A SURVEY OF THE HAZARDS OF USING USB AS A UNIVERSAL CHARGING STANDARD AS PERTAINS TO SMART DEVICES

BY

MICHAEL BRANDON ROGERS

THESIS

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

Urbana, Illinois

Adviser:

Associate Professor Yih-Chun Hu

ABSTRACT

The Universal Serial Bus protocol was designed to be, and has become, the single standard used for interfacing with computer peripherals and electronic components. The proliferation of this protocol has resulted in USB power outlets in public places being a common sight. However, the very universality of the protocol, combined with the rapidly shrinking size of computer processors and microcontrollers, creates some rather severe security vulnerabilities.

Plugging a USB smart device into a compromised port, even one purported to be solely a power delivery system, can allow an attacker to perform many different malicious actions, the range and severity of which depend on the length of time the device remains attached, the specific type of device in question, and the resources of the attacker. This thesis presents a survey of the types of attacks possible against some of the most common and popular devices, a comparison of these devices from the standpoint of defense against these types of attacks, and possible mitigation strategies.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Thanks to my adviser, Professor Yih-Chun Hu, without whom this would not have been possible; to my friends, whose support and commiseration carried me through the tough times; and thanks to my loving family, who never stopped encouraging me.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The USB protocol was originally designed to enable the transfer of both data and power. The average battery life of a smart device and the wireless data capabilities these devices possess, though, have resulted in USB connections being more frequently used for power than for data. However, even if the intent of the user is simply to charge the device, the potential for data transfer is still present if the USB port being used is being driven by a processor such as a computer or microcontroller. In the case of a home or workplace personal computer, this is expected behavior and not necessarily cause for concern (unless the computer has been compromised by malware); in the case of a USB charging station which should not have such a processor, however, this behavior can be used for malicious purposes.

1.1 Proliferation of smart devices

Smartphones in their original incarnation were referred to as personal digital assistants, or PDAs. They were bulky, clumsy, lacked touchscreens, and were significantly more expensive than any other type of phone. Since then, the smartphone has evolved to the point where smartphone users outnumber normal cellular phone users in the United States [1], with models to fit all budgets. This is a self-reinforcing cycle, as the prevalence of smartphones has caused developers to focus on improving smartphone technology, giving rise to smaller, more efficient processors and better communications protocols such as 3G and LTE; these improvements in turn create still greater demand for smartphones.

As smartphones become a part of more people's everyday lives, security for them becomes more and more important. One particular problem comes in the form of perception; cellular phones are viewed as phones, or as mobile ereaders, or as portable social media platforms, when they are actually small, fully-functioning computers. And like any computer, they are susceptible to malware and vulnerable to attacks if proper precautions are not taken.

1.2 Proliferation of USB charging stations

With the proliferation of portable, personal smart devices have come attendant services being offered in public locations. It is not uncommon to see electronic device vending machines in major airports, offering portable batteries, headphones, or even personal electronic devices such as portable gaming systems and music players. One such service which is becoming increasingly widespread is the USB charging station or kiosk.

These kiosks are especially prevalent in areas with a large number of travelers, such as airports, train stations, and hotels. There are even third-party services offering to install and maintain these kiosks as a service [2]. These charging options are becoming more widespread, and their prevalence and utility are leading them to become more widely accepted in light of the increasing number of travelers with smart devices in need of recharging.

1.3 Potential avenues of attack

The fact that USB supports both power and data is what leads to the threats this thesis will analyze. Such threats can come from any USB connection made with a smart device, whether or not the user of the device is aware the connection is capable of being used for more than simply power. This implies there are two different situations which must be taken into account.

The first situation is when the user is aware of the data connection abilities of the system he or she is connecting to. In this case, attacks are caused by prior tampering with the system, normally through infection by malware. A system so infected can then attack a connected smart device, and is particularly insidious because any data transfer activities noticed may be ignored as normal when they are in fact signs of an attack. The problem becomes compounded when the fact that an infected smart device can itself infect a host system is taken into account.

The second situation arises when the victim connects to a USB socket solely for the purposes of charging the smart device, ignorant of the USB host an attacker has connected to the outlet. Such a user may or may not be paying attention to the screen of his or her smart device in such a scenario, depending on the level of charge in the device and other potential draws on his or her attention. Detection of a modified charging station by simple observation is difficult, if not impossible; any marks made by the accessing of the station's internals may be dismissed as normal wear, and microcomputers capable of performing a sophisticated attack while fitting inside a very small space are not only feasible, but available off the shelf. The Raspberry PI computer, in particular, has USB connectivity, is very inexpensive, and is small enough to hide easily, measuring 85.60mm x 56mm x 21mm [3]. Dedicated purpose-built hardware could be made even smaller.

In either event, the problem becomes one of trust. Whether a device is being trusted with or without reason, the problem remains. The following chapters will explore what can be done when that trust is misplaced, and what can be done to guard against the results.

CHAPTER 2

ATTACKS AGAINST IOS DEVICES

The iPhone is, without a doubt, one of the most popular smartphones on the market today. The iPad broke new ground and in many ways opened the door to the current popularity of tablets. Many companies issue iPhones to their employees or only support iPhones, citing ease of mobile device management on the closed platform and the security the closed-source operating system offers, as well as the productivity gains available owing to the large amount of third-party applications available and the ease of creating applications if none of the existing ones serve. However, the USB communication which allows easy synchronization of emails, contacts, and other personal data also presents vulnerabilities which may overlook.

2.1 USB interface

When connecting an iOS device, the host operating system acts as it does for any USB device and binds the appropriate driver(s) based on the device and vendor IDs the USB device reports, along with the device's declared endpoint types. For the iPhone specifically, this consists of a PTP interface (Picture Transfer Protocol, used to access the camera's photograph folder only), the Apple Mobile Device interface (used to transfer media and settings), and optionally a USB ethernet interface (used for internet tethering) [4].

2.1.1 The Picture Transfer Protocol interface

The Picture Transfer Protocol is a communication protocol, rather than defining any particular storage method or format [5]. Because is it solely a communication protocol, implementation of the storage back end is the responsibility of the device, meaning that with regards to this particular protocol, the device is only as secure as the storage handling it presents to the host. The Picture Transfer Protocol is employed in most devices which have a digital camera (or are exclusively digital cameras).

While the Picture Transfer Protocol does allow for live real-time communication between the camera itself and the host computer, this functionality is not enabled on iOS by default (there exists third-party software which allows for this, but not via the PTP USB endpoint and as such are beyond the scope of this thesis).

The Picture Transfer Protocol allows the connected host device sandboxed access to the camera's storage area only, making it mostly unsuitable for security exploits beyond theft of photographs. However, some iOS devices, such as the iPad, actually implement a generic USB mass storage device class as opposed to the PTP class, and as such are considerably more vulnerable [6].

2.1.2 The USB Ethernet interface

The Apple USB Ethernet interface offered by iOS devices presents some endpoints of the device as a vendor-specific USB device, which then requires a specific driver for the host to populate the iOS device as an ethernet connection. This interface is provided primarily for use by cellular data-enabled devices such as the iPhone and some versions of the iPad, so that cellular data may be used to connect to the internet in areas where a standard internet connection is not available. The driver used on both Windows and Mac OSX is Apple's proprietary driver; however, open source versions which are compatible with Linux are available [7].

It is interesting to note that while support for internet tethering is native to iOS, it requires support from the cellular network provider and is routed through a different system than normal device-based cellular data traffic so that tethered internet traffic can be tracked (and billed) separately. Since cellular data providers charge an additional fee for this service, there are several options available for jailbroken devices which configure the device to proxy traffic from a host through the device's built-in cellular data connection, bypassing the native device functionality as well as the service provider's checks on tethered data usage.

2.1.3 The Apple Mobile Device interface

The Apple Mobile Device interface is the primary interface through which a host computer communicates with an iOS-based smart device. On Windows and OSX systems, it provides the interface for iTunes, Apple's media management software; on Linux systems, open-source third party drivers use it to provide a link to other media management software such as Rhythmbox.

On a normal, non-jailbroken iOS device, this interface provides access to the user data partition on the device via the Apple File Connection (AFC) service [8]. This service acts as an interface between the host computer and the filesystem on the device. Files can be read from or written to the device via calls to the AFC service; the AFC service also offers other filesystem commands such as creation of symlinks and hardlinks and setting file sizes and modification times.

Jailbroken devices

On a jailbroken device, a second service (denoted Apple File Connection 2) is installed and set to run with root permissions at boot time. This AFC2 service is functionally identical to the AFC service, but where the AFC service is jailed to the user data area of the device, the AFC2 service has access to the system data partition, allowing low-level root access to the iOS operating system itself. Effectively, jailbreaking has the end goal of installing this service in a stable manner, because with the low-level system access this grants any and every conceivable change to iOS itself becomes feasible [8].

