Windows iTunes, they will eventaully connect iOS devices to a Windows machine. We estimated a lower bound of CIDs with Windows iTunes (Set_{iTunes} in Figure 5). Finally, the intersection of Set_{iOS} and Set_{iTunes} is our measurement result (shaded area in Figure 5).

We use false positives and false negatives to evaluate our methodology. They are defined in Table 2, using identification of Set_{Mac} as an example. If a CID has a Mac but we identify it as without a Mac, it is a false negative. On the other hand, if a CID does not have a Mac but we identify it as with a Mac, it is a false positive. The definition of false positive and false negative are similar to identifying Set_{iOS} and Set_{iTunes}. Since we want to be conservative about the size of $Set_{iTunes} \cap Set_{iOS}$, we accept false positives for Set_{Mac}, false negatives for Set_{iOS} and false negatives for Set_{iTunes}. However, we want to have small or zero false negatives for Set_{Mac} as well as false positives for Set_{iOS} and Set_{iTunes}.

5.2 Datasets

All datasets are provided by Damballa, Inc [3].

DNS Query Dataset. We obtained DNS traffic from two large ISPs in the US, collected in 13 cities for five days[4] in October 2013. A-type DNS queries from all clients to the Recursive DNS Servers of the ISPs were

[4] 10/12/2013, 10/24/2013, 10/27/2013, 10/28/2013, 10/30/2013.

captured by a sensor in each city, and then de-duplicated every hour to be stored. Each record in the dataset is a tuple of sensor ID, hourly timestamp, anonymized CID, query count, queried domain, and resolved IP address of the domain. The CID is a hash of original client IP address. The hash is performed to preserve client IP anonymity, and at the same time to retain a one-to-one mapping property. The query count denotes the number of times that client queried the particular domain and resolved IP pair in the hour. On average, we observed 54 million client IDs, 62 million queries, and 12 billion records daily from 13 sensors in total.

ISP Clients. Most CIDs in the two ISPs are home networks. Only few are small businesses. Since 99.5% of CIDs queried fewer than 1,000 distinct valid domains daily, we identified those as representative of home network users[5].

DHCP Churn. Because of long DHCP lease times (one week) and specific DHCP renewal policies (e.g., a modem's IP address will only be changed if some other modem has acquired the IP address after lease expiration) from the two large ISPs, the DHCP churn rate is very low in the networks we measured. Consequently, we consider a single CID as the same home in a given day. This is different than that of a botnet's perspective, where the bots could reside in any network [48].

Passive DNS Database. We collected and built a passive DNS database for the same days, from the same locations as the DNS query dataset to provide visibility of other types of DNS records. The database contains tuples of date, sensor ID, queried domain, query type, resolved data, and query count. Since there are only A-type records in the DNS query dataset, if there is a CNAME chain [38] for the domain name we want to measure, we use the passive DNS database to reconstruct the CNAME chain to determine the mapping of the domain we are interested in to the eventual returned A record. For example, the CNAME chain for Apple's iMessage server `static.ess.apple.com` is shown in Table 3. The final A-type domain for `static.ess.apple.com` is `e2013.g.akamaiedge.net`. If we want to know how many CIDs queried `static.ess.apple.com` in a day, we first examine the passive DNS database for that day to reconstruct all possible CNAME chains to `e2013.g.akamaiedge.net`. Next, we make sure that only `static.ess.apple.com` resolved into `static.ess.apple.com.edgekey.net`, and only `static.ess.apple.com.edgekey.net` resolved into `e2013.g.akamaiedge.net`. Finally, we measure `e2013.g.akamaiedge.net` in DNS query dataset.

HTTP Dataset. We utilized technology provided by Damballa to collect HTTP headers related to iOS and Windows iTunes, as well as those related to domain names of our selection. If User Agent strings related to iOS or Windows iTunes appeared in an HTTP header, or if any selected domain name was in the "Host" field of an HTTP request, we collected the request and corresponding response headers. The time frame for all HTTP headers we obtained is from 10/18/2013 to 11/11/2013. Since the HTTP data is not always available for all CIDs in our DNS query dataset, we only used the HTTP dataset to obtain ground truth for evaluating our approach.

