SYSTEM AND FRAMEWORK FOR IDENTIFICATION OF INFO MALWARE ON THE ANDROID OPERATING SYSTEM

BY

ROBERT MARSAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

Abstract

The tremendous rise of mobile computing has led to an equally strong rise in mobile malware. We address this issue with several novel concepts, permission fingerprints and user-app agreements. We also present a groundbreaking and substantial new dataset of the state of Android Apps, and provide unique results gathered from it. Finally, we present AndroMEDA, an Android Security Extension which helps address many of the shortcomings to existing techniques of malware identification currently available.

This *seriously* needs to be extended.

To my

Parents, Lisa and Mark

Acknowledgments

I would like to thank my advisor Roy H. Campbell for his mentorship and guidance, as well as Alejandro Gutierrez for putting up with me.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Thesis Statement: Malware on Mobile Operating Systems, especially Android, is better understood when not just analyzing the capabilities of an application, but the expectations the user has as to how it utilizes those capabilities as well.

The rise of smartphones in the last 6 years has been nothing short of meteoric. Since the launch of the Apple iPhone in 2007, over ?1 billion or so? smartphones have been sold, from over ?100 or so?(Author, Year) manufacturers. These new devices marked an unprecedented shift in our relationship with computers, becoming the center point for many personal endeavors, and superseding almost all previous computing devices from cell phones, to cameras, to GPS devices, and to most uses of a desktop pc (Author, Year). Indeed, smartphones continue to become the focal point of almost all personal computing, and consequently the operating systems they run become more important and powerful.

Mobile operating systems (mobile OSs), like the PC operating systems of the 1990s, have a few major players that wield the most influence. The two largest operating systems in the mobile area are Android and iOS. Apples iOS, made exclusively for the Apple iPhone and iPad, currently runs on over 15%(Author, Year) of all smartphones globally. Googles Android, released as an open source OS, has many different hardware manufacturers, Samsung, LG, HTC, Motorola, and many more. It currently runs the majority of smartphones globally, with a 75% marketshare. Some of the less popular, but still significant mobile operating systems are Windows Phone, with 2%, and Blackberry, with 4%.

?Introduce iOS and Android!? Android was started in 2003 by Andy Rubin and Android, Inc, previously the makers of the T-Mobile Sidekick. In 2005 they were acquired by Google Inc, and work continued in secret until its first release in 2008 with the HTC Dream.

The diversity of hardware that smartphones were designed to replace, along other constraints and features, requires a mobile OS thats designed from the ground up to deal with many different challenges than the typical PC OS. Some of the main design challenges for a mobile OS are:

- Small memory footprint, battery conscious, and other resource constrictions
- Access to a wide variety of personally identifiable information (PII)

- Access a wide array of hardware

In order to effectively enforce rules on battery consumption, low-latency UI, and personally identifiable information, a new security model was created, centered around the concept of the App.

This security model has dramatically changed the nature of mobile software, and in turn, mobile malware. By forcing malware to fit inside of this security sandbox, malware authors must choose to either break out of the box, or work inside of it. This constriction has blurred the definition of mobile malware, and ultimately has profound implications for the user of the mobile device. We will examine these implications, and propose new tools and methods to help understand the nuances of modern malware, as well as provide a framework of tools to detect them.

Chapter 2

Background & Modivation

To understand the nature of modern mobile malware, we first examine the context. The two main models, that of iOS and Android, are compared and contrasted here.

2.1 The "App" and Sandboxing

In the mobile world, "Apps" are isolated and sandboxed programs, generally designed with one singular purpose. They lack dependencies, and generally are not as privileges as system software for performing many tasks. The mechanisms for accessing functionality outside of their sandbox is enforced by a set of policies the system holds, specific to that app. On some platforms, like iOS, only one app may run at any given time, and background computation is virtually non-existent (with some exceptions)[1], along with many other restrictions. On Android and other platforms, many more features are available to apps, but in all cases, the "app" lifecycle is well defined and controlled by the system much more than on a PC OS.

There are various reasons for the tight sandboxing of mobile apps. Power and resource consumption are certainly a factor - mobile OSs generally reserve the right to kill apps if they attempt to allocate too much memory. Controlling access to hardware also helps in this: allowing apps to keep the phone awake could easily drain battery. However, another reason for sandboxing, and arguably more important, is protecting Personally Identifiable Information.

