

5-9-2015

## **An Analysis of Vulnerabilities Presented by Android Malware and Ios Jailbreaks**

Charles Matthew Jones

Follow this and additional works at: <https://scholarsjunction.msstate.edu/td>

---

### **Recommended Citation**

Jones, Charles Matthew, "An Analysis of Vulnerabilities Presented by Android Malware and Ios Jailbreaks" (2015). *Theses and Dissertations*. 467.

<https://scholarsjunction.msstate.edu/td/467>

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact [scholcomm@msstate.libanswers.com](mailto:scholcomm@msstate.libanswers.com).

An analysis of vulnerabilities presented by Android  
malware and iOS jailbreaks

By

Charles Matthew Jones

A Thesis  
Submitted to the Faculty of  
Mississippi State University  
in Partial Fulfillment of the Requirements  
for the Degree of Master of Science  
in Computer Science  
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2015

Copyright by

Charles Matthew Jones

2015

An analysis of vulnerabilities presented by Android  
malware and iOS jailbreaks

By

Charles Matthew Jones

Approved:

---

David A. Dampier  
(Major Professor)

---

Robert Wesley McGrew  
(Committee Member)

---

Mahalingam Ramkumar  
(Committee Member)

---

T. J. Jankun-Kelly  
(Graduate Coordinator)

---

Jason M. Keith  
Interim Dean  
Bagley College of Engineering

Name: Charles Matthew Jones

Date of Degree: May 8, 2015

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. David A. Dampier

Title of Study: An analysis of vulnerabilities presented by Android malware and iOS jailbreaks

Pages of Study: 48

Candidate for Degree of Master of Science

Mobile devices are increasingly becoming a greater crutch for all generations. All the while, these users are garnering a greater desire for privacy and style. Apple presents a device that is known for its security, but lacks major user customization. On the other hand, Google has developed a device that is keen to customization with Android, but can be susceptible to security flaws. This thesis strives to discuss the security models, app store protections, and best practices of both mobile operating systems. In addition, multiple experiments were conducted to demonstrate how an Android device could be more easily compromised after altering few settings, as well as to demonstrate the privileges, both good and bad, that could be gained by jailbreaking an iOS device.

## DEDICATION

To Addy. Thank you for your immense support and encouragement.

## ACKNOWLEDGEMENTS

I thank Dr. Ed Allen for supplying the template that was used to create this document in L<sup>A</sup>T<sub>E</sub>X.

I thank my committee for their comments on my thesis and their support.

## TABLE OF CONTENTS

DEDICATION . . . . .	ii
ACKNOWLEDGEMENTS . . . . .	iii
LIST OF FIGURES . . . . .	vi
CHAPTER	
1. INTRODUCTION . . . . .	1
1.1 Limits of Research . . . . .	1
1.2 Definitions . . . . .	2
1.3 Hypothesis and Research Questions . . . . .	3
1.4 Relevance . . . . .	4
2. LITERATURE REVIEW . . . . .	5
2.1 Security Models . . . . .	5
2.1.1 Walled-Garden Model . . . . .	5
2.1.2 Guardian Model . . . . .	6
2.1.3 User Control Model . . . . .	7
2.2 iOS . . . . .	8
2.2.1 Instance of Walled-Garden Model . . . . .	8
2.2.2 App Store Enforcement . . . . .	9
2.2.3 iOS APIs . . . . .	10
2.2.4 Secure Enclave . . . . .	11
2.2.5 Secure Boot Chain . . . . .	11
2.2.6 Runtime Process Security . . . . .	12
2.2.7 Data Execution Prevention and Address Space Layout Randomization . . . . .	12
2.2.8 Encryption . . . . .	13
2.2.8.1 Hardware Encryption . . . . .	13
2.2.8.2 File System Encryption . . . . .	14
2.2.9 Network Security . . . . .	15
2.3 Android Operating System . . . . .	16



2.3.1	Instance of User Control Model . . . . .	16
2.3.2	Architecture . . . . .	17
2.3.3	Sandboxing . . . . .	17
2.3.4	Memory Management Unity & Type Safety . . . . .	18
2.3.5	Permissions . . . . .	19
2.3.5.1	Normal-Level Permissions . . . . .	19
2.3.5.2	Dangerous-Level Permissions . . . . .	20
2.3.5.3	Signature-Level Permissions . . . . .	20
2.3.5.4	Signature-or-System Permissions . . . . .	21
2.3.6	API Framework . . . . .	21
2.3.7	Samsung Knox . . . . .	22
2.3.7.1	Application Security . . . . .	22
2.3.7.2	Platform Security . . . . .	23
2.3.7.3	Mobile Device Management . . . . .	23
2.4	App Store Protection . . . . .	24
2.5	Best Practices . . . . .	25
2.5.1	iOS Best Practices . . . . .	25
2.5.2	Android Best Practices . . . . .	26
2.5.3	Common Best Practices . . . . .	27
3.	METHODOLOGY . . . . .	29
3.1	Secure Environment . . . . .	29
3.2	Android Malware . . . . .	30
3.3	iOS Jailbreak . . . . .	32
4.	RESULTS . . . . .	34
4.1	Android Malware . . . . .	34
4.2	iOS Jailbreak . . . . .	38
5.	CONCLUSIONS & FUTURE WORK . . . . .	41
5.1	Future Work . . . . .	46
	REFERENCES . . . . .	47

## LIST OF FIGURES

2.1	Mobile operating systems separated by mobile security model [5]. . . . .	7
3.1	Network diagram of secure testing environment. . . . .	30
4.1	XML requests containing IMEI and IMSI numbers of the host device. . . .	35
4.2	Exploit exploit in the com.android.root.udevroot directory. . . . .	35
4.3	Exploit rageagainstthecage in the com.android.root.adbroot directory. . . .	36
4.4	Conversion of sqlite.db into DownloadProvidersManager.apk. . . . .	36
4.5	AndroidManifest.xml file of DownloadProvidersManager.apk. . . . .	37
4.6	Network packets being sent to DroidDream IP Address. . . . .	38
4.7	SSH access to a jailbroken iPhone 4S. . . . .	40
4.8	Root access to an iOS device running iOS 8.1. . . . .	40
5.1	Installation blocked by proper default settings. . . . .	42
5.2	Permissions requested by Bowling Time. . . . .	43

# CHAPTER 1

## INTRODUCTION

The goal of this thesis is to explain current smartphone security measures, the vulnerabilities they exhibit, measures to circumvent these weaknesses, and a discussion of whether the smartphones presented are in fact inherently vulnerable. Testing will be conducted to present known exploits and how these software and hardware vulnerabilities can be overcome using various methods.

According to Nielsen's Digital Consumer Report issued in February of 2014, almost two-thirds of average American households own at least one smartphone. Of this 65 percent of smartphone owners, 30 percent of them stated that they were planning to upgrade to a newer device within the next six months, while 49 percent of young adults age 18-21 stated the same [14]. This statistic will only continue to grow as smartphone companies continue to offer new and innovative products that capture the consumer's attention.

### **1.1 Limits of Research**

When it comes to mobile phones, two platforms alone hold the majority of the market. As of the second quarter of 2014, Nielsen reported that 52 percent of the smartphone market share was controlled by Android OS, while 42.7 percent of the devices were operating

on Apple's iOS [15]. Because of this strong hold by Apple and Google on smartphone customers, I am focusing on the iOS and Android mobile operating systems for this study.

Within all modern smartphones resides a mobile operating system, which dictates the actions of the device. A mobile operating system, or a mobile OS, can be defined as an operating system specifically designed to run on mobile devices. It is the lowest software level, on top of which other applications can run. Apple's iOS is built from the Mac OS X and XNU kernel while Android is based on Linux and heavily relies on traditional UNIX security features [1]. Because of their different characteristics and implementations, each has its own unique set of vulnerabilities.

With the majority of smartphone users using these devices to monitor their everyday lives, it becomes essential to ensure that these devices stay as secure as possible. iOS and Android devices may contain sensitive materials in the form of images, documents, and correspondences that would be harmful if leaked to persons without proper need-to-know. While this is an extreme case, technology companies are always looking to provide security to their average user who would not have these concerns, but would still have concern for their personal privacy.

## **1.2 Definitions**

Apple's iOS is a mobile operating system that is designed and developed by the Cupertino, California company and released with the first iPhone in 2006. In September of 2014, the newest version of the software was released in the form of iOS 8. iOS 8 presents many features that require impervious security such as Apple Pay, HealthKit, HomeKit, in

addition to the normal features of the iPhone that were released prior to 2014. If an individual were to gain complete control of a device, they would have access to credit cards, health records, messages, and access to the owner's home depending on which features of the device that the owner employed.

Android is another popular mobile operating system, which was developed by Google, Inc. The software made its debut in late 2008 and was last updated in late 2014 with the release of Android 5.0 Lollipop. Android is no different from iOS with its need for essential security measures. Android features a payment service through a near-field communication (NFC) chip on equipped devices, as well as, premium document editing, and messaging features that must be kept secure from unauthorized persons.