2.2 Existing attacks

The popularity of the iPhone has made it a target of many attacks, despite Apple's best efforts at policing their App Store and their requirement that any installed applications be digitally signed. This is due in part to the comparatively large number of vulnerabilities present in iOS. As recently as March of 2013, the iPhone was found to have more vulnerabilities present than Windows Phone, BlackBerry, and Android combined [9].

Some of these vulnerabilities are what enable jailbreaking, while others are used simply to attack the phone. Moreover, jailbreaking itself lends a device to attacks more easily. The existence of an untethered jailbreak implies that the device is susceptible to boot-time code injection independently of an attached host device, at least after the initial exploit is executed against the phone in the first place.

2.2.1 Stock iOS

Attacks against a smart device running iOS can range from the simple to the sophisticated. The first class of attacks, and the simplest, are not truly active attacks at all; they simply take advantage of the fact that the entire user data partition is available over the data connection. These attacks take the form of data theft. Because iOS and iTunes support synchronizing of emails, contacts, and calendars along with photographs, music, and videos, all of this information is freely accessible to any host computer an iOS device is connected to, regardless of whether or not the device is locked with a PIN. Also, because the AFC service is separate from the mobilesync service, reading from the filesystem does not trigger the on-screen synchronization notification, leaving victims totally unaware that all of their personal data is being copied directly off of their phone.

The next level of attack is slightly more active, and is not aimed at the smart device itself. Rather, it is targeted at any host computer the device normally synchronizes with. Similarly to reading, it is possible to write data to the user partition via the AFC service without triggering any alerts to the victim. It is also possible to perform mobilesync's job manually, registering new files with the device by manipulating Apple's .plist files directly. This was tested experimentally by connecting an iPhone 3G and an iPhone 4 to a compromised virtual machine running Linux. Upon connection of the iDevice, a ringtone was copied to the device and registered with the system manually, bypassing the use of mobilesync. No data transfer or synchronization notification was given; the only way to tell that anything had been done was to look at the list of ringtones and observe the presence of a new one. A naive attacker may plant a link to an infectious website and hope that the victim will follow it. A sophisticated attacker will copy an existing file, such as a document or a photograph; embed a virus or trojan within it; and copy it back to the device, overwriting the original so that a file which the victim has every reason to believe is valid and supposed to be there is in fact a piece of malware waiting to be executed. This may be done over either the AFC or PTP communication channels, although doing so over PTP requires manipulation solely of image files where doing so over AFC allows the attacker to target any type of data file present on the device.

A minor next step up the ladder of attack sophistication keeps the same target of a victim's host computer, but applies to devices which implement the USB mass storage device protocol rather than the PTP protocol, such as the iPad [6]. USB mass storage devices encompass such devices as memory sticks and external hard drives, and as such are capable of executing commands as soon as they are connected to a host via autorun. While some anti-virus software will disable autorun on mass storage devices specifically to guard against this sort of attack, not all do, and not all host computers run anti-virus software in the first place. If autorun is not disabled, then any executable file can be made to run as soon as the compromised device is connected, allowing the immediate execution of malware.

The final class of attacks aims at the smart device itself. Apple attempts to guard against malware being installed on an iOS device by sandboxing applications and requiring each installed application to be signed with a valid Apple developer ID and bound to that specific device. There is, however, a way around these precautions, which is the Apple provisioning profile, used for developers to be able to push builds of their own applications to their own devices prior to release for testing purposes. A malicious USB charger running a microcontroller or miniaturized computer can deploy such a profile to a connected device, then install a malware application. This particular technique has both advantages and disadvantages. Because it uses Apple-signed profiles, it allows malicious applications to be installed without jailbreaking the device. However, acquiring the provisioning profile requires a developer's account, and only 100 devices may be given provisioning profiles per developer's account; additionally, there is currently no automated method for removing a registered device from a developer's account. Couple this with the fact that developer's accounts cost money, and this attack becomes self-limiting fairly quickly, at least as pertains to number of devices that can be attacked by the same setup. Additionally, unlike the previous types of attacks, this one requires the screen to be unlocked, at least briefly [10]. While quantity is not a viable option for this attack, though, quality is; because the user file system is readable, it is possible to assess a connected smart device for information on the owner. If the owner is judged to be a sufficiently high-priority target with an unlocked device, a provisioning profile can be generated and malware installed.

Finally, if none of these types of attacks are sufficient for the attacker's purposes, the attacker is left to either target jailbroken devices, or jailbreak the connected device manually. While most modern jailbreaks require the device to be rebooted in DFU (Device Firmware Upgrade) mode, there exists a history of jailbreak techniques employing vulnerabilities present in the iOS stock software libraries which do not require a device reboot. This includes the relatively infamous 'Slide to Unlock' website which exploited an overflow in libtiff [11]. Depending on the version of iOS the connected device is running, it may be possible to silently jailbreak it, at least to the point where Apple's digital signing can be circumvented.

2.2.2 Jailbroken Devices

By default, the jailbreaking process installs several additional applications meant to enhance the usability of the newly available features and facilitate the installation of jailbroken applications. One such application is the Bourne-Again SHell, or bash [12], which is installed along with a default password of 'alpine' for both the normal user account and the root account. Many users opt to install OpenSSH for ease of remote access and control of their jailbroken iDevice; however, it is far from uncommon for users to neglect to change the default password, allowing anyone with an SSH client and knowledge of the default password to gain full root access to their device. Since SSH can be tunneled over a USB connection [13], this presents a vulnerability in the case of a compromised USB charger even if the device has all wireless radios disabled. Examples of iPhone malware which exploit this (via wireless communication) have already been found in the wild, in the forms of the iKee and Privacy.A worms. The iKee worm was largely a proof-of-concept and fairly benign, disabling the SSH service on infected devices and altering the device wallpaper before attempting to spread in order to patch the very vulnerability it exploits. The Privacy.A worm, however, is actively malicious, stealing the information stored on compromised devices and giving no overt sign of infection [14–16].

In addition to the vulnerabilities jailbroken applications themselves may inadvertently introduce, jailbreaking an iOS device provides two primary avenues of attack for a malicious connected host. The first is simply a method for circumventing Apple's digital signing protocols, enabling the installation of arbitrary applications and the execution of arbitrary code. This was one of the grounds under which Apple was fighting the legality of jailbreaking [17]; by circumventing Apple's signing practices, it becomes possible to install illegally downloaded pirated applications without paying the developers of the software. This same ability allows for the installation of malicious applications, either as system services running with root permissions or, as in the case of unjailbroken devices compromised with a provisioning profile, the installation of what appears to be an innocent or innocuous application which is, in fact, malicious.

This leads into the second avenue of attack. Jailbreaking a device allows root access to the operating system itself, allowing many activities and modifications which are not possible using solely Apple's development kit for iOS applications. Legitimate uses of this access are applications such as Winterboard, which allows full customization of the user interface; SBSettings, which allows the enabling and disabling of some system settings without going through the settings application, as well as the hiding of icons; and modification of or addition to stock system sounds for notifications. This same access allows malware authors to effectively rootkit a jailbroken device running iOS, hiding any trace of their malicious application from the user's notice [11].

2.3 Potential attacks

This section covers theoretical attacks which have not yet been observed experimentally or 'in the wild', but which are nevertheless possible and which may be seen in the future. Since it is all but impossible to predict specific vulnerabilities, this section will restrict itself to speculating on the possible targets of exploits, without regard to the precise mechanism or mechanisms behind the attack.

2.3.1 Stock iOS

It is likely that any new future attacks aimed at compromising non-jailbroken iOS devices will focus on circumvention Apple's protection protocols. The ability to freely read the user partition, which comprises an entire class of data theft attacks, does not require additional attacks at this time. However, the ability to install malware on a smart device with an eye to future exploitation and data theft does require the ability to install applications on a connected device, which in turn requires attacks aimed at Apple's protection mechanisms. Possible exploits of this type include ways to acquire additional provisioning profiles, perhaps counterfeiting them or finding a way to reuse them. It is unlikely, although not impossible, that silent jailbreaks will be seen as part of an attack in the future, due to the increasing experience Apple is accruing in blocking potential jailbreaking exploits.

Another possibility is an attack against Apple's bootloader; such attacks are the cornerstone of current jailbreak techniques, patching the bootloader to inject jailbreak code when the device is rebooted. Because firmware update (DFU) mode can be triggered via standard USB communication, it is possible to modify system files at boot time on a connected device. Such a process is time-consuming and obvious to the victim, but still possible.

Attacks which have not been seen used but which are possible today using existing techniques are primarily of the data theft type, stealing all of a user's emails, contacts, and documents for later exploitation in phishing, spamming, and identity theft purposes. Another possible attack is the aforementioned targeting of the user's host devices, either at home or at work, through the use of infected data files copied to the connected device; this type of attack has already been seen in the Stuxnet worm, although it targeted USB flash drives rather than smart devices. Stuxnet employed a vulnerability in the way Windows handles shortcut files which allowed for the propagation of infection even if autorun was disabled [18]; this technique could be applied to infected iOS smart devices as well.