2.1.1 Persionally Identifiable Information

Personally Identifiable Information (PII), as defined by NIST, is "any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individuals identity... and (2) any other information that is linked or linkable to an individual" [8]. Mobile devices, having replaced cameras, cell phones, GPS devices, and PCs, have an extremely diverse amount of PII, from phone numbers, to contacts, to location history, to bank account numbers and pictures. For many of these datasets, mobile OSs actually organize them into databases with the intention of allowing 3rd parties access to them. Contact lists, SMS, Photographs and location

[1] Minor amounts of computation can be done to compute background audio, and other isolated background tasks.

history are available to apps on virtually every mobile platform in some official way. This is a driving motivation for a greatly improved security model for mobile OSs: controlling 3rd party software's access to PII.

2.1.2 The "App Stores" and Controlled Distribution

The final major difference between mobile OSs and PC OSs is the distribution of code. No mobile OS allows code to be ran outside of the sandbox, and all of them require the user to agree before installing an app. All apps must be signed, and in general, there is 1 main distribution channel for all apps on a mobile OS. This tightly controlled distribution both aids in security, as well as controls the ecosystem around that mobile OS.

The first major distribution platform for mobile apps was Apple's App Store[3]. It's model has been repeated by almost all major mobile app distribution platforms. The basic premise is simple: developers sign up to the app store, pay a fee (usually yearly), and submit fully-finished apps. A reviewer runs the app in a monitored sandbox, watches for unusual behavior, checks for stability and usability, and approves it. Once the app has been approved, it's released onto the app store, at which time anyone can download it. The approval process, as well as the high monetary fee, act as a way to ensure only safe and high-quality apps are available for that platform. In these times of platforms, typically no apps may be installed from other sources. On iOS, initially this was the main method of security: if the app passed the inspection, it was acknowledged as safe and virtually unmonitored unless someone noticed something unusual and reported it. However, in recent years, after certain incidents, apps still must request permission from the user to perform certain tasks.

2.1.3 Android Permissions

Android's distribution platform takes a different approach, and at it's core is also Android's core security model: The Permission System. Android Apps declare when they are packaged what capabilities they will use, and the user reviews them at install time. If the user approves the app, it may use the requested capabilities whenever it wants: little restrictions are placed otherwise. With this barrier in mind, the Google Play Store (formally the Android Market), or GPStore, opts for an alternate model to iOS, where the developer pays a smaller fee, and apps go through no formal approval process. After an app's submission, it's immediately released into the wild for users to download and run. The assumption Android uses is that the metadata the GPStore provides: App name, Developer Name, Description, Reviews and Ratings, are enough for the user to determine if the app should be trusted with the permission set it's given. In fact, Android even allows the device to accept apps from 3rd party sources, a practice known as "sideloading", although it's disabled by default. This has spawned a large number of 3rd party app sources, all of which rely on the Permission system for user protection.

This core concept behind Android security - Permissions as a static contract of capabilities, is the classic example of a rule-based security model. ?expand on this?. However, in this paper we propose an alternate means of conceptualizing security, which focuses on the interrelationship between the user's perception of the app, and the app's actual behavior. We call this the user-app agreement', and will elaborate on it more shortly.

2.2 Mobile Malware

The tighter security model of Mobile OSs has a notable effect on mobile malware. With tight control in sandboxing, and app distribution, the usual viruses, trojans, and other exploits are more difficult to employ. The main vectors are either OS-level exploits, sneaking past the app review process, or through sideloading of apps. When looking at the two main mobile OSs, a stark contrast is shown. Over ?201 or so? known OS-level exploits exist for iOS, where only ?5 or so? have been publicly released for Android. On the contrary, no side-loading is possible for iOS, and there have been very few instances of malware sneaking past Apple's App Store review process, although it has happened[2]. Android's malware situation is very much a product of the sideloading and lack of review process found in GPStore. Of all the mobile malware found for iOS and Android, over ?some percent? used no system-exploits at all, ?some percent? on the main app distribution platform.