Google and Apple have different business models when it comes to their operating systems. Apple prefers a closed model, which requires applications on its devices to receive approval through a review process. Google on the other hand prefers to keep Android as open source software and allow applications from any developer. Some argue that this gives developers more freedom when designing their applications, however, this freedom may contribute to the overall security of the device.

The attacker, for this research, is defined as any malicious entity that attempts to access information or data contained within a mobile device. For mobile devices, this would be anyone other than the owner or someone authorized by the owner to access the device.

### **1.3 Hypothesis and Research Questions**

The hypothesis for this research is as follows:

Running Developer Mode on an Android device and jailbreaking an iOS device make the respective devices more accessible to malware than the devices were in their original states.

In order to test this hypothesis, experiments will be conducted with multiple devices, both iOS and Android, which contain various past operating systems to answer the following questions:

1. What vulnerabilities are made present on an iOS device when the phone is jailbroken with Pangu8?
2. What privileges are gained by jailbreaking an iOS device that are not otherwise available?
3. What are the repercussions of a malware attack when in Developer Mode on an Android Device?

Experiments will be conducted to answer these questions with details provided in chapter IV.

#### **1.4 Relevance**

As stated above, iOS and Android mobile devices are used by a majority of the American people, and thus must be made secure to protect their owners. While the average user is not of much importance to an attacker, with the exception of obtaining payment methods, attackers are vigilant in looking for ways to access the devices of high profile individuals. If these devices are not up to date, available exploits may be used to access the data that resides on them.

The remainder of this paper contains a discussion of mobile security models, security components of iOS and Android OS, and the methodology and results of the experiments conducted.

## CHAPTER 2

### LITERATURE REVIEW

#### **2.1 Security Models**

The literary review revealed many similarities and differences in the iOS and Android operating systems. For the purpose of this research, we will divide mobile operating systems into three system models: the walled-garden model, the guardian model, and the user control model.

##### **2.1.1 Walled-Garden Model**

The walled-garden model of security is the most restrictive method of the three. This model gives the smartphone vendor complete control over the third-party applications installed on their device [5]. This control can be achieved in different ways, but ultimately the apps must be approved and passed onto the user through an app store or marketplace from the vendor [5]. The vendor will also have full control over the marketplace and the removal of applications should they violate the terms and conditions set forth by the vendor [5]. If the vendor deems an application harmful, they can exercise a kill switch and remotely remove the software from all devices upon which it was installed [5]. This level of power is made possible by code signing, signing an application with the vendor's private key before being released to customers [5]. Code signing ensures customers that the

application has been reviewed by the vendor and has not been altered since it was accepted into the marketplace. The walled-garden model is different from other models in its permissions. Using the walled-garden model, the vendor makes the security decisions and is responsible for the security testing of the application [5]. This allows the customer to have a relatively worry-free customer experience when it comes to the security of their devices.

Many users prefer to stray away from the walled-garden model because they believe that it offers the vendor totalitarian control of their device. While this may be more accurate than with other models, it allows the vendor to more closely control security while monitoring trends and controlling feature use [5].

### **2.1.2 Guardian Model**

The guardian model enforces a stance with less control than the walled-garden model. This model passes the security decisions that must be made to a knowledgeable third-party [5]. This third-party could be the OS vendor (in this case, the guardian model more closely resembles the walled-garden model), the mobile carrier (AT&T, Verizon, Sprint, etc.), a security expert acting on behalf of a less technical group of users, or a policy manager who already controls the policy of the devices (an enterprise mobile device manager) [5]. The third-party would be referred to as the “guardian” and would be responsible for making the fundamental security decisions for their users [5]. This would include deciding which applications may be installed on their mobile devices and what features they would have access to, giving the end user minimal involvement with the decisions [5]. The guardian would also perform less testing and enforcement on installed applications than



in the walled-garden model, relying more on crowd sourcing, while only banning those applications that violate policy [5].

The guardian model allows for fine-tuning of controls to fit specific security levels [5]. For instance, the control by a guardian for a secure government device would be vastly more restrictive than the control exercised on the device of a low-level company manager with no access to company secrets.

### 2.1.3 User Control Model

If the guardian of a device in the guardian model were the end user, the model would shift to the user control model. Figure 2.1 shows how the user control model relates to the other two models previously mentioned. This model gives all control of security decisions to the user [5]. In the user control model, applications are released to users with minimal involvement from the vendor or mobile carrier and little oversight in their distribution [5]. Users have the option to install apps from any source they desire with the understanding that these applications may have had little to no oversight in their release and may be malicious [5].

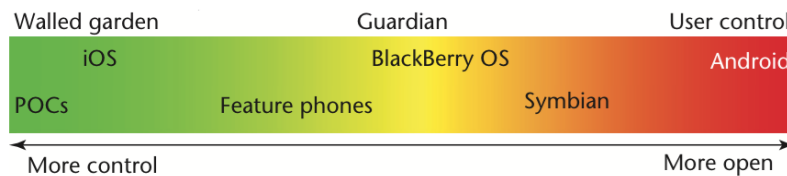


Figure 2.1

Mobile operating systems separated by mobile security model [5].

Upon installation of these applications, the user is responsible for supplying which permissions the app should have [5]. This gives the user complete control of their user experience and the security of their data. While this may be manageable for more technical users, the average user would not understand why Application A would need access to Feature B, possibly providing more access to an application than it actually needs to perform its objective.

## **2.2 iOS**

### **2.2.1 Instance of Walled-Garden Model**

iOS is a firm example of the walled-garden model. As previously stated, in a walled-garden model, the vendor has complete control of the review and release of applications into an application marketplace. No company is a better example of this than Apple. Before an application is released into the iOS App Store, the developer must introduce the app into a review process [7]. During this process, Apple's professional developers analyze and test the application [7]. The company has not released the process that it follows for application review, but we can assume that it involves a code review and verification of the submitted application for security vulnerabilities [13].

Apple enforces the application review process with code signing [13]. Only apps and libraries that have been signed with Apple's private encryption key can be loaded onto an iOS device [13]. This is a critical feature of the walled-garden model. Because Apple exercises code signing, it ensures that every app released into the app store has been properly reviewed and has been deemed safe for user use. It also prevents an application from

running a sniffer or malware tools if the attacker bypasses the initial levels of protection. If an application does contain malicious code and makes it through the review process, Apple has the ability to remove that application from the iOS App Store, as well as from all devices on which it is installed [5].

This review process ensures that Apple makes all security decisions. It allows Apple to have what some would suggest as a dictatorial approach to third-party applications, however, no one can argue that it provides the user with a more assured secure device leading to a stress-free customer experience.

While the majority of iOS firmly fits into the walled-garden model, there is one facet of Apple's iOS that strays away. Apple allows the user to make minute policy decisions on sensitive data such as: geolocation, calendar, contacts, photos, camera, microphone, and other utilities [5]. While some think that this reflects a relatively restrictive guardian model, it is hard to argue that these few permissions are more than a drop in the bucket compared to the thousands of permission decisions made from the kernel to the user interface.

### **2.2.2 App Store Enforcement**

When developers submit their applications to the App Store review process, Apple developers perform a code review to check the application code for any instance of the app reaching out of its sandbox to access services reserved for the system [13]. More information on how the iOS sandbox is protected will be discussed later. In order to use these services, developers are allowed and encouraged to use a set of application program

interfaces (APIs) set out by Apple for application usage [7]. This review ensures that applications are kept restrained in the sandbox unless they are using the specified APIs.

### **2.2.3 iOS APIs**

The APIs that Apple offers are unique to the services they provide. One of the most secure databases that the operating system must protect is the Keychain database. Keychain holds all saved passwords, wireless keys, and certificates [7]. If an application needs to access a key, password, or token, the developer may use the Keychain access APIs [3]. This API allows Keychain items to be shared between apps from the same developer [3]. The sharing process is enforced through code signing, Provisioning Profiles, and the iOS Developer Program [3].

Accessing the network securely is another system function that requires API calls. iOS supports Secure Socket Layer (SSL v3) and Transport Layer Security (TLS v1.0, TLS v1.1, and TLS v1.2) [3]. Most native iOS applications such as Mail, Safari, Calendar, and Contacts already use these services to provide an encrypted channel between the device and the network. Apple makes these services available to developers for use in securing their applications through high-level APIs such as CFNetwork [7]. CFNetwork will create and maintain a secure data stream and allow authentication information to be added to a message [3]. It calls underlying security services to set up the secure connection [3]. Low-level APIs are also available, such as SecureTransport, to provide fine-grained controls [3].