Labeled C&C Domains We acquired all command and control (C&C) domain names for botnets that Damballa is tracking. The threat researchers in Damballa labeled those C&C domains using various methods, including static and dynamic analyses of malware, several public blacklists, historical DNS information, etc. We only picked malware families for Windows for the purpose of this measurement, since we wanted to avoid Mac OS X as much as possible.

`static.ess.apple.com. 3600 IN CNAME static.ess.apple.com.edgekey.net.`
`static.ess.apple.com.edgekey.net. 21600 IN CNAME e2013.g.akamaiedge.net.`
`e2013.g.akamaiedge.net. 20 IN A 23.73.152.93`

Table 3: CNAME chain for static.ess.apple.com.

5.3 Bot Population

First, we used labeled C&C domains to find CIDs containing infected Windows machines in the DNS query dataset. If a CID queried any C&C domain in a day, we consider it as having a bot at home for that day. During the five days in October, we observed 442,399 to 473,506 infected CIDs daily, with an average of 459,326. In particular, there were 473,506 infected CIDs on 10/12/2013. We use statistics from 10/12/2013 as an example below to explain how we determine the population of iOS devices and Windows iTunes.

5.4 Excluding Client IDs with Mac OS X

We utilized unique software update traffic to fingerprint Mac OS X. Apple lists five domain names related to OS X software update services [16]:

`swscan.apple.com`, `swquery.apple.com`, `swdist.apple.com`
`swdownload.apple.com`, `swcdn.apple.com`

Mac OS X is set to automatically check for security updates daily since version 10.6 [14] and to check for general software updates daily since version 10.8 [19]. We assume that if there is any Mac OS X within a CID, there must be a query to at least one of the five domains every day. According to [11], the percentage of OS X versions later than or including 10.6 is 95%. This assumption gives us a low false negative rate. To

[5] We explain why we use 1,000 as the threshold in Section A.2

evaluate the false positives of this approach, we verified using the HTTP dataset (See Table 5). We were able to collect HTTP headers for three of the five domain names. 3,530 headers for `swdist.apple.com`, 9,643 headers for `swscan.apple.com`, and 18,649 headers for `swcdn.apple.com` were observed. Among 115 unique (Source IP, User Agent string) pairs for `swdist.apple.com`, 114 User Agent strings were from Mac OS X, and one was from Mozilla, which gives us a $\frac{1}{115}$ false positive rate. Similarly, nine out of 3,884 pairs of (Source IP, User Agent string) for `swscan.apple.com` were false positives. We identified

Weather App	`apple-mobile.query.yahooapis.com`
Stocks App	`iphone-wu.apple.com`
Location Service	`gs-loc.apple.com` `iphone-wu.apple.com`

Table 4: Domains for finding iOS devices.

6,966 (1.50%) of 473,506 infected CIDs with Mac OS X on 10/12/2013. It is lower than the market share of Mac OS X since we only looked for CIDs infected with Windows malware. After excluding Mac OS X, we have 466,540 bot CIDs without Mac OS X, i.e., Set_{bot}-Set_{Mac}.

5.5 iOS Device Population

We used unique domains from two default apps and one service in iOS (the Weather app, Stocks app, and Location Services) to get a lower bound of CIDs containing iOS devices. We obtained these domains in Table 4 by capturing and analyzing network traffic when Weather, Stock, and Location Services were used in a controlled network environment. We also used the HTTP dataset for evaluation. As Table 5 shows, within all (Source IP, User Agent string) pairs that requested the three domains, all User Agent strings were from either iOS or Mac OS X. There were no User Agent strings from other operating systems. Since we have already excluded CIDs with Mac OS X traffic in the previous step, domains in Table 4 can be used to fingerprint iOS without introducing any false positives. However, if a user did not use Weather, Stocks, or Location Services in the day, it is a false negative.

Of 466,540 CIDs without Mac OS X traffic on 10/12/2013, 142,907 (Set_{iOS}) queried any of the three domains, indicating 30.63% of observed Windows bots have iOS devices in the same home network.

5.6 Windows iTunes Population

After we identified infected CIDs containing iOS devices, we further analyzed how many of these have iTunes installed in infected Windows. The biggest challenge here is that there is no unique domain name that can effectively fingerprint Windows iTunes, because Windows iTunes traffic is similar to the traffic generated by the App Store for iOS.