Malware, as defined by the US-DHS, is "Short for malicious software. Programming (code, scripts, active content, and other software) designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and other abusive behavior" [9]. On mobile devices, the dominant goal of malware is to gather information that leads to loss of privacy, found in over ?some large percent? of known mobile malware.

This trend, of malware that possesses no system exploits, but gathers information that leads to loss of privacy, is one that Android's Permission-based security model is ill-equipped to handle. Android's permission system relies on the user to determine at install-time if a list of capabilities should be entrusted with the given app. The user is not given a say in how or when the capabilities may be used, nor the ability to reject specific capabilities. At the same time, the mechanisms that keep mobile OSs safe are forcing malware writers to use more subtle techniques, often times without exploits. This all works against the user.

In this paper, we attempt to address this key issue through various means. We first introduce several novel concepts for analyzing apps and malware on Android. We then analyze the state of Android apps with the most comprehensive android app database available. Finally, we propose several novel improvements to the Android security architecture, called AndroMEDA, aimed at building off of our conceptual work.

[2] In July 2012, SecureList noticed an iOS app that uploaded all of the user's contacts to a remote location without their consent. [7]

Chapter 3

Permissions & Security on Android

3.1 Android Architecture Overview

Android is an open source project, built on Linux. Designed to be lightweight, modular, extendable and versatile operating system, Android strips away almost all of the typical GNU/Linux stack, and wrote an entire framework from scratch. Built in Java, Android runs the Dalivk VM, a lightweight Java-compatible VM.

The four major application components of Android are Activities, Services, and Content Providers. They are all tied together through the Intent system. Activities are user-facing tasks, and follow the iOS definition of an "app". Only one may run at a time, and they have strict lifecycles. Services run in the background, and follow a less strict lifecycle. Their main purpose is to perform long-running tasks that do not require user input. Lastly, Content Providers "manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process" [1].

Android was built from the ground up to be composed of strongly isolated modules with little dependencies. No traditional SysV IPC is allowed, instead Android provides it's own inter-app communication built off of it's Intent system. Intents on Android, as described in the documentation, are "An intent is an abstract description of an operation to be performed... An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed" [2]. Intents allow apps to describe the operation they'd like to perform, without explicitly identifying a recipient. As an example, when the intent *ACTION_VIEW* is sent with data "http://google.com", Android searches through all installed apps that designate that they respond to that intent, and will pick one to deliver it to.

3.2 Android Permissions

The highly modular and decentralized aspect of Android makes it extremely easy to tap into virtually all Personally Identifiable Information on the device. To protect this, and many other aspects of the system, Android utilizes the Permission security model. The permission security model is a static list of capabilities an app possesses: when presented with this list at install time, a user will either grant the app access to the features, or simply not install the app. When an app requests a permission, the Android system treats it as if the user granted the app that capability. After install, this list will never change unless the app package itself changes, and the user reviews the new permissions.

Android permissions themselves are much more granular than a typical unix permission system. They cover a wide variety of operations, from controlling the sleep state, accessing hardware, accessing PII, and many system operations. Some of the most requested permissions can be seen in 3.1.

In cases like *WAKE_LOCK* and *CAMERA*, the permission seems fairly singular: access to exactly one feature. However, for other permissions, like *INTERNET* and *READ_PHONE_STATE*, many more granularities could be established, as like *INTERNET* gives unconditional access to all domains, unlike webbrowsers. In addition, whereas permissions are intended to be non-overlapping, there still are ways, varying from minor to major, to acquire information guarded by one permission from another. A simple example would be *READ_PHONE_STATE* would get access to the current phone call, thus being able to establish call logs, something normally protected under *READ_CONTACTS*. However, there are some permissions that are explicitly supersets of another other, like *ACCESS_COURSE_LOCATION* - providing network-tower location, and it's superset, *ACCESS_FINE_LOCATION* - providing GPS location.

Permissions do not tend to change through Android's release history. As new hardware is made accessible through Android's SDK, new permissions are added for them, but there are very few times when permissions drastically change meaning or scope. Through Android's 6 year history, only 1 new permission has been added, where previous apps always had access.