2.3.2 Jailbroken devices

Jailbroken devices provide a much greater range of possibilities for future attacks due to the low level root access to the operating system the jailbreak grants. It is possible to see several different classes of attack against jailbroken devices, depending on the sophistication of the attackers and their end goals.

The first class of attack is an extension of data theft. It is possible to insert a recording application into the phone stack of a jailbroken iPhone and record phone conversations, which can then be sent over either a wireless or cellular data connection to a central server. There has already been work done on using the accelerometer built into iOS smart devices for use in recording keystrokes [19] as well as using the built-in camera to perform virtual reconnaissance of a location the attacker does not have physical access to [20]. These attacks are classed as potential future attacks because the proof-of-concept applications have several issues which would need to be resolved before they become viable threats, but the work is demonstrably being done. Another proof-of-concept being worked on is the equivalent of a keylogger for touchscreen-based keyboards, allowing the theft of passwords and personal information typed into the device itself [21].

The second class of attack is not aimed directly at the smart device itself, but rather aims to leverage the infected smart device as a malware delivery vector targeted at one or more host computers. It is possible to modify the USB device descriptor the smart device presents to a connected host. While some anti-virus programs disable autorun for devices classed as USB mass storage device hard disk drives, USB mass storage device CD-ROM drives are still allowed to execute autorun files. Altering the existing descriptor, or adding a new endpoint, would allow the infection of a host machine immediately upon connection to the infected device. Even in the absence of such an exploit, there are other ways to infect a host machine upon connection [18].

CHAPTER 3

ATTACKS AGAINST ANDROID DEVICES

Where iOS is one of the most popular mobile operating systems on the market today due largely to its simplicity and user-friendly interface, Android enjoys a large market share [22] for the opposite reason. Android is a fully open-source operating system, with the kernel source code freely available. Many entirely custom versions of the Android kernel and operating system such as CyanogenMod are available for many smart devices [23, 24]. However, this openness and ability to customize comes with a concomitant price in security. It is possible to gain root access to a smart device running Android over USB, and it is relatively easy to write a piece of malware which can be installed and run at the kernel level. Furthermore, Android supports its own brand of shell over USB in the form of ADB, or the Android Debug Bridge [25].

3.1 USB interface

The USB interface of an Android smart device is an interesting study. The Android kernel itself is based on a Linux kernel, and is capable of functioning in both USB device mode (where the Android device acts as a subordinate USB device for a host system) and USB host mode (where the Android device acts as a host system, allowing it to support USB peripherals such as input and storage devices) [26]. USB host is similar to the default USB device support implemented in a standard Linux kernel; in order for the Android device to function in device mode, however, Android uses the Gadget package of kernel drivers, which has built-in support for the device to identify itself as any number of different classes of device, and is further user-extensible in the form of kernel modules [27]. This thesis will focus on the most common and default USB connection types, and will touch on some of the more dangerous available configurations.

3.1.1 Default Android

Android has been through several versions at this point in time, and different versions interface over USB in different ways. At first, the filesystem of the device was made visible to the host system as a normal USB mass storage device, similar to a flash drive. This had several drawbacks, including the need to keep system files and user files on different partitions to keep users from inadvertently modifying import system data; also, while the unit was mounted as a mass storage device, any other attempts to write to the same filesystem would result in data corruption or fail outright, requiring the device to essentially suspend most processes while it was connected. In order to overcome these problems, Android migrated to using the MTP (Media Transfer Protocol) interface starting with Android 3.0, Honeycomb [28]. The Media Transfer Protocol is similar to the Picture Transfer Protocol employed by iOS, but where PTP is optimized and designed for image files, MTP can handle various different types and sizes of files more robustly. Because MTP mode acts similar to PTP mode in that it is a protocol and access to the filesystem is governed strictly by the USB device, it eliminates the need for separate partitions on disk and allows the operating system total control over input and output, eliminating the need to suspend processes while connected. MTP also offers similar security to PTP, and for the same reason: the files accessible by the host system are governed by the device and so as long as the device does not reveal any sensitive files, there is little damage that can be done over USB.

3.1.2 USB debugging enabled

In order to facilitate application development and to allow ease of communication and control with an Android device, the Android operating system supports developer mode. This mode adds an endpoint to the USB descriptor for the Android Debuging Bridge, or ADB. ADB effectively offers a remote shell via USB, through which data can be transferred, the entire device filesystem can be browsed, commands can be run, and applications can be installed or uninstalled [25]. This option is disabled by default; it must be explicitly activated through the device developer settings menu, which is itself hidden in recent versions of Android [29]. If enabled, however, it grants a level of access greater than that of AFC in iOS devices. Where AFC grants the ability to read and write the entire user filesystem, ADB grants the ability to read, write, and execute over the entire device filesystem, including system files, with normal user permissions. The only restriction on this access is that it normally lacks root permissions, and there is no built-in method for acquiring those permissions. Gaining a root shell requires either a rooted device, or the use of a local root exploit. The discovery of such exploits, however, are frequently what allow a device to be rooted in the first place, and so any device which has a publicly available rooting procedure may be at risk of being fully compromised via USB in the event debugging mode is enabled.

3.1.3 Rooted devices

Rooting an Android device does not, in and of itself, present a new or altered set of USB endpoints to a host system. The primary difference that rooting a device makes is to potentially enhance the level of access the ADB shell has. On an unrooted device, the command ‘su’ fails, and it is impossible to gain root privilege using built-in system commands; on a rooted device, the command ‘su’ succeeds, although most rooting procedures also install an application which prompts a user on the device’s screen to confirm or deny root access. This prompt ostensibly removes the ability to exploit a connected Android device with root privileges stealthily, although it is possible to combine such an attack with social engineering techniques and convince the victim to allow such access.

3.1.4 Custom kernels / ROMs

Because the Android kernel employs the Gadget USB driver to connect to host devices, there is a staggering array of possible ways for an Android device running a custom kernel to identify itself to a host. Effectively, a custom kernel can identify itself as any existing type of USB device, or create a custom device type which works with a custom driver on the host system. There are two ways in which this may be accomplished via the Gadget driver.

The first way is to compile the USB driving code into a kernel module, which can then be loaded into the kernel like any other module. This requires shell access, which is available either through ADB or a terminal program installed on the device itself; it also requires root permissions, which means this can only be done on rooted devices. The Gadget driver can only support one such module at a time, however, so all desired device types must be combined in the same module, the current module unloaded, and the new module loaded in order to affect the descriptor presented to the host.

The second way is to compile the desired module into the kernel itself. Because the module becomes part of the kernel, it cannot be unloaded; altering it requires a modification to the kernel itself. While this does have the drawback of requiring an alteration to the kernel (which will almost certainly require a reboot of the device), it also has the advantage of not requiring a rooted device for the new kernel to be flashed. If, however, the kernel is modified in-place, a reboot is not required, but root permissions are required to make the necessary changes to the system files. This is the method which is initially required on most modern Android devices, as the default Gadget driver is compiled into the kernel by default.

3.2 Existing attacks

While iOS has been found to have more vulnerabilities [9], Android receives more press regarding exploitation of those vulnerabilities, as well as more malware targeted specifically against it [30]. Part of this can be traced to the difference in development environments. Where a developer's license for iOS requires payment, the tools to develop an application for Android are freely available. This, combined with the freely and simply accessible source and kernel code allow for a much deeper understanding of the internal workings of the Android operating system, which in turn enables malware authors to more easily develop malicious applications for the Android platform.

Another cause is the way third-party applications are handled. Apple and Google both verify applications submitted to their marketplaces (the iTunes App Store for iOS and the Google Play Store for Android); however, applications for iOS must bear Apple's digital signature from Apple's own signing authority before they can be installed on a normal device. Android applications go through no such centralized signing procedure, which enables unofficial application marketplaces, and which also allows a malicious application to masquerade as a legitimate application much more easily.

3.2.1 Stock devices

Attacks against stock devices, which have USB debugging disabled by default, again can be classified according to sophistication of threat. The least sophisticated, once again, takes the form of simple data theft and/or infection of ordinary data files. However, outside of a few devices which implement their own non-standard protocols and a few third-party applications, personal data such as contacts and emails are not synchronized directly between Android devices and host systems; as such, data theft of this type is limited to those files presented as user data storage. While this is still problematic if a user keeps personal or confidential documents or other files on their mobile smart device, the potential damage done over USB is significantly lessened owing to Android's preference of cloud-based storage and synchronization.