#### **2.2.4 Secure Enclave**

iOS also features a Secure Enclave. The Secure Enclave is “a coprocessor fabricated in the Apple A7 or later A-series processor” [3]. The coprocessor maintains the integrity of Data Protection and provides all cryptographic operations for key management [3]. It is secure even if the kernel has been compromised [3]. The Secure Enclave is equipped with a hardware random number generator and utilizes encrypted memory [3]. During the creation of the Secure Enclave, it is embedded with its own UID (Unique ID) that is not known by Apple or accessible to any other part of the system [3]. When the device boots, an ephemeral key is created and combined with the UID in order to encrypt the Secure Enclave’s portion of allocated memory space [3]. The Secure Enclave is responsible for performing matches against the Touch ID sensor and registered fingerprints, as well as, enabling access or purchases on behalf of the user [3].

#### **2.2.5 Secure Boot Chain**

Another integral piece of the system security architecture is the Secure Boot Chain (SBC). The SBC helps to ensure that the lowest levels of the system have not been tampered with and will only allow iOS to run on verified Apple devices [3]. When booting up an iOS device, the processor immediately calls code from the Boot ROM, a section of read-only memory [3]. This hardware boot of trust is always trusted [3]. The Boot ROM contains the Apple Root CA public key that is used to verify the Low-Level Bootloader (LLB) [3]. The LLB must be verified to be signed by Apple’s private key, before the booting process moves any further [3]. When LLB completes its tasks, it verifies and runs iBoot, the next

step of the boot process [3]. iBoot will verify and run the iOS kernel, completing the boot sequence [3]. Apple, following the chain-of-trust verification, cryptographically signs the components of SBC [3]. If any step of the SBC process is unable to verify authenticity or is unable to load, the device will display an error screen directing the user to connect the device to iTunes [3]. This screen indicated that the phone is now in Recovery Mode and must be restored to factory settings to correct any errors with the Boot ROM or LLB [3]. This necessary first step in ensuring code signing and runtime process security protects the device from execution of unauthorized code and applications during the boot sequence [1].

### **2.2.6 Runtime Process Security**

In order to protect the user after booting the device, Apple performs runtime process security. Once an app has been verified to be from a known good source through code signing, it is then “sandboxed,” restricting it from accessing files from other applications or the system itself [3]. Each application has a unique home directory for its data, which is randomly assigned in memory when the app is installed [3]. The application cannot access system files and services such as iCloud, without declared entitlements [3]. These entitlements are digitally signed and cannot be changed once issued [3]. This allows specific privileged operations that would normally have to be run as “root” while reducing the potential for a compromised application to escalate privileges [3].

### **2.2.7 Data Execution Prevention and Address Space Layout Randomization**

Apple employs strategies to prevent injected code from being executed. Data Execution Prevention (DEP) distinguishes between data and code making exploitation difficult [13].

DEP prevents an attacker from injecting malicious data into a process and jumping to that process to execute the data [13]. This security protection is typically bypassed by using return-oriented programming (ROP), however, ROP is useless when Address Space Layout Randomization (ASLR) is in use [13]. ASLR renders bypassing DEP useless because the attacker would not be able to find the code in memory [13]. Address Space Layout Randomization is used to randomly arrange the memory addresses assigned to system libraries and executable code [3]. ASLR is a default setting when programming applications within the iOS development environment, XCode [3].

If an attacker does manage to bypass both DEP and ASLR, Apple has another feature named ARM's Execute Never (XN). XN marks memory pages as non-executable [1]. Tightly controlled conditions surround the execution of memory pages that are marked as both executable and writable [3]. In order to execute these pages, the kernel checks to verify that the Apple-only dynamic code signing entitlement is present and valid [3]. Even in that case, only one mmap call is allowed to request an executable and writable memory page, which has a random address [3].

## **2.2.8 Encryption**

### **2.2.8.1 Hardware Encryption**

Encryption can be a complex problem with mobile devices. Cryptographic processes can be complex and high consumers of system resources. With mobile devices, all processes must run as efficiently as possible to preserve performance and battery life. Apple combats this problem by installing a dedicated Advanced Encryption Standard (AES) 256

crypto engine in the DMA path between the main system memory and the flash storage [3]. The devices also feature a SHA-based (SHA-1) integrity algorithm [1]. Because Apple has full control over its products, all devices running iOS contain a UID and a group ID (GID) that are fused together into the application processor using AES-256 during the manufacturing process [1, 3]. The UID is unique to each device while the GID is unique to each device group, such as all devices with an A7 processor [3]. This process allows the software and firmware to receive confirmation of encryption or decryption results without knowing the key used in the operation [3]. Also, with the UID, data is cryptographically tied to a specific device [3]. Because the UID is included in the file hierarchy protecting the file system, physically moving the memory chip from one device to another renders the data inaccessible [3]. Aside from the UID and GID, all other operations are performed using the device's random number generator (RNG) [3].

While protecting data and cryptography keys is important, erasing the data and keys securely can be essential. iOS devices include Effaceable Storage, a feature dedicated to the secure erasure of data [3]. Effaceable storage will access the underlying storage technology to directly readdress and erase a small number of blocks on a very low-level to obliterate all cryptographic keys in order to render the device cryptographically inaccessible [3]. This option is available using the "Erase all Content and Settings" feature.

### **2.2.8.2 File System Encryption**

In addition to hardware encryption, Apple's iOS also offers file data encryption to native applications, as well as third-party applications, with a technology known as "Data



Protection” [3]. Data Protection extends the hardware encryption technologies built into iOS by constructing and managing a hierarchy of keys [3]. When a file is created, it is assigned a new 256-bit key created by Data Protection [3]. This “per-file key” is passed to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES cipher block chaining (CBC) [3]. The per-file key is then used to create a hash using SHA-1, which will be used to create the initialized vector (IV), calculated with the block offset of the file [3].

The per-file key is then wrapped inside one of many class keys, depending on under what circumstances the file should be accessible [3]. This wrapped key is then stored in the file’s metadata [3].

When a user opens a file, the file’s metadata is decrypted with the file system key that is stored in Effaceable Storage [3]. This reveals the wrapped per-file key and which of the classes protects it in order to unwrap the per-file key with the class key [3]. The result of this operation is supplied to the hardware AES engine, which decrypts the file as it is being read from flash memory [3]. In order to protect the data on a device, the file system key is designed to be quickly erased [3]. Erasure of the file system key by a mobile device management server, iCloud, or user immediately renders all files cryptographically inaccessible [3].

### **2.2.9 Network Security**

iOS features network security for all clients from the home user to a corporate user. As addressed earlier, iOS offers SSL v3, as well as TLS v1.0, TLS v1.1, and TLS v1.2

[2]. However, iOS's network security does not stop there. For common users, iOS supports industry-standard Wi-Fi protocols, including WPA2 Enterprise and integration with a broad range of RADIUS servers for authentication [3]. iOS 8 also employs a randomized Media Access Control (MAC) address while scanning for networks to prevent logging of the unique device [3]. For corporate users, VPN connections are supported through the use of the following protocols: Cisco IP security (IPSec), Layer 2 Tunneling Protocol (L2TP)/IPSec and Point-to-Point Tunneling Protocol (PPTP) with the latter two requiring user authentication by a Microsoft Challenge Handshake Authentication Protocol version 2 (MS-CHAPV2) password [1].

## **2.3 Android Operating System**

### **2.3.1 Instance of User Control Model**

When dividing mobile operating systems by model, Android would squarely fit into the User Control Model. In a User Control model, the user is responsible for all software installations and security decisions [5]. Android is a prime example of this principle. Android users have the ability to download and install applications from any marketplace, including the Google Play Store. These apps are installed with minimal involvement from Google and with no involvement when downloaded from a third-party marketplace [5]. This freedom of choice for the user places Android in the category of a User Control model.

In some cases, the carrier may choose to ship a customized version of Android. If this is the case, Android shifts towards the Guardian model with the carrier becoming the

guardian [5]. There is also an argument that Android exhibits qualities of a Walled Garden model because Google can exercise a kill switch to remotely remove applications that are downloaded from the Google Play Store from devices running Android [5].

### **2.3.2 Architecture**

At the heart of the Android operating system is a Linux kernel [9]. The kernel is responsible for tasks such as: network management, memory access, process management, access to physical devices through drivers, and security [7]. The operating system also includes middleware, libraries, and APIs that are written in C and running on an application framework with Java-compatible libraries based on Apache Harmony [9].

These libraries and APIs allow for a Dalvik virtual machine (VM) to be installed above the Linux kernel [7]. Accompanying this VM are basic system libraries that allow for its execution [9]. The Dalvik VM is used primarily to provide a sandbox isolation environment for the applications that run on the device [17].

### **2.3.3 Sandboxing**

Sandboxing techniques are used in the Android operating system for the same reason they are used in iOS, to keep applications from accessing system data and services. However, Android employs sandboxes in a different fashion. Android places every app inside its own unique sandbox in order to keep it from interfering with other applications [13].

This isolation is enforced using standard Linux access control mechanisms on the Linux kernel level [7]. During the installation of the application, each application package file is assigned a unique Linux user ID (UID) [7]. This UID is associated with the

owner's UID and therefore cannot access files belonging to other applications without being granted specific permissions [7]. Read, write, and execute permissions can be issued on a file-specific basis [7]. This also prevents malicious apps from influencing system files, as they are owned by "root" [7].