3.3 Permission Enforcement

Permission enforcement on Android is generally performed in two main ways: UNIX permissions and explicit runtime checking. Most hardware is accessible using c or other low-level calls, so permissions are more effectively enforced via UNIX Group Permissions. On app install, Android assigns each app a unique UID, and assigns different group permissions to that user. For example, socket access is granted to a UNIX group, and all apps that request INTERNET are added to that group. This is simple, effective, and has very little performance overhead. Unfortunately, it makes it difficult to put any additional enforcements as to when and how these resources are accessed.

Permission	Description
INTERNET	Network communication. full Internet access. Allows the app to create network sockets.
WRITE_EXTERNAL_-STORAGE	Storage. modify/delete USB storage contents modify/delete SD card contents. Allows the app to write to the USB storage. Allows the app to write to the SD card.
READ_PHONE_STATE	Phone calls. read phone state and identity. Allows the app to access the phone features of the device. An app with this permission can determine the phone number and serial number of this phone, whether a call is active, the number that call is connected to and the like.
ACCESS_FINE_-LOCATION	Your location. fine (GPS) location. Access fine location sources such as the Global Positioning System on the tablet, where available. Malicious apps may use this to determine where you are, and may consume additional battery power.
ACCESS_COARSE_-LOCATION	Your location. coarse (network-based) location. Access coarse location sources such as the cellular network database to determine an approximate tablet location, where available. Malicious apps may use this to determine approximately where you are.
WAKE_LOCK	System tools. prevent tablet from sleeping prevent phone from sleeping.
READ_CONTACTS	Your personal information. read contact data. Allows the app to read all of the contact (address) data stored on your tablet. Malicious apps may use this to send your data to other people.
CALL_PHONE	Services that cost you money. directly call phone numbers. Allows the app to call phone numbers without your intervention. Malicious apps may cause unexpected calls on your phone bill. Note that this doesn't allow the app to call emergency numbers.
CAMERA	Hardware controls. take pictures and videos. Allows the app to take pictures and videos with the camera. This allows the app at any time to collect images the camera is seeing.
WRITE_CONTACTS	Your personal information. write contact data. Allows the app to modify the contact (address) data stored on your tablet. Malicious apps may use this to erase or modify your contact data.
GET_TASKS	System tools. retrieve running apps. Allows the app to retrieve information about currently and recently running tasks. Malicious apps may discover private information about other apps.
RECORD_AUDIO	Hardware controls. record audio. Allows the app to access the audio record path.
SEND_SMS	Services that cost you money. send SMS messages. Allows the app to send SMS messages. Malicious apps may cost you money by sending messages without your confirmation.
READ_HISTORY_-BOOKMARKS	Your personal information. read Browser's history and bookmarks. Allows the app to read all the URLs that the Browser has visited, and all of the Browser's bookmarks.
READ_CALENDAR	Your personal information. read calendar events plus confidential information. Allows the app to read all calendar events stored on your tablet, including those of friends or coworkers. Malicious apps may extract personal information from these calendars without the owners' knowledge.
WRITE_HISTORY_-BOOKMARKS	Your personal information. write Browser's history and bookmarks. Allows the app to modify the Browser's history or bookmarks stored on your tablet. Malicious apps may use this to erase or modify your Browser's data
RECEIVE_SMS	Your messages. receive SMS. Allows the app to receive and process SMS messages. Malicious apps may monitor your messages or delete them without showing them to you.
WRITE_CALENDAR	Your personal information. add or modify calendar events and send email to guests without owners' knowledge. Allows the app to send event invitations as the calendar owner and add, remove, change events that you can modify on your device, including those of friends or co-workers. Malicious apps may send spam emails that appear to come from calendar owners, modify events without the owners' knowledge, or add fake events.
MOUNT_UNMOUNT_-FILESYSTEMS	System tools. mount and unmount filesystems. Allows the app to mount and unmount filesystems for removable storage.
READ_SMS	Your messages. read SMS or MMS. Allows the app to read SMS messages stored on your tablet or SIM card. Malicious apps may read your confidential messages.
READ_LOGS	Your personal information. read sensitive log data. Allows the app to read from the system's various log files. This allows it to discover general information about what you are doing with the tablet, potentially including personal or private information.
DISABLE_KEYGUARD	System tools. disable keylock. Allows the app to disable the keylock and any associated password security. A legitimate example of this is the phone disabling the keylock when receiving an incoming phone call, then re-enabling the keylock when the call is finished.