Fortunately, we found that because of the Apple Push Notification Service [2], iOS devices need to constantly query a certain domain name for push server configurations. Based on this feature, we define **iOS heartbeat DNS queries** as DNS queries that an iOS device always makes as long as it is connected to the Internet.

To pinpoint Windows iTunes, our observation is that if we observe App Store purchases but do not find iOS heartbeat DNS queries, then the purchases must originate from iTunes. Next, we describe how we identify the iOS heartbeat DNS queries and App Store purchase queries.

5.6.1 App Store Purchase

To fingerprint App Store purchases, we experimented with several App Store purchases in both Windows iTunes and iOS. By analyzing PCAP traces of these purchases, we discovered that domain names of the pattern `p*-buy.itunes.apple.com` is related to a purchase, where * denotes numbers. We used the HTTP dataset to check this pattern (See Table 5). 487 HTTP headers were collected from 10/26/2013 to 10/29/2013. From the 119 (Source IP, User Agent string) pairs, we confirmed that this pattern comes from the purchase of apps in either iOS or iTunes.

5.6.2 iOS Heartbeat DNS Queries

To discover iOS heartbeat DNS queries, we first collected all domains in the "Host" field of HTTP requests containing iOS-related and Windows iTunes-related User Agent strings from the HTTP dataset. Next, we examined domains that received a large number of requests, and concluded that the domain name `init-p01st.push.apple.com` is constantly queried for Apple push server configurations and certificates from iOS, but queried much less often by Windows iTunes.

Apple does not disclose how often iOS devices query their push server. To confirm our observed queries were iOS heartbeats, we utilized the following three methods.

1. HTTP Traffic Analyses: 8,990 HTTP headers were gathered for `init-p01st.push.apple.com`, from 10/18/2013 to 10/31/2013. By inspecting the distribution of "max-age" values of the Cache-Control field in the HTTP response headers, we were able to know the intended cache policy for the push server certificate in the response. For iOS, the observed max-age values were from 338s to 3,600s; for APSDaemon.exe (part of Windows iTunes), values ranged from 131,837s to 1,295,368s. Compared to Windows, iOS caches the push server certificate for a shorter time, with one hour maximum. Consequently, iOS devices must

Domain	Time Frame	HTTP headers	(Source IP, UA)	iOS	Mac	Mozilla	Other
swdist.apple.com	10/24-11/04/2013	3,530	115	0	114	1	0
swscan.apple.com	10/24-11/04/2013	9,643	3,884	0	3,875	9	0
swcdn.apple.com	10/19-11/11/2013	18,649	613	0	140	473	0
iphone-wu.apple.com	10/18-11/04/2013	17,606	1,772	1,174	598	0	0
apple-mobile.query.yahooapis.com	10/22-11/02/2013	16,808	3,018	2,999	19	0	0
gs-loc.apple.com	10/22-11/04/2013	2,380	561	367	182	0	0
p*-buy.itunes.apple.com	10/26-10/29/2013	487	119	-	-	-	App Store
init-p01st.push.apple.com	10/18-10/31/2013	8,990	-	-	-	-	-

Table 5: Ground truth of fingerprint domain names.

query `init-p01st.push.apple.com` when the cache expires.

2. Reverse Engineering: We also reverse engineered the push service daemon process `apsd` in iOS located at `/System/Library/PrivateFrameworks/ApplePushService.framework/`. We found that the URL `http://init-p01st.push.apple.com/bag` is used to set up the push-server's configuration path in the class `APSEnvironmentProduction` through the method `setConfigurationURL`. Furthermore, in another private framework called `PersistentConnection`, we found that the maximum length of time this connection can last is set to 1,800s by `setMaximumKeepAliveInterval(1800.0)`. This means iOS devices must re-send an HTTP request to get push server configurations at least every 1,800s. Moreover, the final A-type domain from `init-p01st.push.apple.com` has a TTL of 20s. As a result, every time the HTTP request is made, there must be a DNS query for the A-type domain.

3. Idle iPod Touch Experiment: We set up an iPod Touch to never auto-lock, connected it to a WiFi hotspot, and left it idle for 35.5 hours. During this period, there were 150 DNS queries to `init-p01st.push.apple.com` in total. Average query interval was 859s, with a maximum value of 1,398s and a minimum value of 293s. The maximum query interval is consistent with our 1,800s result via reverse engineering.