#### **2.3.4 Memory Management Unity & Type Safety**

In order to force memory isolation, each application is running in its own process [7]. This means that a memory space is uniquely assigned to each app [7]. More protection is added with the Memory Management Unit (MMU). The MMU is a part of the hardware that translates between physical and virtual address spaces [7]. This prevents a malicious user from compromising the system by running code in a privileged mode, such as root, by ensuring that the user cannot modify the memory space assigned to the operating system [7].

Android also offers type safety, a programming aspect that protects buffers and memory from attacks [2]. In order to ensure this protection, Android is programmed using Java, a type safe language, and an Android binder to communicate with other languages [2].

While these methods prevent applications from communicating with another, Android provides a safe solution with shared user ID [7]. A shared UID allows two applications signed by the same developer to communicate with each other [7]. To accomplish this, the developer must digitally sign both applications with the same private key [7]. Because these two apps share the same signature, they are allowed to run in the same process to-

gether [7]. Developers should be careful to keep their private key secure when using this approach however, as there is no central authority.

### **2.3.5 Permissions**

At the heart of all Android security decisions are permissions. Permissions are used to gain access to critical system functions that are necessary for the operation of the application [7]. Each application has the ability to request and define a set of permissions to be accepted by the user during installation. This process relies on the user to make good choices and evaluate the reputation of the application they are seeking to install. Setting permissions allows applications to open up a feature of their app to other applications that have been granted permission, or to access the system [7]. Once these permissions are set they cannot be changed [7].

#### **2.3.5.1 Normal-Level Permissions**

There are four levels of Android permissions that control the access applications have to others and to the system itself [10]. The first of these permissions is “Normal”. Normal permissions are lower-risk and grant apps access to application-level features [10]. This action presents a minimal risk to other applications, the user, and the system [10]. Applications requesting normal permissions are deemed to have little impact on either the user’s or the system’s security and are therefore installed without the user’s consent of their permissions request [7]. However, the user can review the permissions prior to install if he/she so chooses [7]. An example of a normal-level permission request would be access to the phone’s vibration feature [7]. Because the vibration function does not compromise the

user's privacy or security, it is seen as an isolated feature, and is not considered a security risk [7].

### **2.3.5.2 Dangerous-Level Permissions**

Dangerous-level permissions open the system and the user to many more attack vectors than normal-level permissions. “Dangerous” permissions would supply a requesting application access to sensitive user data or control of the device [10]. For example, an application that has been granted dangerous-level privileges has access and can utilize telephony services, network services, location data, and other sensitive and personal data stored on the device [7]. Because of the security risk presented with an application with this level of privilege, the operating system is not allowed to make a decision regarding permissions during installation. The decision lies on the user to determine if the requesting application is authentic and legitimately needs access to these features in order to properly operate [10].

### **2.3.5.3 Signature-Level Permissions**

Signature-level permissions are granted to applications that are signed with the same private key as the application that declared the permission [10]. If the two applications are signed with the same certificate, the system will approve the permissions request without the input of the user [10]. Signature-level permissions are an alternative to the shared UID structure mentioned earlier [7]. Signature-level permissions allow more control over the sharing of application data and components than using a shared UID [7].

#### **2.3.5.4 Signature-or-System Permissions**

Signature-or-system level permissions are similar to signature-level permissions, however, signature-or-system permissions grant permissions to applications installed in the Android system image [7]. Permission can also be granted to applications if there is an application in the system image that is signed with the same certificate [10]. This permissions level is unique and is seen when vendors have applications built into the system image and need to share modules with an applications as they are working together [10]. Signature-or-system permissions are decided by the mobile operating system during installation and are not presented to the user [10].

#### **2.3.6 API Framework**

Android offers an API to developers through the C/C++ libraries mentioned above in order to reach lower-level system resources [7]. Android protects these APIs through additional permission label checks [8]. This means that an application must disclose its permission level in the manifest file of the app [8]. This level of permissions checking forces developers to declare their intention to interact with the system through its APIs [8].

In order to access top-level services, such as content providers, location manager, or telephony manager, Android provides a development framework. This framework makes it possible for applications to use the same system resources as the native Android applications, such as the web browser or mail client [7]. However, opening system resources to a third-party presents at attack vector for stealing data or disrupting the operation system. To combat this, Android implements the permissions structure described above [7].

### **2.3.7 Samsung Knox**

As evidenced above, Android is more prone to security threats than iOS, therefore, it is up to the individual manufacturers to ensure the security of their consumers. Samsung's solution is their product Samsung Knox. Samsung Knox is an enterprise security product originally released for the Samsung Galaxy S4 [1]. Knox bases its protection in three main areas: application security, platform security, and mobile device management (MDM) [1].

#### **2.3.7.1 Application Security**

Samsung Knox's application containers isolate outside data from the application/data pair and the inside container applications [1]. This prevents "data leakage" in the "Bring Your Own Device" environment [1]. Knox also offers on-device data encryption [1]. Due to Samsung Knox being an enterprise product, Knox has the capability to encrypt all transactions within the device using a Federal Information Processing Standards (FIPS) 140-2 certified AES-256, which is certified for both financial and healthcare applications [1]. FIPS 140-2 presents four levels of security where level 1 is the lowest, and level 4 is the highest level of security, which can protect against compromises due to environmental and operational conditions [1]. This encryption protocol uses a user-created passphrase [1]. This passphrase is fed into Password-Based Key Derivation Function 2 (PBKDF2), which performs a Hash Message Authentication Code (HMAC) on the passphrase along with a salt [1]. The process is repeated many times until a cryptographic key is the resultant [1]. Encryption also covers external storage of the device, such as an SD card [1].



### **2.3.7.2 Platform Security**

Samsung Knox's platform security takes a three-pronged approach to security using the customizable secure boot (using X.509 certificates and public key systems), TrustZone Integrity Management Architecture (TIMA), and Security Enhancements (SE) for Android [1]. TIMA utilizes ARM TrustZone hardware [1]. ARM TrustZone hardware provides constant monitoring of the Linux kernel [1]. This monitoring is achieved by physical partitioning the CPU and memory resources into normal and secure modes [1]. If a kernel violation is detected it used the MDM to notify the enterprise IT to take action [1]. The National Security Agency (NSA) created the SE for Linux mechanism in 2000 [1]. The mechanism has been adopted by Android and is used to enforce separation of information based on integrity and confidentiality requirements [1].

### **2.3.7.3 Mobile Device Management**

MDM allows for enterprise IT managers to control and monitor all mobile devices in an organization that may be spread across multiple service providers [1]. Through MDM, managers can control all privacy and security settings of a device in order to assure that if the device were to be compromised, company secrets and information could not be obtained. Knox also employs ASLR since its release in Android 4.0 to help protect applications and the device itself [1]. Position Independent Executable (PIE) and DEP were enabled in Android 4.1 and use ASLR to make the location of executable code impossible to know to an attacker [1].

## 2.4 App Store Protection

Apple and Google both host marketplaces filled with applications for their respective operating systems, and must structure them to protect against the vulnerabilities that each present. According to Steve Gold, in late 2011, approximately 95.8 percent of malware for mobile devices was written with Android in mind, while only 0.62 percent was written for iOS [9]. Most applications that are written for mobile devices are developed to be useful. They are not developed with security in the forefront, allowing malware to take advantage of these oversights.

As stated above, Apple opens its development center to registered developers in order for them to submit their applications for review and admission into the app store. For this service Apple charges a nominal fee of \$99. Aside from a revenue stream for the company, this may be in place in order to prevent malware authors from establishing many accounts to submit many malicious applications in hopes that one will make it through the process. The only exception to the development center is the iOS Developer Enterprise Program, which allows companies to develop apps and distribute them to their employees for business purposes [6].

Android offers a store similar to that offered by Apple, but with many differences behind the scenes. Google charges a nominal fee for their developers just as their counterpart does; however the fee charged is only \$25 [9]. After this fee is paid, developers are free to upload their applications with no review process [9]. Android does offer code signing, a feature that Apple heavily utilizes to keep unsigned applications from executing, however, Android's implementation is mostly for bookkeeping [13]. Code signing for Android

is mostly utilized to compare old and new app signatures, as well as, for verification of signature or signature-or-system permissions [9].

Instead of preventing malware by using a top-down approach, the Google Play Store relies on crowd sourcing [13]. Customers will rate and review applications that they use from the app store. They can also see how many other customers have downloaded a particular application and report malicious ones to Google [13]. If enough users complain about a specific app, it will be presented to Google, who has the authority to remove it from the app store and remotely wipe the app from all devices.

## **2.5 Best Practices**

In order to reach the highest level of security, there are basic principles that should be followed. Some of these principles are unique to the mobile operating system, but many apply to any mobile device. The following are some of the most noted practices that should be followed.