Table 3.1: Frequently Requested Android permissions, and the GPStore's description of them

System operations, like wake locks, changing system settings, turning on and off components, is checked on every API call, through a centralized PackageManager class. The PackageManager first checks if the app has requested that permission. If it has, it then proceeds to check if the permission is protected by the system. Some permissions are only available to trusted code, through either a shared key, or being located in a system folder. Many system operations fall into this category, including *WIPE_DEVICE*, and a personal favorite: *BRICK*. These operations typically are extremely dangerous, and have the potential to destroy a device, or perform elaborate phishing operations. Even if an app requests the permission, the PackageManager still has the right to reject the operation. This allows system-level operations to be exposed to trusted developers post-deployment on a device, while still protecting the operations from untrusted developers.

The final aspect of permissions deal with protecting PII. Android implements the bulk of PII sources through Content Providers, which sit in front of a dataset and provide access to remote services. It is therefore imperative for each individual data source to check the permissions of the incoming app, every time it's requested. This, however, provides opportunities to extend on it's behavior.

3.4 Third Party Permissions

Android was built to distribute PII in a modular fashion, and it extended these features to other 3rd party apps. Any developer may write Content Providers, and likewise, can create Permissions to protect them. Other apps must request that permission before accessing the data. An example of this: Alice writes a messaging app, MyMessenger. She wants to expose the PII to other apps, so she makes a Content Provider around it. To protect the user's information, she defines her own permissions: mypackage.MyMessenger.READ_MESSAGES and mypackage.MyMessenger.WRITE_MESSAGES. Bob writes an app that uses MyMessenger's data to build a picture. His app requests the permission mypackage.MyMessenger.READ_MESSAGES. When his app contacts Alice's Content Provider, she checks this custom permission before proceeding with the query. However, if Bob does not request mypackage.MyMessenger.WRITE_MESSAGES and attempts to perform a write action, Alice's Content Provider will reject the operation.

If two apps request the same permissions, as long as neither are signed with the system key, it can be assumed that they have the same capabilities. These capabilities are permanent as well, as permissions can not be added nor removed. This Permission Fingerprint, or set of capabilities, uniquely defines what an app has access to. All apps that request the same set of permissions have access to the exact same set of actions, and only through those permissions do they have those actions (with a few exceptions). However, Android permissions are static, therefore a Permission Fingerprint doesn't guarantee a specific pattern of behavior. In fact, since not all permissions are guaranteed to be

granted, a Permission Fingerprint simply establishes the absolute maximum capabilities of an app - even if the system rejects some of them.

Chapter 4

Malware on Android

Malware on mobile devices has seen a departure from past exploits. The wealth of Personally Identifiable Information easily available on mobile OSs has made them the increased focus of malicious software. In addition, the tight sandboxing constraints have often forced malware writers to either find exploits to break out of the sandbox, or to write malware that cloaks itself as benign. Without finding exploits, code may not be ran by the user unless it is in *app* form, and come through a trusted channel (unless that security feature has been disabled). This has caused two main vectors of attack: finding exploits to remotely install software, or masquerade as benign apps, and pass through trusted channels. On Android, this last technique is especially popular.

After masquerading as a benign app, malware on Android has three main techniques: privilege escalation attacks, monetary service attacks, and PII stealing attacks. The first one, privilege escalation attacks, takes many forms on android. The basic premise is simple: acquire access to operations beyond what the sandbox and granted capabilities provide. The main way to accomplish this is via *rooting*.

Rooting is the act of acquiring root privileges on an OS. Typically mobile OSs do not provide the user, or apps, with root permissions, and instead reserve that for a set of trusted system processes. However, by finding vulnerabilities in these services, or exploiting the OS itself, apps can escape the sandbox. After an app has been granted root capabilities, the permission system no longer applies to it: it can simply access whatever it wants. These attacks are difficult for the system to detect: all monitoring of apps relies on monitoring the sandbox - when an app escapes that, theres no tracking it. This technique is commonly employed by botnets - giving remote access to the core system.