On the opposite end of the spectrum of threats against the device itself are techniques which target not Android itself, but the bootloader. Google makes available as part of its SDK the program *fastboot*, which allows interaction with the bootloader over USB. Using this program and other exploits, it is possible to gain full access to the Android operating system itself. This access allows reading and modification of system and application files, as well as installation of new system services and applications (or the replacement of existing services and applications with infected or otherwise malicious versions). Because this class of attack grants access to system and application files, far more user information is potentially at risk, including contacts, email, and even stored passwords if an application has improper security. Perhaps the most dangerous part of this type of attack is that with such low-level access, it becomes possible to install and/or execute normal C binaries (compiled for the appropriate mobile processor) which run outside the Dalvik sandbox which normal Android applications are constrained to. The few drawbacks this type of attack possesses from the point of view of the attacker are that dumping the entire system contents is a time-consuming process (a problem exacerbated by the increasing size of storage available in such devices combined with the relatively slow increases in access speeds for that memory) and the fact that some steps of the attack are obvious to anyone who observes the screen of the device while it is occurring [31].

Going in the other direction, attacks by compromised stock Android devices against a victim's host computer are also potentially a severe problem. The simplest such attack simply involves running a version of Android which communicates by enumeration as a USB mass storage device rather than using the Media Transfer Protocol; such devices are vulnerable to the same autorun exploit some iOS devices are susceptible to. Some versions of Android allow the user to switch between using MTP mode and mass storage mode as a matter of convenience, which broadens the pool of potential victim devices [32].

The more complicated attack types leverage similar or identical vulnerabilities as the bootloader attack. Once access to the system files or root access is gained, it becomes possible to alter the USB endpoints presented to a connected host device. Because the Gadget driver allows complete specification of the behavior of the driver on the side of the mobile device, it is entirely feasible to present the compromised device as something it is not, and to engage in malicious behavior after it is bound to the host with the (in)appropriate driver.

One attack would be to mimic a human-interface device such as a keyboard or mouse [31]. Another such attack would be to mimic a USB CD-ROM drive backed with an ISO image containing an autorun file which would in turn execute a malicious binary; there is already software available which can perform similar tasks with the innocuous goal of allowing the use of an Android smart device to serve as bootable media for system recovery purposes. This application depends on mass storage mode already being installed and enabled in the system kernel [33], but attacks which allow doing so are well known.

The last class of attacks against stock Android devices involves those which, for one reason or another, have USB debugging mode enabled. While under normal circumstances the Android operating system requests user confirmation of actions such as installation of new applications, it is possible to counterfeit such confirmation using the USB interface itself [31]. With that safeguard effectively neutralized, it becomes possible to both add and remove normal Android applications over USB without the victim being aware of it. This allows the installation of new malicious applications, or the replacement of legitimate applications with malicious versions. This is much easier to take advantage of on Android devices as opposed to iOS devices because there is no signing authority required for Android applications; while Android applications must be signed, a self-signed certificate may be used [34].

3.2.2 Rooted devices

While it seems unnecessary to devote a separate section to rooted devices when an Android smart device can be completely subverted with no root access necessary, a closer look at that particular attack highlights one potentially fatal drawback from the perspective of an attacker. That attack is aimed first at the bootloader and requires not only a device reboot, but potentially several minutes of uninterrupted access. While it is not unreasonable to make the assumption that a victim will believe the reboot to be an innocent malfunction on the part of the phone or that charging will take more than the several minutes the attack requires, neither is it an optimal situation. However, the bootloader exploit is meant to accomplish two things: first, to enable access to the system via a shell, similar to ADB, and second, to gain root access. Since rooting a device often requires USB debugging mode to be enabled, and any user knowledgeable enough to root their device is likely to have USB debugging mode enabled for their own purposes anyway, it is well within the realm of possibility that a rooted device will have USB debugging mode enabled, effectively granting an attacker a root shell on the Android smart device over USB.

With that understanding, a rooted device with USB debugging enabled is effectively completely under the attacker's control. Applications can be installed, removed, and replaced with malicious copies. System processes running with administrative privileges can be installed to perform such actions as sending documents, keystrokes, and user information back to a central server over covert (or possibly even overt) channels. At this point, the only two limits to what an attacker can accomplish are the attacker's imagination and the need for the attack to remain hidden from the victim, lest he or she realize the compromised status of their device and perform a factory reset.

A factory reset, however, by its very nature only affects the operating system itself; the bootloader remains untouched by system updates and restores unless there is an overriding reason to modify it, because as long as the bootloader is intact, a failed firmware update can be fixed; if the bootloader code becomes damaged, the device essentially transforms into an expensive paperweight. An attacker could conceivably modify the bootloader itself to re-infect the device on every reboot, and for these reasons a system restore would never touch the malicious code.

3.3 Potential attacks

Predicting potential attacks against the Android operating system over USB is difficult due to both the wide variety of Android devices and versions on the market, and to the wide variety of malicious activities possible once a device has been compromised. Analysis of existing technologies and attacks can however give a reasonable indication of the direction from which future attacks are likely to come.

3.3.1 Stock devices

Since a stock device by default has only MTP mode (or in some cases USB mass storage mode) enabled, attacks against such configurations are likely to continue to seek out other, more vulnerable points of ingress. Existing attacks target the bootloader, which most normal users never realize is a possibility. However, the attacks seen thus far exploit the bootloader directly to gain root access as soon as the device is connected. A potential future attack is to plant malicious code in the device's memory using the default data connection (or, more likely, on any memory card installed in the device at the time, since the MTP protocol as implemented on Android grants access to the memory card's root directory) and then to patch the bootloader so that the added code is executed the next time the device is rebooted. A user is unlikely to notice that the device reboot cycle is taking an extra few seconds, which would be all that is necessary to install a back door into the device.

3.3.2 Rooted devices

Since attacks against rooted devices are as varied as the tasks which the processor the device is equipped with can accomplish, speculation on this particular area will be restrained to the particularly novel. In this case, that comprises a class of attacks which allow modification of the kernel itself without requiring the flashing of a new kernel (a task which necessitates a device reboot). There is a tool developed for Linux systems called Ksplice, which is designed to allow kernel patches to be applied in-place, without a system reboot, in order to make regular system updated more palatable to end users [35]. Because Android is a Linux-based operating system, it is theoretically possible to port Ksplice to Android.

Ksplice applies patches in a process which consists of two primary tasks. The first analyzes the patch to be applied in the form of source code and compares it to the existing kernel, then compiles the patch and appends it to the appropriate area of the kernel. The second tool temporarily freezes execution of all programs other than Ksplice itself, goes to the areas of compiled kernel code the patch deals with, and replaces the patched code with a long jump to the patch code. It also ensures the return at the end of the patch code will jump back to the appropriate assembly instruction in the original kernel code; if the patch code does not possess a return statement, the long jump is simply inserted at the end of the new code.

The development of the first step is primarily to allow ease of automation across a number of kernel patches and versions. While the Android operating system does indeed possess a large number of kernel versions, an attacker intent on employing Ksplice or a similar kernel modification tool will only be concerned with a small number of areas of kernel code. Because the Android kernel is open source, and many device manufacturers make the code for the modified kernel which runs on their devices publicly available, manual analysis of the necessary code to be inserted is feasible, especially if an attacker chooses to pursue a small number of potential target devices.

Creation and testing of a Ksplice-like tool and the patches it generates and applies is made far simpler by the existence of two other tools. The first is the set of cross-compilers made available by Google for the express purpose of allowing developers to create native binary applications and services that run outside of the Dalvik sandbox normal Android applications are constrained to. These cross-compilers enable development on a normal Linux workstation. The second tool is a set of freely available virtual machine images of several different versions on the Android operating system [36]. By using a virtual machine of an Android device, it is possible to take a snapshot of the current state of the system, test a kernel modification, and revert to the snapshot in the event the modification renders the kernel inoperable. This allows the testing of in-place kernel patching neither the risk of permanently damaging a physical device nor the tedium of restoring a kernel image onto a malfunctioning device.

CHAPTER 4

ATTACKS AGAINST WINDOWS MOBILE DEVICES

This section will address the USB characteristics of both Windows Phone 8 (Microsoft's latest smartphone operating system) and Windows 8 RT (Microsoft's latest tablet operating system, designed for use on mobile processors such as the Tegra). The decision to focus on Windows mobile operating systems rather than the more established BlackBerry family of smart devices was made due to the rapid growth of Microsoft's platforms in the market [37], as well as the lack of a BlackBerry-based tablet computer. While Windows Phone 8 may not command the same market share as iOS or even Android, it is making strong inroads at a rapid pace, and even Apple has been forced to admit that certain Windows Phone devices surpass their own offerings in popularity at times [38].

4.1 USB interface

While Windows Phone 8 and Windows 8 RT are different operating systems, both are optimized for use on a mobile device, and those mobile devices possess USB connectivity. Therefore, both operating systems are potentially vulnerable to a compromised USB charging station. Since they are distinct platforms, this thesis will investigate each separately.

4.1.1 Windows Phone 8

The USB interface for Windows Phone 8, like modern versions of the Android operating system, enables transfer of data between the device and a host machine using the Media Transfer Protocol. Unlike Android, however, there is no USB debugging option available for this operating system; the MTP interface is the only USB interface the device provides.