Our findings show that as long as an iOS device is connected to the Internet, it constantly queries `init-p01st.push.apple.com` for push service configurations, with query intervals shorter than an hour. Each query interval is determined by both the HTTP Cache-Control value and an enforced maximum interval value. The query interval for `init-p01st.push.apple.com` from iTunes is much longer. The reason for this query interval difference may be that iOS devices are more mobile than PCs. As an iOS device moves, it might need a push server closer to its current location, to ensure small push service delay.

Given the unique fingerprint for App Store purchases and a unique iOS heartbeat query pattern, we inferred a lower bound of Windows iTunes by estimating the number of CIDs that queried `p*-buy.itunes.apple.com` but did not query `init-p01st.push.apple.com` in the hour before and after the purchase query. If the Windows iTunes happened to query `init-p01st.push.apple.com` within those two hours, or if the user did not purchase anything in the iTunes App Store, it would be a false negative, since we cannot recognize Windows iTunes even though it exists. However, there are no false positives. Note that we cannot use `init-p01st.push.apple.com` to estimate the number of iOS devices for two reasons: i) Windows iTunes can query this domain; ii) the final A-type domain of `init-p01st.push.apple.com` can come from multiple original domain names[6]. Therefore, by removing CIDs that queried `init-p01st.push.apple.com` in the small time window, we exclude a larger set of CIDs that purchased from within iOS, resulting in a lower bound for Windows iTunes estimation.

On 10/12/2013, from the 142,907 infected CIDs with iOS devices, we further identified 112,233 CIDs with Windows iTunes purchases on the same day, i.e., $Set_{iTunes} \cap Set_{iOS}$. This indicates that 112,233 (23.70%) of CIDs have both iOS devices and Windows iTunes, but no Mac OS X within the home network.

5.7 Mobile Banking Traffic

The iOS devices behind bot CIDs with Windows iTunes are potential victims of our attacks. To estimate the percentage of those devices that may have banking apps installed, we chose mobile domains from eight banks (Citibank, Wells Fargo, PNC, Bank of America, Suntrust, Bank of the West, and U.S. Bank), and examined how many of those iOS devices queried them. We discovered that 4,593 (4%) of 112,233 potential victims queried mobile banking domains on 10/12/2013. This indicates that these devices are likely installed with mobile banking apps that could be replaced with fake mobile banking apps once they are infected, as discussed in Section 3.

[6]This is the only fingerprint domain name we used whose final A-type domain could be resolved from multiple different domain names.

5.8 Result Summary

We used the methodology described in this section to measure the number of iOS devices that can be potentially infected using the MitM attack described in Section 2, with five days of DNS query data. The results are summarized in Table 7. On average, we identified 459,326 bots daily. For 30% of bots, there exist iOS devices used from the same CID; and for 23% of all bots, there are both Windows iTunes installed and an iOS device used. Statistics for individual botnets as tracked by Damballa are presented in Table 6. For example, if the botmaster of botnet α bundled our attacks into their payload, there would be 75,714 potential victims in 13 cities, within the networks we monitor.

Botnet	Size	$Set_{bots} \cap Set_{iOS} \cap Set_{iTunes}$	Percentage
α	287,055	75,714	26.38%
β	69,895	12,517	17.91%
γ	49,138	10,216	20.79%
δ	16,236	3,232	19.91%
ε	13,732	2,662	19.39%
ε	5,024	1,182	23.53%
ζ	4,554	944	20.73%
η	4,377	929	21.22%
θ	4,231	834	19.71%
ϑ	4,067	806	19.82%

Table 6: Statistical analysis of the top 10 botnets with highest number of infected CIDs on 10/12/2013.

Date	Set_{bots}	$Set_{bots} \cap Set_{iOS}$	$Set_{bots} \cap Set_{iOS} \cap Set_{iTunes}$
10/12	473,506	142,907 (30.63%)	112,233 (23.70%)
10/24	452,003	134,838 (29.83%)	104,225 (23.06%)
10/27	442,399	134,271 (30.35%)	104,075 (23.53%)
10/28	461,144	138,793 (30.10%)	105,056 (22.78%)
10/30	467,579	141,242 (30.21%)	102,795 (21.98%)

Table 7: Measurement results summary, October 2013.