Many examples of root exploits exist, dating back to 2011 with *RageAgainstTheCage*[6]. These root exploits were very popular for nonmalicious purposes, circumventing the device's sandbox to install a permanent root binary, creating a similar state to a typical UNIX computer, where root may be acquired after a password/permission. However, in March 2011, DroidDream was discovered. DroidDream used *RageAgainstTheCage* to silently install additional applications in the background. From there, it proceeded to steal PII and become a botnet. By the time it was remotely removed from the market, it had been downloaded an estimated 50,000 to 200,000 times[4] - the largest bulk-remote-removal of apps seen from the GPStore. From then on, a stream of root exploit malware was found, based off of *RageAgainstTheCage*, *udev*, and one exploit called *GingerMaster*[5].

Several exploits became very popular for malware writers, starting with DroidDream in 2010 ¡cite¿, then RageAgainstTheCage, PSNeuter, and GingerBreak. When DroidDream was first discovered, it was downloaded over ¡50,000¿ times in the GPStore - prompting the largest bulk-remote-removal of apps Google has ever been discovered to have performed. However, most of these attacks have been fixed by vendors as of Android 2.3.3 - meaning the total amount of current devices in use that are vulnerable to these bugs being roughly ¡10

The second main vector for privilege escalation attacks is the Confused Deputy attack. In this scenario, services that guard sensitive operations are "tricked" into performing them. An example of this would be if a Content Provider forgot to check a permission, or finding APIs that do not correctly perform a permission check. Perhaps the simplest example of this is the ability for any Android app to contact remote servers, simply by asking the web browser to open a url. By passing sensitive data in the URL, an app may still contact a remote server without ever requesting the INTERNET permission. Despite all of these concerns, its worth noting that no known malware has been used to exploit this.

The second malware technique is possibly the simplest: perform services on behalf of the user that cost money. Examples of this include calling costly phone numbers and sending premium SMS messages. Typically these actions are performed without notifying the user, and only visible after the user checks their sent messages, or calls. These attacks have been prevalent in the Android market for quite a while, but recent versions of android - after 4.2 Jelly Bean - have taken the step of warning the user before premium SMSs are sent. This attack vector, while common, will see a reduction in the future thanks to these measures.

The last malware technique is the most significant, and represents the largest departure from typical malware. Apps that steal PII. The theme is fairly straightforward: Provide the user with a seemingly legitimate app, but in the background acquire large amounts of valuable data, from call logs to contacts to photos, and send them to a remote server. This fits in well with the main themes of mobile computing: the consolidation of many sources of PII all in 1 device. However, this is the biggest departure from typical malware. To the system, no unusual operations are performed, and no exploits are ran. The qualification for malware in this category thus lies in the "use" vs "misuse" of PII. Often times, this line is blurred.

A large distinction in what constitutes as privacy malware to an individual stems from their expectations of how the app will use their PII. Consider the case of the Path iPhone app, which in Feb 2012 was discovered to be uploading the users entire contact list to Paths servers, without any permission from the user. Its fairly uncontroversial for a social network to read your contact data, and the act of scanning contacts to help "find your friends" on Path wasnt out of the ordinary, but the unexpected behavior of how it handled very personal data was startling for many people.

The Path incident sparked several key changes in iOSs security model: specifically, having a popup occur when an app requests access to the contacts database, and allowing the user to reject the request. This, in general, is a

one-time request, after which the app is given free reign over content ¡cite and double check¿, which doesnt fully address situations like Path, where its less about the app simply having access to the data, but what the app actually did with the data behind the scenes. When these actions did not match up with user expectations, it was treated as malware until the situation was cleared up by Path. The next day, they issued an update immediately explaining to the user what they were going to do with the data, and why.

Its worth noting that the only reason Paths contacts upload mechanism was discovered was by accident: Arun Thampi discovered it as part of a company hackathon (http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html), and only via sniffing the HTTP requests coming from the phone. An ordinary smartphone user would not have access to these tools, nor have the time and patience to sift through the data to spot unusual behavior. This incident, however, lies at the heart of mobile malware: Misuse of PII lies in the abstract definition of how the app is expected to behave. Apps that violate this expectation of behavior are classified as malware, and apps that do not are not. This agreement between the user and the app, the User-App Agreement, or UAA, is an informal understanding the user has as to what actions an app will take. This differs from the Permission Fingerprint, which is a measure of what actions the app is capable of performing, instead dealing with exactly when and how those actions are taken. Since this agreement is not formally defined, its acquired through external trust in an app. This happens in various ways, through the description the app provides, to the knowledge and referral of the app from other trusted sources, or the trust in the developer. The UAA is not a measure of how trustworthy an app is, but rather a framework for consenting and trusting specific actions an app may take.