### **2.5.1 iOS Best Practices**

One feature that is unique to iOS and should always be enabled is Find My iPhone. This feature allows the owner of the device to locate, disable, or wipe the private key if the device is ever stolen or lost, in order to protect their data [16]. The device owner can also display a message or trigger a sound notification that will be audible even if the device is on silent in hopes that if the device is lost someone may be able to hear the alert.

Another security feature available that should be enabled is data wipe. Data wipe prevents someone from attempting to brute force a device password. Apple already offers

some default protection to brute forcing. After six incorrect password attempts, the phone will lock for one minute. After this minute, if another incorrect attempt is logged the device will lock for five minutes, then 15 minutes, and finally one hour. After the tenth incorrect attempt the device will prompt the user to connect the device to iTunes (if data wipe is not enabled) or will erase the device (if the option is enabled) [16]. To make use of this feature, iOS users should set a low autolock time. This prevents an unauthorized person from having access to data if the phone is left unlocked or in the time immediately after the device is locked [16].

iOS users should also routinely verify which applications are using their location services [16]. Over time users may stop using apps that they once granted permission to receive their location. This step prevents unused and unneeded applications from gathering location data.

Two-factor authentication should be a must for any iCloud user. After the recent celebrity photo leaks, iCloud security was brought into scrutiny for its security practices. In response, Apple introduced two-factor authentication. Before a user can be logged into iCloud, they must be verified with either another iOS device or a predetermined SMS message. The only exception to this rule is the Find My iPhone software, due to the fact that the user may not have their device when trying to utilize this security feature.

### **2.5.2 Android Best Practices**

The primary security weakness for Android is third-party applications. To protect against this weakness, Android users should avoid downloading applications from third-

party marketplaces [9]. While malware can be found in the Google Play Store, the chances of downloading an application infected with malware from the Google Play Store are less than the chances of malware infection from a less reputable source [9].

Android users should also utilize anti-virus protection [16]. These applications are designed to prevent infected apps from being installed and executed. When searching for an anti-virus application always utilize the rule above and download an app from the Google Play Store that has a large download count and good reviews [16].

### **2.5.3 Common Best Practices**

Many best practices are not unique to a mobile operating system but are general enough to fit any device. The first and most important rule to follow is to always set a device password [16]. For iOS users, this password should be a complex passphrase instead of a four-digit combination [16]. Android users have the possibility of a pattern, pin, or password, but like iOS should always stick to the longer and more complex password, as a pattern can become recognizable [16].

Another important practice that should never be skipped is device updates. Software updates offer security patches to vulnerabilities that exist within the device [16]. If the device is not updated, these vulnerabilities lie open to exploitation [16].

Mobile device users should never bypass their device's built-in security mechanisms. For iOS users, this means never jailbreaking a device and for Android users, never rooting your device [16]. Jailbreaking and rooting essentially bypass the protections designed by

the manufacturer such as code signing, and open the device to malicious content such as rootkits and worms [16].

## CHAPTER 3

### METHODOLOGY

In order to test the hypothesis and answer the research questions presented above, experiments were conducted with both Android and iOS devices. The iOS device was jailbroken with a popular jailbreak exploit to observe what privileges are gained and what vulnerabilities are exposed while the Android device was exposed to mobile malware in order to test the effects of the malware in both regular and developer operating modes.

#### **3.1 Secure Environment**

Before any experiments were conducted a secure network was created for the testing of malware and to enable packet capture. The network was setup as shown in Figure 3.1 below with a wireless router for the connection of mobile devices, a network hub for distribution of packets and a cable modem to act as a DHCP server. There was no Internet connection in this configuration.

After a secure network was established, a machine was formatted to capture network packets, install software, and analyze malware. A mid-2010 15-inch MacBook Pro was used for this process. A new version of Mac OS X 10.9.5 (Mavericks) was installed with all available updates. The disk was then partitioned into five partitions. The first two were composed of the Mac OS X version mentioned above with its accompanying recovery

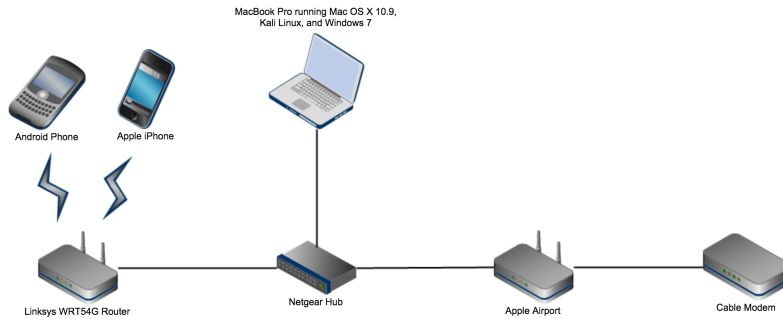


Figure 3.1

Network diagram of secure testing environment.

partition. The remaining partitions were Windows 7 Service Pack 1 64-bit with all available updates, Kali Linux 1.0.9a 64-bit with all available updates, and rEFInd, an EFI boot manager utility, to select a partition when booting.

### 3.2 Android Malware

Software installed on the Mac OS X partition, excluding native applications, include the following: Wireshark, Android Studio, Sublime Text 2, and X11. The OS X partition was used to capture packets from the network, as well as, to decompile and analyze Android malware applications. The Windows partition was used to install malicious packages onto the Android device.

The device used in these experiments was an LG Optimus L7 P715 with Android OS v4.1.2 (Jelly Bean). The device was restored to factory settings to delete any user data that might be on the device. Once erased, Astro File Manager was downloaded and installed from the Google Play Store to open packages once they were placed on the device.



The malware selected for analysis was a copy of DroidDream (md5 hash: d4fa864eedcf-47fb7119e6b5317a4ac8). The sample was obtained from Virus Share. DroidDream is a dangerous version of Android malware that obtains information about the device to send to a control and command server whom then instructs the local process to attempt to root the device. This operation will successfully open a back door in the device that allows for additional payloads to be downloaded and executed at any time without the users knowledge or authorization. According to Kevin Mahaffey, more than 50 legitimate applications from the Google Play Store were downloaded, infected with malware, and uploaded as a free version of a popular application [12]. The malicious applications were uploaded with the intention of tricking unsuspecting users into downloading the malicious application instead of the application they were intending to download. To normal users, the mistake may be one that they never realize, as the malicious application acts in the same manner as the legitimate application, with the exception of a separate process being started upon launch of the app. Users could have possibly avoided this breach by only downloading applications that have been downloaded numerous times and were recipients of overall generous reviews. This practice would have led users away from these malicious apps, which only existed for a short time before being discovered. In general, if there are two seemingly identical versions of an application, a red flag should be raised, as one of them is possibly malicious.

In order to test the network actions of DroidDream, Wireshark was used to capture the network traffic. Before installation of the malware, the Android device was connected to the closed network for a period of fifteen minutes in order to gather an array of packets

that would be transmitted during legitimate uses of the device. These packets would serve as our control for the experiment. At the conclusion of this process, the application Bowling Time, a DroidDream infected application, was uploaded to the Android device, and installed through Astro File Manager. Once installed, the application was started with all packets on the network being recorded in Wireshark for another period of approximately fifteen minutes. During this period, all features of the application were utilized in order to make sure all functions of the malware that may be controlled by game functions were executed.

The device was then again wiped to factory settings with Astro File Manager being the only third-party application installed. Another control sample was gathered with an active network for a period of approximately ten minutes. After this sample was recorded, the malware was once again installed and exercised for a period of fifteen minutes in order to gather all packets being sent and received with an open network.

### **3.3 iOS Jailbreak**

For the iOS experiment an iPhone 4S with iOS 8.1 was used. The device was wiped clean using the functions preinstalled on the device that were described in Chapter Two. Once erased, the device was set up with no network connectivity and no iCloud support. Before any experiments were conducted a packet capture was completed on the closed network to act as a control for any experiments with the iOS device.

To test the effectiveness and privileges of a jailbreak on an iPhone, I first had to successfully gain root access to an iOS device. In my research I discovered two worthy candidates

for such a task with Pangu8 and TaiG. I first experimented with Pangu8. I experienced immediate difficulties as the application presented a network error on both the Windows and Mac OS X partitions. After unsuccessful troubleshooting, I attempted to use the restore function of the Windows application to install both iOS 8.1 and iOS 8.1.2, the latest version of iOS susceptible to a jailbreak. This action only succeeded in placing the device in recovery mode. I was able to escape recovery mode with the application Tinyumbrella without having to restore the device to a signed version of iOS. After these failed attempts I downloaded, and was immediately successful, with the application TaiG. Packet captures were also captured of any network connections that may be attempted during the TaiG exploit and installation of the jailbreak software.

## CHAPTER 4

### RESULTS

After conducting the experiments detailed in Chapter Three, I examined the devices and data gathered during the study. Using these resources, I observed the following outcomes that are the result of my experiments. Interpretation of parts of the source code were aided by Lookout Mobile Security who provides a detailed technical analysis of the DroidDream malware [11].