6 Related Work

USB Attack Vector. The USB interface has been demonstrated to be an attack vector for mobile devices for some years [18, 31, 35, 52]. Z. Wang and A. Stavrou [52] studied attacks that take advantage of USB interface connectivity and presented three attack examples on Android platforms that spread infections from phone to phone, from phone to computer, and from computer to phone. The work in [31, 35] further demonstrated that these USB based attacks can take place through USB charging stations or chargers. It is worth noting that infecting connected mobile devices from the PC has happened in the real world. Symantec has found malware samples on Windows that can inject malicious apps to USB-connected Android devices [6].

Our work is the first to show measurement results that indicate a large number of users are likely to connect iOS devices to compromised computers, potentially leading to a large scale infection of iOS devices. We hope that our measurement results could drive Apple and other mobile manufacturers to redesign the security model of USB connections, and remind app developers to securely store credentials. In addition, while most previous works focus on Android, we present various attacks against non-jailbroken iOS devices that can be launched via USB or Wi-Fi connections.

Attacks Against iOS. In recent years, more attacks against the iOS platform have been observed As one of the most representative attacks, jailbreaking is the process of obtaining root privilege and removing certain limitations (such as code signing) on iOS devices by exploiting vulnerabilities in the kernel, the boot loader, and even the hardware [37]. Since most jailbreaking tools [22, 30] deliver the exploits through a USB connection, attackers could also take advantage of these jailbreaking tools to root USB-connected iOS devices. In this case, attackers can easily inject malicious apps with the ability to read and send SMS (e.g., [5]), which will allow for more advanced attacks against SMS-based two factor authentication schemes [21, 39].

Many researchers have shown that the App Store cannot prevent publishing of malicious apps [26, 36, 51]. They also proposed defenses [20, 53] for jailbroken devices. As previously mentioned, these malicious apps could only affect a limited number of iOS users who downloaded them. Our research describes the means to deliver malicious apps to a significant number of iOS devices and could significantly amplify the threat of iOS-based malware.

Many works focus on reverse engineering iOS and its protocols. Researchers analyzed the iMessage protocol and proposed Man-in-the-Middle attacks [44]. Requiem [47] reverse engineered the Apple Fairplay DRM protection algorithm for music, movies, and eBooks, and can bypass Fairplay to decrypt protected media files. However, Requiem does not support Apple Fairplay DRM protection for apps in the App Store. The libimobiledevice project [33] enables a computer to communicate with USB-connected iOS devices without requiring iTunes, such as syncing music and video to the device and managing SpringBoard icons and installed apps. However, libimobiledevice does not contain the iTunes authorization process (Section 2.1). In other words, libimobiledevice can install an app purchased by Apple ID_A to an iOS device bound to Apple ID_B, but the app cannot successfully run due to the failure of DRM validations. In comparison, we analyzed the iTunes authorization process, and found a way to bypass the DRM validations. This can allow attackers to deliver malicious apps downloaded by their Apple IDs to different iOS devices.

Mobile OS Fingerprint. To fingerprint mobile OSes in a multi-device network environment, the User Agent field of the HTTP header, DHCP request header fields, Organization Unique Identifier (OUI, i.e., the first 3-bytes of a MAC address), or a combination of these were commonly used [12, 24, 28, 34, 41]. Unfortunately, it is not scalable to collect these data in ISP-level networks. Furthermore, it is common for each client IP in a cellular network to represent only one device. In ISP networks, many client IPs represent a NAT endpoint. To cope with the complexity caused by multiple devices per client IP in ISP networks, we used unique domains from two default apps and one service within iOS to measure the number of iOS devices. We also found a domain name related to the push service with a unique query frequency that allowed us to determine the presence or absence of an iOS device for a given client IP.

7 Discussion

7.1 Accuracy of the Measurement

We emphasize that the goal of our measurement study is to show that there is a large number of users who are likely to connect their iOS devices to compromised computers, which we argue may lead to a large scale infection of iOS devices through botnets. There are many reasons that may lead to underestimations in our measurement, which implies that even more iOS devices could be infected by the botnets that we monitor. For example, in our measurement, we did not consider cellular traffic and the case that people often have multiple iOS devices in their household.