4.1.2 Windows 8 RT

As anemic as the USB options on Windows Phone 8 are, the USB connectivity for Windows 8 RT are fewer still. Windows 8 RT does not function in USB device mode at all; the USB connectivity on this platform is limited strictly to serving as a USB host. In many ways, Windows 8 RT is a desktop operating system with a user interface optimized for use with touchscreen devices such as tablets, rather than an operating system designed specifically for use on mobile smart devices. This does not, however, render Windows 8 RT immune from USB-based attacks; it merely means that an attacker must present their USB connection as a subordinate device rather than as a host system.

4.2 Existing attacks

For a variety of reasons, including but not limited to the relative newness of the operating systems; the lack of sophisticated USB connectivity; and the closed-source nature of the operating systems, USB attacks against Windows mobile devices have yet to come to light. This does not mean that none are possible, or that none have occurred, however. Using current knowledge, with little or no additional effort or research, certain existing attacks can still be leveraged against these devices.

4.2.1 Windows Phone 8

As Windows Phone 8 supports the Media Transfer Protocol for transfer of data between a host and the mobile device, any of the previously mentioned attacks involving MTP are valid here as well. Attacks which leverage the mobile device as a malware delivery vector by infecting data files on the device itself are more likely to be successful, however, as Windows Phone 8 is designed to integrate well with Windows desktop operating systems, which are by far the most common target for malware. Because the only USB endpoint Windows Phone 8 reveals to a host is the MTP data transfer interface, no other novel attacks are feasible 'off-the-shelf'.

4.2.2 Windows 8 RT

While the other sections in this thesis refer to attacks from a malicious USB host to a potentially vulnerable USB device, this is not an option against Windows 8 RT. However, those sections dealing with infecting a mobile device as a vector to then infect a host system are relevant here as well, since due to the nature of Windows 8 RT, a victim who connects a Windows 8 RT tablet to a compromised USB outlet has essentially brought their host system to the attacker, cutting out the middleman of the infected mobile device altogether. There is a fair amount of work that has already been done with regards to compromising a host system with a malicious USB device, much of which is relevant to this platform.

Windows 8 RT is effectively Windows 8 designed to work on mobile CPU architectures. As such, the device drivers must be compiled specifically to work on the different architecture, which means that existing USB attacks will require some alterations. However, while the specifics of any given exploit may change, the high-level procedure of masquerading as a legitimate device and then exploiting known security vulnerabilities in the driver for that device remain the same. It is already known that various attacks such as heap overflows are possible given the correct USB device and corresponding driver [39]. The Gadget driver present in Android which allows the device to emulate any type of USB device can be implemented on a Linux-based USB controller posing as a USB charging station, and as such a malicious station can take advantage of any USB device driver vulnerability discovered. This class of attacks is problematic to develop for Windows 8 RT for one simple reason: because Windows 8 RT is compiled for a different architecture, normal Windows binaries will not run on it. This means that debugging of exploit code must be done using a remote connection to a debug service. While this is far from insurmountable, it does make an attacker's task more difficult.

4.3 Potential attacks

Because Windows Phone 8 possesses only a Media Transfer Protocol interface and Windows 8 RT has no USB interface wherein the mobile device serves as a device rather than as a host, any potential attacks will likely be the result of one or more of three possibilities: first, that there is an unknown vulnerability in the existing protocols or connections that will be discovered; second, that a future update will introduce a new communication channel which will be vulnerable; or third, that a future update will introduce a vulnerability in the existing communications channels. The third possibility would seem to be most likely, given Microsoft's history of vulnerability patching.

4.3.1 Windows Phone 8

Potential future attacks against Windows Phone 8 are likely to be the result of an update to the functionality of the USB connection on the device. Both iOS and Android allow for USB-enabled debugging of applications, which is a feature that makes development of platform-specific applications much easier. The relative newness of the Windows Phone 8 platform means that the application store for this operating system has fewer programs available than its competitors, and allowing USB debugging would be one way to encourage developers to close the gap. However, as has been pointed out elsewhere in this thesis, USB debugging mode brings with it a host of potential vulnerabilities, up to and including vulnerabilities severe enough to allow the mobile operating system itself to become compromised.

Another possibility would be the introduction of USB-based firmware recovery. As it stands now, a hardware factory reset can be performed on a Windows Phone 8 device with nothing more than a power source to ensure the recovery process is not interrupted by loss of power [40]. If USB flashing of the operating system is ever enabled, it will present a viable avenue of attack which has the potential for total system compromise.

4.3.2 Windows 8 RT

While Windows 8 RT is no more vulnerable to USB-based attacks than a normal installation of Windows 8 on a normal host system, indirect attacks against a host system are still possible. It is conceivable for a compromised USB charging station to pose as a USB device and exploit a vulnerability in a USB device driver on a Windows 8 RT tablet. That exploitation could in turn be used to install malware which can then in turn infect other systems.

The precise mechanism behind that infection could be through wireless communication with a home or workplace network, or by infecting USB memory drives used to transfer files between the tablet and a host system.

Of course, the same possibility for future attacks through expanded USB capabilities possessed by Windows Phone 8 also applies to Windows 8 RT. Should Microsoft release an update which adds USB device-mode functionality to Windows 8 RT, associated vulnerabilities will almost certainly follow. Even something as simple as adding an MTP interface would then make the device vulnerable to the same types of data theft and malware infection attacks as other popular smart device operating systems.

CHAPTER 5

COMPARISON

All mobile smart devices have certain commonalities, which are forced upon them by the purposes they must fulfill. One of those commonalities is the ability to copy data both to and from them; this ability, which is necessary for these devices to function as personal communications and media platforms, also forms the crux of their vulnerabilities to malicious USB connectivity. Additional connectivity, no matter how benign or beneficial the intended usage, also brings with it additional risk regardless of the specifics of the device in question.

5.1 Attacks common to all platforms

The one form of USB connectivity common to all three investigated platforms (counting Windows Phone 8 but not the tablet counterpart Windows 8 RT) is access to user documents, photographs, music, and videos. Therefore, any attack designed to take advantage of this functionality has the potential to be successful against any herein studied smartphone, regardless of the specific operating system that device is running.

5.1.1 Data theft attacks

Data theft-type attack risks are serious across all platforms. This is because all platforms offer relatively easy access to user media with no precautions taken to ensure the identity of the connecting host device, or indeed whether the USB connector is attached to a host device at all rather than the expected innocuous charging station. However, between the three investigated platforms, iOS has the dubious honor of boasting the highest risk, owing to the ability to synchronize not just personal media and documents but emails, contacts, and internet bookmarks as well; all of these become potential targets for theft along with the standard documents and media which are equally at risk regardless of device type.

5.1.2 Phone to computer attacks

Where the ability to read user data enables data theft attacks and a potential breach of privacy, the ability to both read and write data creates the possibility of significantly greater mischief. The addition of malicious files, or the infection of existing benign files, engenders the possibility of using an infected smart device to spread a worm or trojan to host systems, either at home or at a workplace. Because this attack can originate from any malicious USB connection, a possible (and disturbingly plausible) scenario is as follows: a teenager on vacation with her family takes several pictures with her phone's camera. At the airport on the way back, she plugs her phone into a charging station which has been compromised by an attacker; the microcomputer hidden in the charging station makes a copy of all her data for later mining, and infects several with a novel worm designed to bypass antivirus software. She returns home, and connects her phone to her personal computer; she then views the photographs on her phone, infecting her computer with a worm. The worm spreads over her family's local area network, infecting all computers it encounters. Her father, a government contractor, plugs his phone in to his computer to synchronize and charge it; the worm on his personal computer acts as the compromised USB charging station did, and infects files on his phone with the same worm, this time targeting a presentation he is to give to his manager. When he accesses his presentation at work, the worm on his phone infects the entire local area network at his workplace, compromising government contracts and secrets.

While this is a worst-case scenario and makes several optimistic assumptions (from the perspective of the attacker), it is nevertheless possible, and only a single example of how blind trust of USB outlets can have disastrous security implications. Of particular interest is that, by infecting a trusted home network, any defenses aimed at preventing infection from covert systems such as compromised USB chargers are completely circumvented by turning trusted systems into assets of the attacker. While this scenario, and others like it, are possible on any of the three platforms, the greatest risk is to Android and iOS devices, each for different reasons. Android devices are at greater risk because of the possibility of reconfiguring the device itself to bypass protections through subversion of the operating system and/or bootloader; iOS devices are at greater risk because of the greater variety of user data available to manipulate. A possible attack iOS enables is manipulation of internet bookmarks to redirect to malicious websites; they need not even directly attack the user's computer, but enable cross-site scripting or cross-site request forgery attacks for further data theft.

5.2 Attacks common to Android and iOS

Windows Phone 8 has only the MTP data connection for a USB interface; as such, the already discussed attacks are the only attacks that platform is currently known to be vulnerable to. Android and iOS, however, have expanded USB functionality, with attendant vulnerabilities. One such vulnerability is an extension of the data theft attacks all devices are vulnerable to. Where Windows Phone 8 and the default PTP/MTP interfaces on iOS and Android offer access only to the user's data, the additional connectivity present in iOS and Android allows for a complete imaging of the entire device, including the system files. This allows such data as hidden files and encrypted stored passwords to be compromised in addition to the user's personal files.