#### 4.1 Android Malware

The DroidDream malware deployed on the Android device is a rootkit type of malware. The sample is disguised as a legitimate Android application, and hides its exploit in *com.android.root*. Once the user opens the seemingly legitimate application, DroidDream will launch its own service within *com.android.root.main* before starting the intended service of the host application.

After the services are started, the malicious application attempts to make contact with the command and control server. As seen in Figure 4.1, the application will send a request with the device's IMEI and IMSI numbers and wait for a reply for the server. When the reply is received, the malware checks to see if the device is already infected by checking for the existence of *'/system/bin/profile'*.

```

HttpURLConnection httpURLConnection;
OutputStream outputStream;
ByteArrayInputStream byteArrayInputStream;
byte abyte1[];
Formatter formatter = new Formatter();
Object aobj[] = new Object[5];
aobj[0] = "582";
aobj[1] = "18011";
aobj[2] = adbRoot.getIMEI(context);
aobj[3] = adbRoot.getIMSI(context);
aobj[4] = (new StringBuilder(String.valueOf(Build.DEVICE))).append("-").append(android.os.Build.VERSION.SDK_INT).toString();
formatter.format("<?xml version='1.0' encoding='UTF-8' ?><Request><Protocol>1.0</Protocol><Command>0</Command>
<ClientInfo><Partname>Partname</Partname><ProductID><?</ProductID><IMEI><?</IMEI><IMSI><?</IMSI><Module><?</Module></ClientInfo></Request>", aobj);
byte abyte0[] = formatter.toString().getBytes();
adbRoot.crypt(abyte0);
httpURLConnection = (HttpURLConnection)(new URL(s)).openConnection();
httpURLConnection.setDoOutput(true);
httpURLConnection.setDoInput(true);
httpURLConnection.setRequestMethod("POST");
outputStream = httpURLConnection.getOutputStream();
byteArrayInputStream = new ByteArrayInputStream(abyte0);
abyte1 = new byte[1024];

```

Figure 4.1

XML requests containing IMEI and IMSI numbers of the host device.

At this point, DroidDream attempts to root the device using two different exploits. The first attempt is ‘exploid’ that attempts to exploit a vulnerability in Androids handling of ‘init’. This reference is found in *com.android.root.udevroot* and is seen Figure 4.2.

```

file = new File(ctx.GetFilesDir(), "exploid");

```

Figure 4.2

Exploid exploit in the com.android.root.udevroot directory.

If ‘exploid’ does not successfully complete, the malware attempts to root the device using ‘rageagainstthecage’ that attempts to utilize the vulnerability of adb’s attempt to drop its privileges. This exploit is run from *com.android.root.adbroot* and is seen in Figure 4.3.

After the device has been successfully rooted by either of the previous two exploits, DroidDream attempts to deploy a second payload. The malware copies the file ‘sqlite.db’ from within the assets folder to */system/app* and renames it to a more discrete name of ‘DownloadProvidersManager.apk’ as seen in Figure 4.4.

```
private boolean runExploit()
{
    File file = new File(ctx.getFilesDir(), "rageagainstthecage");
    boolean flag = file.exists();
    boolean flag1 = false;
}
```

Figure 4.3

Exploit rageagainstthecage in the com.android.root.adbroot directory.

```
if (flag && !isPackageInstalled(ctx, "com.android.providers.downloadsmanager"))
{
    cpFile(ctx, "sqlite.db", "DownloadProvidersManager.apk");
}
```

Figure 4.4

Conversion of sqlite.db into DownloadProvidersManager.apk.

As this operation is being conducted with ‘root’ privileges, the user will not be prompted for permission or authorization. After ‘sqlite.db’ is deployed, the operations of the original payload are completed.

The second payload is unlike the first in the fact that it does not have to be activated by the user. This discrete application is triggered by the Android system actions ‘BOOT\_COMPLETED’ and ‘PHONE\_STATE’ as seen in the ‘AndroidManifest.xml’ file of the second payload. This is shown in Figure 4.5 below.

Because the application is stored in the system directory and is not in the application menu, neither the user nor any other non-system applications can monitor it. This allows the application to run silently and without detection in order to download any other payloads that it desires.

```
<intent-filter>
  <action android:name="android.intent.action.BOOT_COMPLETED" />
  <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
<intent-filter>
  <action android:name="android.intent.action.PHONE_STATE" />
  <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Figure 4.5

AndroidManifest.xml file of DownloadProvidersManager.apk.

After successfully deploying the DroidDream malware onto a clean Android device, I gathered all network packets sent and received from both a closed and open network configuration. During the closed connection test, the only packets being sent outside the network, with the exception of DNS requests, were to <http://184.105.245.17:8080/GMServer/GMServlet> as seen in Figure 4.6. Every time the application is launched the malicious payload attempts to contact this server up to five times before it eventually fails and halts.

Upon establishing an open network, I again examined packets from both a clean control and infected device. Once the device was infected with malware, data was once again being sent to <http://184.105.245.17:8080/GMServer/GMServlet> and being retransmitted up to 4 times before failure. Upon further examination of the command and control server, the IP address is registered to Hurricane Electric in Fremont, California. Hurricane Electric is an Internet backbone provider that has servers around the world. Because the server does not respond to the requests sent from the malware nor is it responding to pings, it appears that the provider has since taken down the server registered at the above IP address. As the server is not available to respond to the requests from the first payload, the malware halts

38	2015-02-10	09:32:59.656560000	192.168.100.11	184.105.245.17	TCP
40	2015-02-10	09:33:00.649212000	192.168.100.11	184.105.245.17	TCP
44	2015-02-10	09:33:02.650318000	192.168.100.11	184.105.245.17	TCP
50	2015-02-10	09:33:06.658728000	192.168.100.11	184.105.245.17	TCP
66	2015-02-10	09:33:14.680262000	192.168.100.11	184.105.245.17	TCP
145	2015-02-10	09:34:05.798363000	192.168.100.11	184.105.245.17	TCP
150	2015-02-10	09:34:06.799100000	192.168.100.11	184.105.245.17	TCP
153	2015-02-10	09:34:08.798788000	192.168.100.11	184.105.245.17	TCP
155	2015-02-10	09:34:12.808997000	192.168.100.11	184.105.245.17	TCP
166	2015-02-10	09:34:20.838939000	192.168.100.11	184.105.245.17	TCP
424	2015-02-10	09:37:24.192135000	192.168.100.11	184.105.245.17	TCP
425	2015-02-10	09:37:25.187932000	192.168.100.11	184.105.245.17	TCP
430	2015-02-10	09:37:27.189178000	192.168.100.11	184.105.245.17	TCP
435	2015-02-10	09:37:31.197525000	192.168.100.11	184.105.245.17	TCP
441	2015-02-10	09:37:39.227516000	192.168.100.11	184.105.245.17	TCP
713	2015-02-10	09:40:58.925213000	192.168.100.11	184.105.245.17	TCP
717	2015-02-10	09:40:59.925697000	192.168.100.11	184.105.245.17	TCP
720	2015-02-10	09:41:01.926652000	192.168.100.11	184.105.245.17	TCP
722	2015-02-10	09:41:05.936435000	192.168.100.11	184.105.245.17	TCP
734	2015-02-10	09:41:13.958428000	192.168.100.11	184.105.245.17	TCP
760	2015-02-10	09:41:30.014641000	192.168.100.11	184.105.245.17	TCP
959	2015-02-10	09:43:15.234275000	192.168.100.11	184.105.245.17	TCP
960	2015-02-10	09:43:16.227833000	192.168.100.11	184.105.245.17	TCP
965	2015-02-10	09:43:18.228750000	192.168.100.11	184.105.245.17	TCP
971	2015-02-10	09:43:22.238606000	192.168.100.11	184.105.245.17	TCP
978	2015-02-10	09:43:30.248120000	192.168.100.11	184.105.245.17	TCP

Figure 4.6

Network packets being sent to DroidDream IP Address.

and does not attempt to run either exploit to root the device or install ‘sqlite.db’ onto the system partition.

## 4.2 iOS Jailbreak

While testing the iOS jailbreak, the TaiG jailbreak successfully exploited the iOS device and deployed Cydia, an app store for jailbroken devices. Once Cydia was on the device, I was able to download applications straight to the device that had not achieved Apple App Store approval. These applications available in Cydia can range from dangerous malware to simple lock screen or home screen personalization.

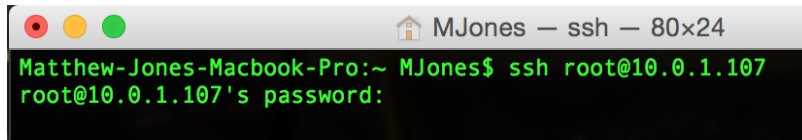
The TaiG jailbreak exploit utilizes several vulnerabilities in iOS versions prior to iOS 8.1.3. Apple released all four of the following vulnerabilities in their security content report for iOS 8.1.3 [4]. The first of these vulnerabilities was a problem with the symbolic linking mechanism of the Apple File Conduit that allowed an attacker to craft AFC commands that



may allow access to protected sections of the file system (CVE-2014-4480). This vulnerability was addressed with additional path checks. The second vulnerability exploited by TaiG was an issue in the handling of Mach-O executable files with overlapping segments that allowed the jailbreak team or an attacker to execute code that had not been signed by Apple (CVE-2014-4455). This issue was overcome with additional segment size validation. Yet another vulnerability was a buffer overflow in the IOHID family that allowed a malicious user to execute code with system privileges (CVE-2014-4487). Size validation plugged this hole to future attacks. The final issue utilized by the TaiG jailbreak team was an issue with the `mach_port_kobject` kernel interface that could leak kernel address and heap permutation values that allowed an attacker to bypass Address Space Layout Randomization (CVE-2014-4496). After this discovery, the `mach_port_kobject` interface was disabled in production configurations.

Once an iOS device is jailbroken, it is important to download OpenSSL and change the root password of the device. Because passwords can sometimes be difficult to guess, most malware for jailbroken iOS devices uses the default root password “alpine” due to the fact that many users who jailbreak their devices fail to change this password and therefore leave themselves vulnerable to malware with root access to their device. As seen in Figure 4.7, after installing OpenSSL through the Cydia app store I was able to SSH into the device.

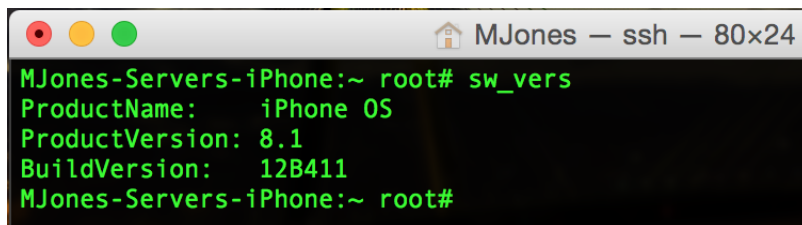
Once I was logged in, I could run any command I wished, however the command run in Figure 4.8 below, only details the operating system on the device to demonstrate it is in fact an iOS device running iOS 8.1.

A terminal window titled "MJones - ssh - 80x24" with standard macOS window controls. The text inside the terminal is green on a black background. It shows a user named "MJones" at a "Matthew-Jones-Macbook-Pro" machine initiating an SSH connection to "root@10.0.1.107". The prompt "root@10.0.1.107's password:" is visible, indicating the connection is in progress or waiting for input.