Next, we discuss the potential reasons that may result in overestimations and analyze why they are unlikely to happen in our dataset.

Multiple Windows machines. The data we have only allows us to determine what *type* of devices are behind an IP address, but not how many of each. Thus, it is possible that there are multiple Windows machines behind an IP. In the case that there are multiple Windows machines and not all of them are infected, we may have an overestimated infection number since iOS devices could be connected to only the uninfected computers. To reduce this risk of overestimation, we excluded IPs that queried more than 1,000 unique domains in a day because these IPs are likely to be a gateway with many users. On the other hand, we can expect that within a small environment, if one Windows machine in a NAT environment is infected, it is likely that all Windows machines will eventually be infected. This is because 1) most likely all the Windows machines in the network are managed in a similar manner, and have the same level of updates and defenses, thus share the same vulnerabilities, and 2) it is likely that there is some kind of communication or data sharing (e.g., using the same Wi-Fi network or a USB thumb drive) between the machines.

Mobility of iOS devices. Due to the mobility of iOS devices, it is possible that the same iOS device appears in different "infected" IP addresses, which leads to an overestimation of the number of potential iOS victims. However, we argue that this is extremely unlikely in our dataset because the overestimation can only happen when the same iOS device is present behind different NAT-networks in the same ISP, and the NATs have infected computers that make purchases from Windows iTunes in the same day.

7.2 Mitigation and Prevention

Since Apple has remote removal and revocation abilities, they have complete mediation over what app can run on an iOS device. However, due to a significant number of apps on the App Store and the lack of runtime monitors on iOS devices, malicious apps are only detected when the users perceive adverse effects of the malicious apps. As a result, many iOS users may have already been affected by such attacks before the manually-triggered revocations and removal are applied. We have observed many Android botnets [42, 43] even though Google also has remote app removal ability [4]. In addition, this ability cannot prevent an attack that steals a user's cookies (as discussed in Section 4).

Since we only tested a few devices in the attack presented in Section 2, we cannot confirm whether Apple is able to impose a limit on the number of iOS devices that can be authorized per Apple ID. However, since registering an Apple ID only requires a valid email address, attackers can easily prepare a number of Apple IDs and use them to distribute malicious apps. Nonetheless, we still suggest that Apple should monitor the anomalous Apple IDs that deliver purchased apps to excessive number of devices. In addition, we advocate that iOS should warn the user when an app purchased by a different Apple ID is to be installed.

The attack in Section 3 relies on iOS developer licenses. An individual iOS developer license can only register up to 100 iOS devices, which prevents large scale exploitation. However, Apple has the enterprise developer license program, which allows provisioning of an arbitrary number of iOS devices. Although the application for an Enterprise iOS Developer License is arguably very difficult because one would require a Dun & Bradstreet (D-U-N-S) number, we have observed a lot of real world cases of abusing enterprise licenses, such as distributing pirated iOS apps [10], running GBA emulators [9], and delivering jailbreak exploits [7]. It is also

possible for attackers to obtain such licenses through infected machines under their control, rather than applying for one. As result, we suggest that iOS should warn the user when a provisioning profile is installed or prompts the user the first time an app is run that is signed by an unknown provisioning profile.

To prevent the malicious PC from stealing cookies through the USB connection, third-party developers should be aware that plaintext credentials could be easily leaked through the USB interface and store the credentials in a secure manner.

8 Conclusion

This paper discussed the feasibility of large scale infections of non-jailbroken iOS devices. We demonstrated three kinds of attacks against iOS devices that can be launched by a compromised computer: delivering Apple-signed malicious apps, delivering third-party developer signed malicious apps, and stealing private data from iOS devices. Furthermore, by analyzing DNS queries generated from more than half a million IP addresses in known botnets, we measured that on average, 23% of bots are likely to have USB connections to iOS devices, potentially leading to a large scale infection.

9 ACKNOWLEDGMENTS

We thank our anonymous reviewers and Manos Antonakakis for their invaluable feedback. We also thank the various members of our operations staff who provided proofreading of this paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. 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, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] Apple property list. https://developer.apple.com/library/mac/documentation/cocoa/conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html.

[2] Apple push notification service. http://en.wikipedia.org/wiki/Apple_Push_Notification_Service.

[3] Damballa, inc. https://www.damballa.com/.