The most severe vulnerability they have in common, however, stems from the ability to install new applications over USB; additionally, this vulnerability is similarly exacerbated in both types of device by the ability to escape the restrictions manufacturers place on what can and cannot be done with the device. Rooting and jailbreaking add a new aspect to an already serious vulnerability.

Because data can be read from these devices and programs can be installed to them, even on unmodified stock systems it becomes possible to replace a legitimate application with a malicious one which can have behaviors ranging from the annoying (such as inserting advertisements) to the intrusive (such as stealing personal information and redirecting websites) to the destructive (such as deleting personal files, over-using limited resources such as cellular data, or sending false messages from accounts on the device).

While certain safeguards must be overcome to install applications on unmodified devices, techniques already exist (and have been discussed elsewhere in this thesis) for circumventing those safeguards. Furthermore, it is far from uncommon for these devices to be modified to allow a greater degree of control over the system substrate for the user; these modifications also allow a greater degree of control for an attacker. A jailbroken iOS device allows Apple's application signing requirements to be completely bypassed, while USB debugging mode on an Android device allows the arbitrary installation of applications without the need for an additional exploit. Without a doubt, though, iOS is at greater risk for this type of attack simply because unmodified devices allow applications to be installed with no additional exploits required, provided the attacker is in the possession of an Apple developer's license.

5.3 Attacks unique to a single platform

While Android and iOS both offer similar functionality in terms of user experience, they arrive at the same location via very different routes. This difference manifests in many ways, but of particular interest here is the difference in USB connectivity and what implications that difference has for device security. Attacks unique to each family of devices are enabled by the uniqueness of their USB interfaces.

5.3.1 iOS

The unique USB interface possessed by iOS is the proprietary Apple File Connection link, which allows synchronization of a wide variety of personal information and access to the complete device filesystem. This access is not, in and of itself, completely unique, as a similar level of access can be gained to Android devices; however, unlike Android, this vulnerability exists on completely unmodified, out-of-the-box devices with no actual exploits necessary. Connection of any iOS device to a compromised USB port can result in unauthorized access to email accounts, internet bookmarks, and any data stored by applications on the device such as stored encrypted passwords for internet browsers or social media services.

5.3.2 Android

While both Android and iOS allow USB interaction with their respective system bootloaders, iOS frequently patches and updates their bootloader code in an effort to defeat jailbreaking. As such, bootloader attacks are far more likely (and likely to be successful) against Android devices. Indeed, one such exploit has already been published [31]. Additionally, the open-source nature of the Android kernel (and the extensive Android modification community) makes it uniquely vulnerable to attacks which involve partial or complete reconfiguration of the kernel; these attacks are made even easier by the fact that the Android kernel is based on the Linux kernel, which means skills at working with the Linux kernel and Linux kernel coding resources are applicable, reducing the necessary learning curve for any would-be attacker.

CHAPTER 6

MITIGATION

While there are a sizable number of attacks against all modern mobile smart devices, there are ways to reduce the risk of using untrusted USB charging outlets. There are three primary categories of these mitigation strategies: methods implemented in software, methods implemented in hardware, and modification of user behavior. The first category is perhaps the best from a deployment standpoint, as developers can implement and push out these changes as a normal update without requiring any additional effort or expenditures from the end users. The second category often results in the greatest security when the attacker is limited strictly to dealing in software; however, it also causes users to incur the expense of the additional or new hardware and so is difficult to get adopted. While the last category is traditionally one of the most difficult security strategies to implement, it is becoming increasingly necessary in the modern world. Still, any or all of these options can present effective mitigation possibilities, and so this thesis will explore all of them.

6.1 Software-based mitigation techniques

Many software-based mitigation techniques exist for other types of threats, such as malware being installed through application marketplaces; some of these techniques can be extended or modified for use as protection against malicious USB connections. Different types of USB connectivity, however, can require alternate security solutions.

The connectivity all smartphones investigated in this thesis share, however, can benefit from the same solutions. Each solution addresses a specific problem or type of threat. The first threat common to all devices is that of data theft. Use of the MTP or PTP communication channels by covert

USB hosts to steal personal information can be solved simply by prompting the user to verify that they wish to connect to the host; if the USB outlet the device is connected to is not meant to have connectivity, the user will immediately know something is wrong. This feature is already present in iOS 7 [10]. However, concomitant with this approach is the need to ensure that user acceptance cannot be counterfeited, as is currently possible with Android [31]; otherwise this protection does nothing more than engender a false sense of security. Another concern is those devices which have been jailbroken, rooted, or otherwise altered to allow modification of system files and settings; it would be possible for an attacker to use this access to remove the prompt altogether if done quickly enough (or if the user is not paying sufficient attention).

The second threat related to this connectivity is that of infection of personal files for use as an infection vector against other systems. One possible solution to this would be to take advantage of the increasing processing power and memory of smart devices and run an anti-virus scan on every user file on the device in real-time as they are modified, similar to on-access scanning present in anti-virus software for personal computers. Android already supports a wide variety of anti-virus applications in the official Google Play store [41]. These applications are meant to guard against installation of malware, but could easily have their scope enhanced to include user data files as well. This is not a perfect solution, however, as many mobile anti-virus applications fail to recognize some known threats [42], no anti-virus software is proof against unknown threats, and any anti-virus software will impact the performance of either the device itself or the device's battery life, particularly if running an on-access service.

Another safeguard against this type of attack is to ensure the integrity of the user's data. Requiring user confirmation of any file transfer operation, either to or from the device, if instituted and used correctly, would prevent unauthorized modification of personal files. However, this is not a user-friendly approach, particularly if the user performs data transfers frequently. It is likely that if this feature was instituted and made optional, many users would disable it; if the feature was not made optional, those platforms such as iOS and Android which can be made to support modification of the operating system will likely see modifications aimed specifically at disabling this feature. This brings up another problem: where a prompt to accept a connection to a USB host would appear almost immediately upon connection to a compromised USB charger, requiring any circumvention to be very prompt, malicious modification of user data is initiated at the behest of the attacker, and so can be delayed until the user is more likely to not be paying attention; this would allow more time for circumvention techniques to be applied, making this method less effective.

6.1.1 Techniques for iOS

The specific USB interface unique to iOS devices is the Apple File Communication link, which is a proprietary communications protocol that requires a specific driver implementation. Because of this, it is well within the bounds of possibility to alter the interaction between device and host in such a way as to protect against security threats. Perhaps the best way to do this would be to pair an iOS device with a host system, similar to the way Bluetooth devices are paired; after pairing, the device can only communicate with that specific host, and the pairing must be canceled before the device can be synchronized with any other host. If this is enabled with a cryptographic signature, perhaps tied to the device owner's Apple ID, it becomes infeasible for an attacker with space-limited hardware (as in the case of a compromised USB charging station) to circumvent this protection, assuming the attacker is unable to employ the USB connection to cancel the pairing.

As this option would restrict the communication possibilities of an iOS device to a single host system, and a user may very well desire to synchronize such a device with multiple systems, an alternative would be to require device pairing with a certain maximum number of host systems. This number could be as low as two (for a personal desktop and a laptop for traveling), or as high as five, the maximum number of iTunes installations which an Apple ID may be associated with. Allowing multiple pairings, however, introduces the threat of a compromised USB outlet initiating a pairing with a device which has not yet reached its maximum number of partners; therefore, if this option is pursued, it becomes necessary to design the pairing procedure in such a way as to absolutely require user knowledge and interaction for a pairing to be successful.

Another vulnerability is the use of false or fraudulent information to acquire an Apple provisioning profile to be used to install unsigned malicious applications. While pairing will stop this attack by the simple expedient of not allowing the installation of any applications from an untrusted host, there is a way to prevent this particular attack in software without resorting to that specific strategy. The provisioning profile attack works because it assigns the debugging permission to the application, and registers the device as being used for debugging without any actual interaction with the device aside from identification [10]; prevention, then, can be accomplished by adopting Android's policy of enabling debugging mode on the device in addition to the existing protocols. If a user must manually enable debugging mode on their device, then it does not matter if that device is covertly registered for debugging.

6.1.2 Techniques for Android

The threats to Android systems from a compromised USB device are manifold and severe. One vulnerability, exploiting a feature intended only for developers and device recovery, allows the subversion of the entire operating system [31]; another allows full control of the device with no additional exploits necessary. Each of these vulnerabilities must be addressed, although software updates alone may not be sufficient in this case. This is because almost any successful exploit against an Android device carries the possibility of granting the attacker complete control over the operating system, enabling the bypassing of software-based security measures.