```
Matthew-Jones-Macbook-Pro:~ MJones$ ssh root@10.0.1.107
root@10.0.1.107's password:
```

Figure 4.7

SSH access to a jailbroken iPhone 4S.

A terminal window titled "MJones - ssh - 80x24" with standard macOS window controls. The text inside the terminal is green on a black background. It shows a user named "MJones" at an "MJones-Servers-iPhone" machine running the command "sw\_vers" as root. The output displays system information: "ProductName: iPhone OS", "ProductVersion: 8.1", and "BuildVersion: 12B411".

```
MJones-Servers-iPhone:~ root# sw_vers
ProductName:   iPhone OS
ProductVersion: 8.1
BuildVersion: 12B411
MJones-Servers-iPhone:~ root#
```

Figure 4.8

Root access to an iOS device running iOS 8.1.

## CHAPTER 5

### CONCLUSIONS & FUTURE WORK

After conducting my experiments and gathering the results, I was able to arrive at a set of conclusions that would prevent or mitigate the threats allowed in the previous exercises.

While the DroidDream exploit failed due to the failure of its command and control server, the outcome could have been drastically different. If the server had been operational, the device could have been exploited with one of the encompassed rootkits and utilized by a malicious user. Heeding the warning seen in Figure 5.1 below can thwart this attempt.

By only downloading applications from known and signed sources, such as the Google Play Store, users can protect themselves from most malicious apps. In the event a non-verified app is downloaded and an attempt to install it is made, the message in the above Figure 5.1 will be displayed warning the user of the danger. This message will only be displayed if the default settings concerning applications from unknown sources are selected in the security settings. While this setting may not be of any protection if a malicious application is downloaded from the marketplace, as DroidDream originally was, it will protect users from these applications once they are removed from the marketplace. However, it is my recommendation that the setting never be changed except in extreme circumstances,

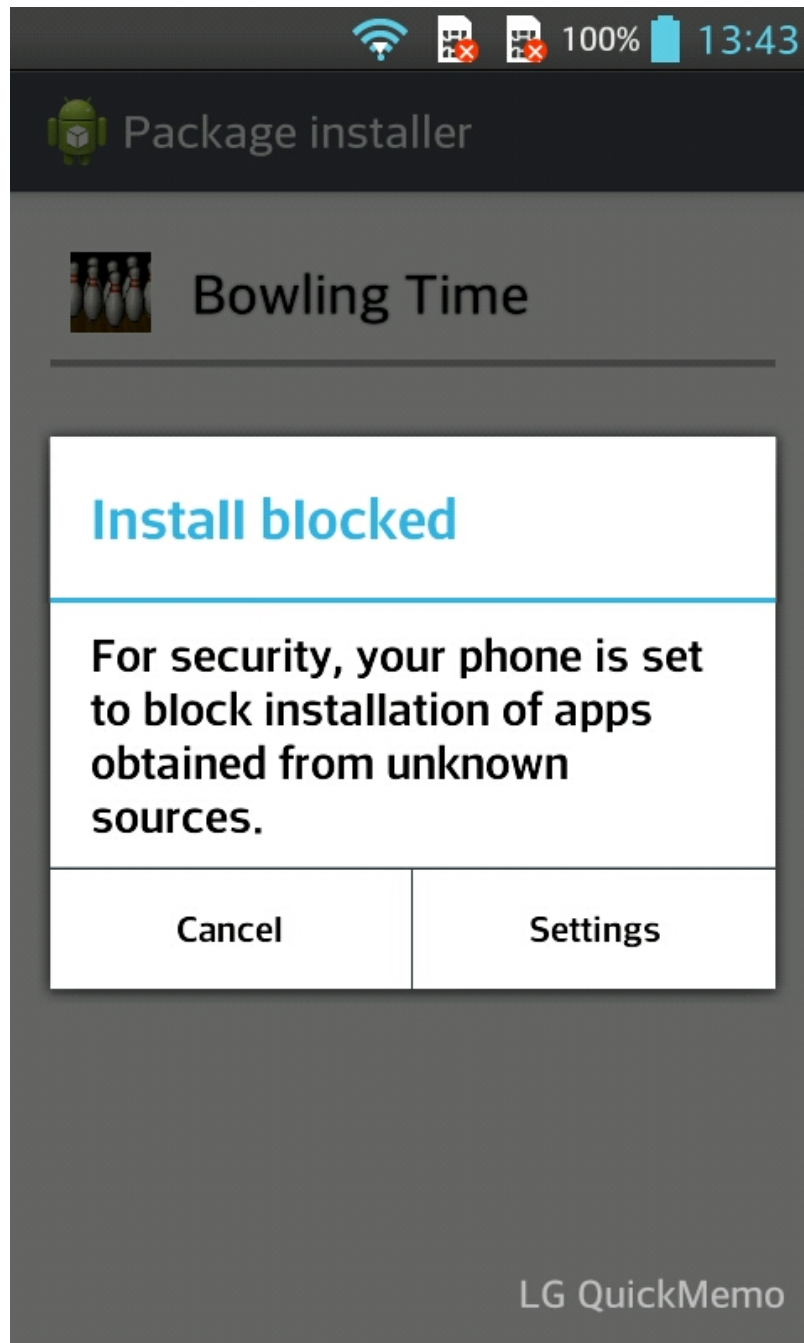


Figure 5.1

Installation blocked by proper default settings.

such as the installation of a custom application created for an individual or organization by a trusted developer. After installation, it is recommended to return the setting to its default action.

Another sign of a rogue application is a request for permissions outside the scope of the application. Figure 5.2 below shows the permissions requested for the DroidDream infected app Bowling Time.

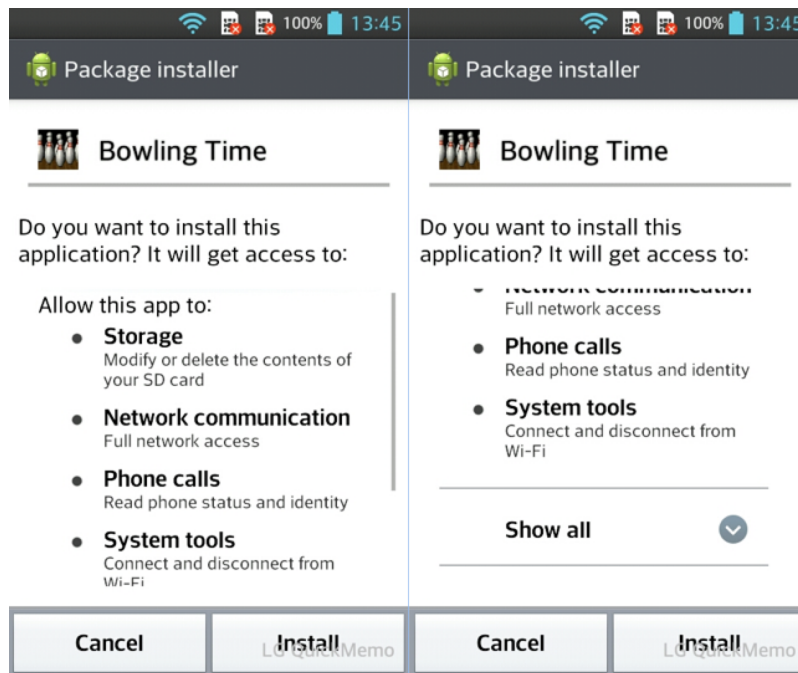


Figure 5.2

Permissions requested by Bowling Time.

As seen in Figure 5.2 above, the malicious application requests access permissions to storage, network communications, and phone calls. With the exception of network access, all of these resources are seen as unusual for a bowling game. Another resource to protect

against a malware threat is a virus scanner installed on the device. While a virus scanner may not flag every threat, it is useful for the amount of threats that it does block and remove from the device. I installed Lookout on the Android device I was testing and let it update its threat data and scan the native applications for threats before deploying malware. As soon as the malware sample was dropped onto the device, the virus scanner recognized it and removed it. These applications also often offer additional features such as location information in the event of theft and automated backups.

Once the iOS device was successfully jailbroken, a new realm of possibilities was opened for the device. While these possibilities are enticing to iOS users, the danger to the jailbroken devices user lurks readily under the surface as the device looks and functions in the same manner as an iOS device that has not been jailbroken with the exception of the Cydia application. Danger looms for many jailbreak users because the jailbreak software effectively bypassed and is continuing to bypass Apples Secure Boot Chain described in Chapter Two. Without this protection, the user is susceptible to threats that they were once protected against. The best defense against malware on a jailbroken iOS device is to only download applications from the Apple App Store and to change the root password immediately. If the root password is not changed, malware can potentially login as the root user and change the root password to any value. The only recourse to this action would be to completely restore the device and change the root password once the device has been jailbroken again.

While jailbreaking an iOS device can give a user much more freedom with their device than they previously had, it is not recommended because the user will not be protected

against new threats that are addressed in future iOS releases without restoring their devices to a “locked” state. As of the writing of this paper, a jailbreak is not currently available for iOS 8.1.3, leaving all users on older jailbroken versions susceptible to released security flaws with no means of updating their devices without losing their jailbreak features.

After examining the experiments conducted and their results, I returned to my questions and hypothesis proposed in Chapter One of this paper. When jailbreaking an iOS device, the device is made vulnerable to a slew of exploits that have been made publically available until a jailbreak update has been created for the newest iOS version, as has been the case between iOS 8.1.2 and iOS 8.1.3. However, installing an updated operating system does not thwart all exploitation attempts due to the bypassing of the Secure Boot Chain and device protections offered with a non-jailbroken device. It is widely known that the root password of an iOS device is “alpine”, and it does not look as if Apple has any plans to change it as they may see it as a jailbreaking deterrent. Therefore, since root access is possible, root privileges are possible with a jailbroken device. With this access, any modification can be made to the device once the secure boot chain is bypassed and root access is granted.