[4] Google throws kill switch on android phones. http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html.

[5] irealsms. http://irealsms.com/.

[6] New malware tries to infect android devices via usb cable. http://www.symantec.com/security_response/writeup.jsp?docid=2014-012109-2723-99.

[7] Pangu jailbreak for ios7.1.x. http://pangu.io/.

[8] Starbucks app leaves passwords vulnerable. http://money.cnn.com/2014/01/15/technology/security/starbucks-app-passwords/.

[9] Abusing enterprise distribution program to let users install gba emulator, 2013. http://www.iphonehacks.com/2013/07/apple-revokes-gba4ios-signing-certificate.html.

[10] Offering pirated ios apps without the need to jailbreak, 2013. http://www.extremetech.com/mobile/153849-chinese-app-store-offers -pirated-ios-apps-without-the-need-to-jailbreak.

[11] Operating system market share, 2014. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpsp=2014&qpnp=1&qptimeframe=Y.

[12] M. Afanasyev, T. Chen, G. M. Voelker, and A. C. Snoeren. Analysis of a mixed-use urban wifi network: when metropolitan becomes neapolitan. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, IMC '08, pages 85–98, New York, NY, USA, 2008. ACM.

[13] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *IEEE/ACS Computer Systems and Applications*. 2009.

[14] Apple Inc. Mac OS X Snow Leopard and malware detection. http://support.apple.com/kb/HT4651, June 2011.

[15] Apple Inc. NSURLConnection Class Reference, Nov 2013. https://developer.apple.com/library/ios/documentation/cocoa/reference/foundation/Classes/NSURLConnection_Class/Reference/Reference.html.

[16] Apple Inc. Requirements for Software Update Service. http://support.apple.com/kb/ht3923, October 2013.

[17] A. Barth. HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265, April 2011.

[18] A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. Butler. Leveraging usb to establish host identity using commodity devices. In *Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[19] K. Bell. Apple Increases Mountain Lion Security With Daily Update Checks, Automatic Installs, More, June 2012. http://www.cultofmac.com/175709/apple-increases-mountain-lion-security-with-daily-update-checks-automatic-installs-more/.

[20] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[21] A. Dmitrienko, C. Liebchen, C. Rossow, and A.-R. Sadeghi. On the (in)security of mobile two-factor authentication. Technical Report TUD-CS-2014-0029, CASED/TU Darmstadt, Mar. 2014.

[22] Evad3rs. Evasi0n jailbreaking tool. 2013. http://evasi0n.com/.

[23] 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.

[24] A. Gember, A. Anand, and A. Akella. A comparative study of handheld and non-handheld traffic in campus wi-fi networks. In *Proceedings of the 12th international conference on Passive and active measurement*, PAM'11, pages 173–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] E. Hammer-Lahav. The OAuth 1.0 Protocol. `http://tools.ietf.org/html/rfc5849`, April 2010.

[26] 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.

[27] D. Hardt. The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`, October 2012.

[28] L. Invernizzi, S. Miskovic, R. Torres, S. Saha, S.-J. Lee, C. Kruegel, and G. Vigna. Nazca: Detecting Malware Distribution in Large-Scale Networks. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, Feb 2014.

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

[30] Y. Jang, T. Wang, B. Lee, and B. Lau. Exploiting unpatched ios vulnerabilities for fun and profit. In *Black Hat USA*, 2014.

[31] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal. Mactans: Injecting malware into ios devices via malicious chargers. In *Black Hat USA*, 2013.

[32] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Proceedings of The 20th Annual Network and Distributed System Security Symposium*, 2013.

[33] Libimobiledevice. 2013. `http://www.libimobiledevice.org/`.

[34] G. Maier, F. Schneider, and A. Feldmann. A first look at mobile hand-held device traffic. In *Proceedings of the 11th international conference on Passive and active measurement*, PAM'10, pages 161–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] B. Markus, J. Mlodzianowski, and R. Rowley. Juice jacking, 2011. `http://www.techhive.com/article/238499/charging_stations_may_be_juice_jacking_data_from_your_cellphone.html`.

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

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

[38] P. Mockapetris. Domain Names - Concepts and Facilities. `http://www.ietf.org/rfc/rfc1034.txt`, November 1987.

[39] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert. Sms-based one-time passwords: Attacks and defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2013.