The first threat which requires addressing is the bootloader. The ability to trigger the bootloader via USB enables the complete compromise of a stock Android device [31]; therefore, it becomes of paramount importance that this vulnerability be patched. The most obvious solution would be to require a combination of hardware buttons to be pressed in order to access the bootloader, removing the ability for USB to trigger it. While this will make legitimate development on the Android platform slightly more difficult, the security dividends it pays will more than make up for it. This change would ideally be made to the bootloader code itself, enabling this mode to be entered even in the event the primary device firmware is corrupted or damaged.

The second threat which requires addressing is that of USB debugging mode, which enables user-level access to the entire device sufficient to install applications. Perhaps the best compromise between ease of use and security is to keep the existing policy which requires user interaction to enable USB debugging mode, but have that mode automatically deactivate when the USB cable is disconnected. By requiring USB debugging mode to be enabled specifically when debugging operations are desired, and automatically disabling it afterward, devices will not be left in debugging mode when connected to potentially compromised USB outlets. While this may be a nuisance to any developer who needs to frequently plug and unplug their device for development purposes, again, the benefits from a security standpoint far outweigh the minor inconvenience of a small portion of the user base.

6.2 Hardware-based mitigation techniques

The main drawback to any software-based defense against these attacks is that a successful attack has the potential to allow a level of access which enables the attacker to bypass the safeguards. Consequently, while it may be more inconvenient both from a usability and a cost standpoint, hardware based threat mitigation has the potential to be significantly more efficacious than techniques based in software. There are two primary methods for hardware-based protections: those aimed at the devices themselves, and those targeted at the USB connection itself. While both methods can benefit all devices, those methods which require modification of a smart device would effectively necessitate a user to purchase a new device in order to gain the security benefits; as these smart devices cost hundreds of dollars, this thesis will focus on the less-expensive (and thus more likely to be adopted) technique of safeguarding the USB connection itself.

The main problem with using USB as a universal charging standard stems from the fact that a USB cable comes with both data pins and power pins. While it is possible to simply disable the power pins, this is a suboptimal option as for two reasons. First, it requires a cable or adapter with the sole purpose of enabling charging while disabling communication; this will be a problem whatever the safeguard and is more of an inconvenience than anything else. Second, and more problematic, is that the data pins can also be used to negotiate the power transfer parameters between the device and the charger; simply disabling the data pins could result in substandard charging performance, and may, depending on the charger, actually result in damage to the device if an attempt is made to supply more power than the device was designed to handle. However, it is possible to create a circuit which enables power negotiation but disables all other data transfer; indeed, such a cable is already being sold [43].

6.3 User prevention techniques

The final class of threat mitigation techniques is simply user awareness. It has long been known that the greatest threat to any security system is the users who must interact with and enforce it; this is one of the reasons social engineering attacks are still prevalent. Educating users in what to do (and more importantly, what not to do) when it comes to using USB connections can be an effective threat management strategy, but only if users can be convinced to take the appropriate precautions.

The first precaution to be taken is, quite simply, not to trust public USB connections. Anywhere there is a USB charging station, there are likely to be standard power outlets, and standalone AC-to-USB power adapters are readily available, inexpensive, and in some cases actually come bundled with a smart device. Use of these adapters (provided the adapter comes from a trusted source; manufacturers of USB picture frames have been known to ship infected products, so it is a possibility that someone may ship infectious chargers as well) bypasses any modifications an attacker may have made to a USB charging station completely.

Other precautions are to not engage in risky behavior with a smart device. Users who refrain from jailbreaking, rooting, and/or enabling USB debugging modes are at less risk than those who indulge in such modifications. Users who do not store personal or private information on their smart device are not at risk of having that information stolen off of that device. Of course, these solutions are problematic as many users would consider this to be defeating the entire point of owning a smart device in the first place.

Finally, if a user must use a public USB charging station, they should be aware of the normal behavior of their device, and be aware that any deviation from that behavior may indicate an attack. Specifically, a device should not reboot when connected to a power source, and a simple charger should not register as a connected host system. Any sign that the device is synchronizing should be assumed to mean that the device is in communication with a host system, and if no such communication is intended, then something is wrong.

CHAPTER 7

CONCLUSIONS

The prevalence of smart devices in society is only going to increase. The USB standard has risen to meet the demand of greater bandwidth, progressing from standard 1.1 to 2.0 and most recently to 3.0 with theoretical file transfer rates most modern hard disks would be hard-pressed to keep up with. These two facts in tandem suggest that USB as a charging standard is here to stay. Accordingly, it is becoming more important than ever that USB outlets are not blindly trusted. Malware already exists that spreads via USB [18]; therefore, assuming the trustworthiness of any arbitrary USB outlet is foolish in the extreme.

The knowledge already exists to subvert a USB charging station with a microcomputer, and the knowledge already exists to launch a number of dangerous and potentially crippling attacks against smart devices. Furthermore, the knowledge already exists to leverage those smart devices as a further infection vector against otherwise adequately protected personal computers and workstations. These existing attacks underscore the importance of securing devices against hostile or malicious USB connections, which subvert interfaces meant to be beneficial to the user.

Worse than the existing attacks are the potential attacks which can be created with sufficient time, effort, and resources. Relatively little work has been done thus far on USB vulnerabilities in general and smart device-based USB vulnerabilities in particular, but what work has been done presents an alarming picture. Wherever there is the ability to transfer data to and from a device, there is at least the possibility of turning that channel to nefarious purposes. Even those devices such as Windows Phone 8 which restrict their USB communication to a sandboxed user data area are not immune.

Of the devices investigated by this thesis, the most vulnerable type is Android. An attack already exists which grants total control of an Android device if it is connected to a compromised USB port, and the open source nature of the operating system (along with its Linux roots) make finding vulnerabilities and creating malware targeted at this platform far easier than for any other.

The next most vulnerable mobile operating system is iOS. While total subversion of the firmware requires either a jailbroken device or for the attacker to effect a jailbreak upon connection, the default USB connectivity grants sufficient access to the device filesystem to permit a wide range of malicious activity. The ability to install applications over USB also creates potential problems in light of Apple's policies regarding developer licenses.

The least vulnerable of the smart phone operating systems is Windows Phone 8. This is due largely to the fact that Windows Phone 8 has elected to simply not use any but the most basic of USB connectivity. Windows Phone 8 is also a comparatively new operating system, and accordingly less research has been done into any vulnerabilities it may possess. Nevertheless, from a USB security standpoint it is currently the most secure option of those investigated herein.

A special mention goes to Windows 8 RT. While it is not in use on smartphones, it is in use on tablet computers, and is the safest of all from the threat of connecting to a covert USB host. However, Windows 8 RT is still a Windows operating system, and receives security updates via the same mechanism and with approximately the same frequency as its personal computer counterpart, Windows 8. This means that an attack aimed specifically at Windows 8 RT, using a microcontroller meant to emulate the behavior of a USB device rather than a host, has a high probability of being successful.

While the attacks presented and postulated in this paper represent symptoms of the problem, the problem itself is complacency; USB outlets are trusted far beyond what they deserve, especially those in a public setting which anyone may access. The mitigation techniques presented in this thesis are aimed at reducing that trust, both from smart devices and from the end users themselves. Given that software-based security mechanisms have always been — and likely always will be — an arms race between attackers and defenders; hardware-based mechanisms are intrusive, expensive, or both; and users can always be counted upon to misinterpret, ignore, or forget basic security precautions, defending against these threats will require constant vigilance.

REFERENCES

[1] J. Brenner, “Pew Internet: Mobile,” 2013, archived: http://www.webcitation.org/6L5YMVJlz. [Online]. Available: http://pewinternet.org/Commentary/2012/February/Pew-Internet-Mobile.aspx

[2] ChargeAll Tutorials, “Cell Phone Charging Station Guide,” 2013, archived: http://www.webcitation.org/6L5ZQWCT5. [Online]. Available: http://chargeall.com/cell-phone-charging-station-guide/

[3] Raspberry Pi, “FAQs — Raspberry Pi,” 2013, archived: http://www.webcitation.org/6L5bHPZEh. [Online]. Available: http://www.raspberrypi.org/faqs

[4] Moosy Research, “iPhone USB Tethering in Linux,” 2013, archived: http://www.webcitation.org/6KrUHQs0b. [Online]. Available: https://sites.google.com/site/moosyresearch/projects/iphone-usb-tethering-in-linux

[5] Wikipedia, “Picture Transfer Protocol,” 2013, archived: http://www.webcitation.org/6KrVm7OtC. [Online]. Available: http://en.wikipedia.org/wiki/Picture_Transfer_Protocol

[6] L. Gomez-Miralles and J. Arnedo-Moreno, “Universal, fast method for ipad forensics imaging via usb adapter,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 2011, pp. 200–207.