Transitioning across platforms to Android where malware is much more prevalent, we look at the experiment conducted. Judging by the actions of this malware, it is irrelevant in which mode you are running an Android device. If a piece of malware is installed, it can send out private information to a third party and potentially root the device depending on the type of rootkit and Android version installed. Thankfully, the best protection against this is very simple. As long as the default setting concerning downloading from unknown

sources is not altered, an Android user is protected from most threats, as the Google Play Store usually has the lowest amount of malware.

After answering these questions, I have come to the conclusion that my hypothesis was partially correct. While jailbreaking an iOS device does make it more susceptible to malware than a non-jailbroken device, developer mode has little to no impact on the state of an Android device. Instead, changing the installation from unknown sources default setting of the device and the absence of virus protection make an Android device more accessible to malware.

Upon these conclusions, I would revise my hypothesis to the following in order to more accurately reflect my experiments:

Changing the “Installation from Unknown Sources” default setting and the absence of virus protection on an Android device; and the jailbreaking of an iOS device make the respective devices more accessible to malware than the devices were in their original states.

## **5.1 Future Work**

Future work on this topic would include the testing of more malware samples involving different rootkits on different releases of the Android operating system. This would allow for us to see a broader range of malware on a broader range of operating system versions. It would also be helpful to find and demonstrate iOS malware samples on jailbroken and non-jailbroken devices to demonstrate their differences.



## REFERENCES

- [1] S. Adibi, “Comparative Mobile Platforms Security Solutions,” *Proceedings: 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, Toronto, Ontario, IEEE, pp. 1–6.
- [2] F. Al-Qershi, M. Al-Qurishi, S. Md Mizanur Rahman, and A. Al-Amri, “Android vs. iOS: The Security Battle,” *Proceedings: 2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, Hammamet, Tunisia, Jan. 2014, IEEE, pp. 1–8.
- [3] Apple, “iOS Security,” Apple Inc., Oct. 2014, [https://www.apple.com/br/privacy/docs/iOS\\_Security\\_Guide\\_Oct\\_2014.pdf](https://www.apple.com/br/privacy/docs/iOS_Security_Guide_Oct_2014.pdf) (current 21 Feb. 2015).
- [4] Apple, “About the Security Content of iOS 8.1.3,” Feb. 2015, <http://support.apple.com/en-us/HT204245> (current 21 Feb. 2015).
- [5] D. Barrera and P. Van Oorschot, “Secure Software Installation on Smartphones,” *IEEE Security & Privacy*, vol. 9, no. 3, May/June 2011, pp. 42–48.
- [6] Z. Benenson, F. Gassmann, and L. Reinfelder, “Android and iOS Users’ Differences Concerning Security and Privacy,” *Proceedings: CHI ’13 Extended Abstracts on Human Factors in Computing Systems*. ACM, Apr/May 2013, pp. 817–822.
- [7] G. Delac, M. Silic, and J. Krolo, “Emerging Security Threats for Mobile Platforms,” *Proceedings: 34th International Convention on MIPRO*, Opatija, Croatia, May 2011, IEEE, pp. 1468–1473.
- [8] W. Enck, M. Ongtang, and P. McDaniel, “Understanding Android Security,” *IEEE Security & Privacy*, vol. 7, no. 1, Jan/Feb 2009, pp. 50–57.
- [9] S. Gold, “Android a Secure Future At Last?,” *Engineering & Technology*, vol. 7, no. 2, Mar. 2012, pp. 50–54.
- [10] L. Jeter and S. Mishra, “Identifying and Quantifying the Android Device Users’ Security Risk Exposure,” *Proceedings: 2013 International Conference on Computing, Networking and Communications (ICNC)*, San Diego, California, Jan. 2013, IEEE, pp. 11–17.

- [11] Lookout Mobile Security, “Technical Analysis: DroidDream Malware,” Mar. 2011, <https://blog.lookout.com/droiddream/> (current 21 Feb. 2015).
- [12] K. Mahaffey, “Security Alert: DroidDream Malware Found in Official Android Market,” Mar. 2011, <https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/> (current 21 Feb. 2015).
- [13] C. Miller, “Mobile Attacks and Defense,” *IEEE Security & Privacy*, vol. 9, no. 4, July/Aug 2011, pp. 68–70.
- [14] Nielsen, “The Digital Consumer,” The Nielsen Company, Feb. 2014, <http://www.nielsen.com/content/dam/corporate/us/en/reports-downloads/2014%20Reports/the-digital-consumer-report-feb-2014.pdf> (current 21 Feb. 2015).
- [15] Nielsen, “Top U.S. Smartphone Operating System by Market Share,” The Nielsen Company, Dec. 2014, <http://www.nielsen.com/us/en/top10s.html> (current 21 Feb. 2015).
- [16] O. Tae, B. Stackpole, E. Cummins, C. Gonzalez, R. Ramachandran, and L. Shinyoung, “Best Security Practices for Android, Blackberry, and iOS,” *Proceedings: First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT)*, Seoul, South Korea, June 2012, IEEE, pp. 42–47.
- [17] R. J. G. Vargas, R. G. Huerta, E. A. Anaya, and A. F. M. Hernandez, “Security Controls for Android,” *Proceedings: 2012 Fourth International Conference on Computational Aspects of Social Networks (CASoN)*, Sao Carlos, Brazil, Nov. 2012, IEEE, pp. 212–216.