[40] C. Mulliner and J.-P. Seifert. Rise of the iBots: 0wning a telco network. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software (Malware)*, Nancy, France, October 2010.

[41] I. Papapanagiotou, E. M. Nahum, and V. Pappas. Smartphones vs. laptops: comparing web browsing behavior and the implications for caching. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 423–424, New York, NY, USA, 2012. ACM.

[42] V. Pidathala, H. Dharmdasani, J. Zhai, and Z. Bu. Misosms: one of the largest advanced mobile botnets. `http://www.fireeye.com/blog/technical/botnet-activities-research/2013/12/misosms.html`.

[43] H. Pieterse and M. Olivier. Android botnets on the rise: Trends and characteristics. In *Information Security for South Africa (ISSA)*, 2012.

[44] pod2g and gg. imessage privacy. In *Hack In The Box Amsterdam*, 2013.

[45] P. Porras, H. Sadi, and V. Yegneswaran. An analysis of the ikee.b iphone botnet. In *The Second International ICST Conference on Security and Privacy in Mobile Information and Communication Systems*, 2010.

[46] M. RENARD, 2013. Hacking apple accessories to pown iDevices.

[47] Requiem. 2013. `http://en.wikipedia.org/wiki/FairPlay#Requiem`.

[48] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[49] The Restkit Project. RestKit, Nov 2013. `http://restkit.org/`.

[50] M. Thompson. Afnetworking, Nov 2013. `https://github.com/AFNetworking/AFNetworking`.

[51] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *The 22nd USENIX Security Symposium (SECURITY)*, 2013.

[52] Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[53] 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.

[54] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning. Andbot: Towards advanced mobile botnets. In *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, LEET'11, 2011.

[55] M. Zheng, H. Xue, and T. Wei. Background monitoring on non-jailbroken ios 7 devices. `http://www.fireeye.com/blog/author/twei`.

A Appendices

A.1 Fairplay

The code segments of iOS apps on the App Store are encrypted with AES-128. Specifically, rather than using a single pair of key and IV (Initialization Vector) per app, each 4K bytes (i.e., memory page size) of the code segment of an app are encrypted with a unique pair of key and IV. These keys and IVs are also encrypted and stored in an `supp` file that is inside the `SC_Info` folder within the app archive. Upon loading an encrypted app into memory, iOS will derive the decryption keys and IVs from the `supp` file, the `sinf` file that is also inside the `SC_Info` folder, and the `sidb` file that resides

under `/private/var/mobile/Library/FairPlay/iTunes_Control/iTunes/`. A heavily obfuscated kernel module `FairPlayIOKit` and a heavily obfuscated user space daemon `fairplayd` are involved in this process.

Furthermore, this user space daemon `fairplayd` is also involved in the generation of `afsync.rq` and `afsync.rq.sig` mentioned in Section 2.2. After receiving the syncing request from a PC, the air traffic control service `atc` running in iOS devices will communicate with `fairplayd` through Mach messages to generate `afsync.rq` and `afsync.rq.sig`.

A.2 Measurement

Exclude small business networks. We plotted the cumulative distribution for number of distinct valid domains queried from all CIDs in a day in Figure 6. Some CIDs queried a lot more than 2,000 distinct domains in a day, e.g., 25,138,224. However, we only show in Figure 6 until 2,000 unique domains in the x-axis. The CDF curve grows extremely slowly after 1,000 unique domains, and 99.5% of CIDs queried fewer than 1,000 unique domains in a day. Therefore, we only use CIDs that queried fewer than 1,000 distinct valid domains per day in our experiment.

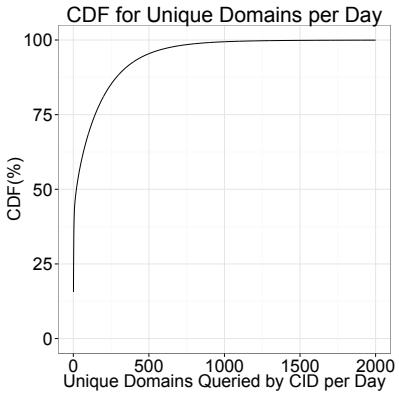


Figure 6: Cumulative distribution for number of distinct valid domains queried from all CIDs, on 10/12/2013.