[7] D. Giagio, “ipheth/ipheth-driver/ipheth.c at master dgiagio/ipheth GitHub,” 2013, archived: http://www.webcitation.org/6Krc6sSsF. [Online]. Available: https://github.com/dgiagio/ipheth/blob/master/ipheth-driver/ipheth.c

[8] iPhone Wiki, “AFC - The iPhone Wiki,” 2012, archived: http://www.webcitation.org/6KrnKK1Yz. [Online]. Available: http://theiphonewiki.com/wiki/AFC

[9] B. Reid, "iPhone Found To Be More Vulnerable Than Windows Phone, BlackBerry And Android Smartphones, According To New Report — Redmond Pie," 2013, archived: http://www.webcitation.org/6Kv6PS8rG. [Online]. Available: http://www.redmondpie.com/iphone-found-to-be-more-vulnerable-than-windows-phone-blackberry-and-android-smartphones-according-to-new-report/

[10] P. Bright, "Trusting iPhones plugged into bogus chargers get a dose of malware — Ars Technica," 2013, archived: http://www.webcitation.org/6KvAfFfAT. [Online]. Available: http://arstechnica.com/security/2013/07/trusting-iphones-plugged-into-bogus-chargers-get-a-dose-of-malware/

[11] D. Damopoulos, G. Kambourakis, and S. Gritzalis, "iSAM: An iphone stealth airborne malware," in *Future Challenges in Security and Privacy for Academia and Industry*, ser. IFIP Advances in Information and Communication Technology, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds. Springer Berlin Heidelberg, 2011, vol. 354, pp. 17–28. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21424-0_2

[12] iPhone Wiki, "Cydia.app - The iPhone Wiki," 2013, archived: http://www.webcitation.org/6KwDlzbck. [Online]. Available: http://theiphonewiki.com/wiki/Cydia.app

[13] iPhone Development Wiki, "SSH Over USB - iPhone Development Wiki," 2013, archived: http://www.webcitation.org/6KwEAQJ42. [Online]. Available: http://iphonedevwiki.net/index.php/SSH_Over_USB

[14] S. Perez, "How to Secure Your Jailbroken iPhone ReadWrite," 2009, archived: http://www.webcitation.org/6KwEfmpDh. [Online]. Available: http://readwrite.com/2009/11/12/how_to_secure_your_jailbroken_iphone

[15] iPhone Wiki, "Ikee-virus - The iPhone Wiki," 2009, archived: http://www.webcitation.org/6KwEpRJFv. [Online]. Available: http://theiphonewiki.com/wiki/Ikee-virus

[16] G. Duncan, "Intego Claims Malware Targeting Jailbroken iPhones — Digital Trends," 2009, archived: http://www.webcitation.org/6KwEwwTDV. [Online]. Available: http://www.digitaltrends.com/mobile/intego-claims-malware-targeting-jailbroken-iphones/

[17] K. Rogers, "Jailbroken: Examining the policy and legal implications of iPhone jailbreaking," *Pittsburgh Journal of Technology Law and Policy*, vol. 13, no. 2, 2013.

[18] Trend Micro, "Stuxnet Malware Targeting SCADA Systems," 2010, archived: http://www.webcitation.org/6KwJKMeus. [Online]. Available: http://threatinfo.trendmicro.com/vinfo/web_attacks/Stuxnet Malware Targeting SCADA Systems.html

[19] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iPhone: decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 551–562.

[20] R. Templeman, Z. Rahman, D. J. Crandall, and A. Kapadia, "Placeraider: Virtual theft in physical spaces with smartphones," *CoRR*, vol. abs/1209.5982, 2012.

[21] D. Damopoulos, G. Kambourakis, and S. Gritzalis, "From keyloggers to touchloggers: Take the rough with the smooth," *Computers & Security*, vol. 32, no. 0, pp. 102 – 114, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404812001654

[22] Z. Epstein, "iOS Market Share, September 2013: Apple lost ground to Android — BGR," 2013, archived: http://www.webcitation.org/6L1BovBg7. [Online]. Available: http://bgr.com/2013/10/01/ios-market-share-september-2013/

[23] CyanogenMod Team, "CyanogenMod — Android Community Rom based on Jelly Bean," 2013, archived: http://www.webcitation.org/6L1C0X9pI. [Online]. Available: http://www.cyanogenmod.org/

[24] D. Franc, "Major Custom ROM comparison database," 2013, archived: http://www.webcitation.org/6L1CDO5TP. [Online]. Available: https://docs.google.com/spreadsheet/lv?key=0Auzhy7U8YwLodEtnZ-G9GMzdvLXI0N2RfWlV2NHNWNVE&toomany=true#gid=1

[25] Android Developers, "Android Debug Bridge — Android Developers," 2013, archived: http://www.webcitation.org/6L1CUMjdE. [Online]. Available: http://developer.android.com/tools/help/adb.html

[26] Android Developers, "USB Host and Accessory — Android Developers," 2013, archived: http://www.webcitation.org/6L1Cfv6bq. [Online]. Available: http://developer.android.com/guide/topics/connectivity/usb/index.html

[27] Google, “Android Git repositories - Git at Google,” 2013, archived: http://www.webcitation.org/6L1D34Zrx. [Online]. Available: https://android.googlesource.com/?format=HTML

[28] Stack Overflow, “USB - Mass storage replaced with MTP from Android HoneyComb - Stack Overflow,” 2013, archived: http://www.webcitation.org/6L1DH5482. [Online]. Available: http://stackoverflow.com/questions/14392022/mass-storage-replaced-with-mtp-from-android-honeycomb

[29] R. Shukla, “How to Enable Developer Options on Android Devices with Android 4.2/4.3 JB and Android 4.4 KitKat,” 2013, archived: http://www.webcitation.org/6L3cdjxZ3. [Online]. Available: http://www.droidviews.com/how-to-enable-developer-optionsusb-debugging-mode-on-devices-with-android-4-2-jelly-bean/

[30] United States Department of Homeland Security, “Threats to Mobile Devices Using the Android Operating System,” 2013, archived: http://www.webcitation.org/6L1DrnLSr. [Online]. Available: http://info.publicintelligence.net/DHS-FBI-AndroidThreats.pdf

[31] Z. Wang and A. Stavrou, “Exploiting smart-phone usb connectivity for fun and profit,” in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 357–366.

[32] Device Recovery, “How to connect Android to PC with USB Mass Storage Mode,” 2012, archived: http://www.webcitation.org/6KzOrigDN. [Online]. Available: http://www.device-recovery.com/how-to-connect-android-devices-to-pc-with-usb-mass-storage-mode

[33] B. Reid, “How To Boot Your PC Directly From Any Android Device,” 2013, archived: http://www.webcitation.org/6KzNylz5M. [Online]. Available: http://www.redmondpie.com/how-to-boot-your-pc-directly-from-any-android-device/

[34] Android Developers, “Signing Your Applications — Android Developers,” 2013, archived: http://www.webcitation.org/6KzPcAc0d. [Online]. Available: http://developer.android.com/tools/publishing/app-signing.html

[35] Wikipedia, “Ksplice,” 2013, archived: http://www.webcitation.org/6L1EO9sKZ. [Online]. Available: http://en.wikipedia.org/wiki/Ksplice

[36] D. Fages, “AndroVM blog — Running Android in a Virtual Machine,” 2013, archived: http://www.webcitation.org/6L1EsPGDC. [Online]. Available: http://androvm.org/blog/

[37] D. Sunnebo, “News - Record share for Windows phone - Kantar Worldpanel,” 2013, archived: http://www.webcitation.org/6L2IX1cco. [Online]. Available: http://www.kantarworldpanel.com/Global/News/Record-share-for-Windows-phone

[38] T. Worstall, “Apple’s Siri Says Nokia Windows Phone is Best Cell Phone Ever - Forbes,” 2012, archived: http://www.webcitation.org/6L2Ix3pBb. [Online]. Available: http://www.forbes.com/sites/timworstall/2012/05/15/apples-siri-says-nokia-windows-phone-is-best-cell-phone-ever/

[39] D. Barrall and D. Dewey, “Plug and root, the USB key to the kingdom,” *Presentation at Black Hat Briefings*, 2005.

[40] Tech Inspiration, “How to soft/hard reset Windows Phone 8 like Lumia 920, HTC 8x to factory settings,” 2013, archived: http://www.webcitation.org/6L2Po2JZH. [Online]. Available: http://www.ekoob.com/reset-to-factory-settings-windows-phone-8-soft-hard-reset-12142/

[41] L. Seltzer, “Android antivirus products compared,” 2013, archived: http://www.webcitation.org/6L5NTsCjr. [Online]. Available: http://www.zdnet.com/android-antivirus-comparison-review-malware-symantec-mcafee-kaspersky-sophos-norton-7000019189/

[42] R. Whitwam, “Android antivirus apps are useless, heres what to do instead,” 2011, archived: http://www.webcitation.org/6L5NdYRdD. [Online]. Available: http://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead

[43] S. Colaner, “Wrap that Rascal with a USB Condom,” 2013, archived: http://www.webcitation.org/6L5O9klRs. [Online]. Available: http://hothardware.com/News/Wrap-That-Rascal-With-A-USB-Condom/