An example of UAA can be seen in the expected behavior of two different hypothetical apps and a hypothetical user. One is a large Social Networking app, which requests permission to access internet, send SMS messages and read the contacts database. The other is a little known developers game, which also requests permission to access internet, send SMS messages and read contacts. These apps have the same Permission Fingerprint, but the exact behavior of how they access these capabilities may or may not violate the UAA. If the Social Networking app accesses contacts when the user requests it “find my friends”, and it sends SMS messages after the user messages another user who is not “online”, these actions fall within the UAA of the Social Networking app and the user. However, if the contacts database is read and uploaded to a remote server without the consent of the user, this will violate the UAA (building off the example of Path).

In the case of the game, these documented capabilities in the Permission Fingerprint may be enough to violate the UAA: the user may not trust an app with the capability of these actions. However, in the case that the user does, or simply doesnt pay attention, the app still may not violate the UAA. If the app is an online game, and asks you to find other people you know who are playing it, this would more likely than not fall within the UAA. However, if it sends SMS messages to your friends telling them to download the game, this would quickly violate the UAA.

In both examples, the apps have the exact same Permission Fingerprint, but vary wildly in their expected behavior, and which actions are trusted and untrusted. This fits right along with our definition of malware, with the misuse of PII and other device capabilities. This also highlights the shortcomings of the Permissions framework - being unable to deal with the subtle differences between trusted behavior and untrusted behavior. Indeed, for any given user, they may have a very different understanding of what acceptible behavior is. Therefore, UAA plays a crucial role in classifying apps in relation to PII malware.

In the realm of research malware, several prominent examples further demonstrate the vague line between use and misuse of PII, and the User-App Agreement. The most notable one is SoundComber ¡cite¿. It passes off as a benign app, but in the background records audio, and does on-phone processing to find sensitive PII, after which it uploads the information to a remote server. This app is unique because of its simple Permission Fingerprint, and its ability to gather sensitive PII from a channel not suspected to be very rich in PII.

The second prominent example of research malware on Android is TapLogger. TapLogger imitates a simple touch-based game, learning the vibration patterns of the device for each tap. After which, TapLogger records the vibration patterns in the background, attempting to discover passwords and other sensitive keyboard events, all through a seemingly trusted sensor. TapLogger requests no Permissions, therefore its behavior is a possible behavior of virtually all apps.

summary: The landscape of malware on android follows many clear patterns. The first is the use of masquerading as benign apps, and passing through trusted/semi-trusted channels to enter the device. Once on the device, the three main categories of attacks are privilege escalation attacks, monetary service attacks, and PII attacks. Of these three attacks, privilege escalation and monetary service attacks are the easiest to protect against, and indeed Android has taken serious steps to mitigate these. However, the third time of attack, PII attacks, is the most difficult to mitigate on Android, due to the shortcomings of the Permissions framework, and the wide spectrum of severity these attacks can take. Since these attacks may vary in interpretation per user, and lay in the subtle communication between the user and the app, we highlight the need for a concise UAA.

Chapter 5

Conclusion and Future Work

References

[1] Android Developers - Content Providers. http://developer.android.com/guide/topics/providers/content-providers.html.

[2] Android Developers - Intents. http://developer.android.com/reference/android/content/Intent.html.

[3] Apple App Store. http://itunes.apple.com/.

[4] C. A. Castillo. Android malware past, present, and future. *McAfee),[online]*, 2010.

[5] GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3. http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/.

[6] K. Mahaffey. Security alert: Droiddream malware found in official android market. https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/, 2011.

[7] D. Maslennikov. Find and call: Leak and spam. http://www.securelist.com/en/blog/208193641/Find_and_Call_Leak_and_Spam, 2012.

[8] E. McCallister. *Guide to protecting the confidentiality of personally identifiable information*. DIANE Publishing, 2010.

[9] T. Nash. An undirected attack against critical infrastructure. Technical report, Technical Report, US-CERT Control Systems Security Center, 2005.