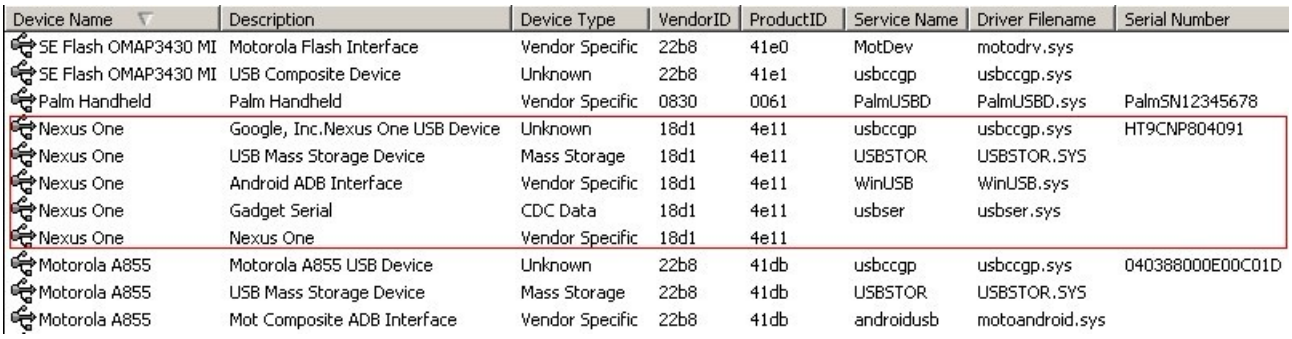
Exploiting Smart-Phone USB Connectivity For Fun And Profit

Angelos Stavrou, Zhaohui Wang
Computer Science Department
George Mason University, Fairfax, VA
{astavrou, zwange}@gmu.edu

1 Motivation & Background

Recent advances in the hardware capabilities of the mobile hand-held devices have fostered the development of open source operating systems for mobile phones. These new generation of smart phones such as iPhone and Google Android phone are powerful enough to accomplish most of the tasks that previously required a personal computer. Indeed, this newly acquired computing power gave rise to plethora of applications that attempt to leverage the new hardware. This includes Internet browsing, email, GPS navigation, messaging, and custom applications to name a few. In addition, the ubiquitous use and the wide-spread adoption of Universal Serial Bus (USB) [3] led the phone device manufacturers to equip the majority of third-generation phones with USB ports. In fact USB is currently employed as a means of charging, communicating, and synchronizing the contents of the phone with computers and other phones. Moreover, to support an open programming model that allow third party developers to contribute their applications, these new devices come with an extended set of features. These features enable them use the USB interface to perform more complex functions including data and application synchronization.

USB connections are inherently trusted and assumed secure by the users. This can be partly attributed to the physical proximity of the device and the desktop system and the fact that, in most cases, the user owns both systems. However, as we show, this trust can be easily abused by a malicious adversary. For instance, in a typical usage scenario, an unsuspected user connects the smart phone device to her computer to charge its battery and to synchronize the two devices including her contact list, calendar and media content. All of these tasks are performed automatically either completely transparently to the user or with minimal user interaction: the simple press of a mouse click upon connecting the USB cable. To make matters worse, the computer is completely unaware of the type of the device that is connected to the USB port. As we elaborate later, this observation can



Device Name	Description	Device Type	VendorID	ProductID	Service Name	Driver Filename	Serial Number
SE Flash OMAP3430 MI	Motorola Flash Interface	Vendor Specific	22b8	41e0	MotDev	motodrv.sys	
SE Flash OMAP3430 MI	USB Composite Device	Unknown	22b8	41e1	usbccgp	usbccgp.sys	
Palm Handheld	Palm Handheld	Vendor Specific	0830	0061	PalmUSBD	PalmUSBD.sys	PalmSN12345678
Nexus One	Google, Inc.Nexus One USB Device	Unknown	18d1	4e11	usbccgp	usbccgp.sys	HT9CNP804091
Nexus One	USB Mass Storage Device	Mass Storage	18d1	4e11	USBSTOR	USBSTOR.SYS	
Nexus One	Android ADB Interface	Vendor Specific	18d1	4e11	WinUSB	WinUSB.sys	
Nexus One	Gadget Serial	CDC Data	18d1	4e11	usbser	usbser.sys	
Nexus One	Nexus One	Vendor Specific	18d1	4e11			
Motorola A855	Motorola A855 USB Device	Unknown	22b8	41db	usbccgp	usbccgp.sys	040388000E00C01D
Motorola A855	USB Mass Storage Device	Mass Storage	22b8	41db	USBSTOR	USBSTOR.SYS	
Motorola A855	Mot Composite ADB Interface	Vendor Specific	22b8	41db	androidusb	motoandroid.sys	

Figure 1: The logical communication channels of the composite USB Device as they appear in Windows XP systems.

be exploited by a sophisticated adversary to launch attacks against the desktop system. Furthermore, there are no mechanisms to authenticate the validity of the device that attempts to communicate with the host. This lack of authentication allows the connecting device to disguise and report itself as another type of USB device, abusing the ubiquitous nature operating system.

Traditionally, a smart phone device is connected to the host as a peripheral USB device. Being controlled by the host, the device is more prone to be taken over by a compromised computer. However, the potential attack surface is much wider: the USB creates a bidirectional communication channel, permitting, in theory, exploits to traverse both directions. New generation phones are equipped with complete operating systems which make them as powerful as a desktop system. These recent hardware advancements enables them to perform attacks that are far beyond their previous computational and software capabilities. Additionally, unlike desktop computers and servers that do not change their physical location, phones are mobile. This empowers them to potentially communicate to an even larger number of un-infected devices across a wider range of administrative domains. For example, a smart phone left unattended for a few minutes can be completely subverted and become an point of infection to other devices and computers. Lastly, because USB-borne attacks have not been seen before, there are no defenses in place to prevent them from taking place or even detect them.

Devices	**USB interface types**
iPhone/iTouch	Apple Proprietary 5-pin wide USB
Motorola Droid and other Android based	Micro USB AB
HTC Windows CE-Based	Micro HTC ExtUSB with 11-pin connector
Old Nokia models	Pop-Port connector
Google's Nexus One	Micro USB AB

Table 1: USB interfaces of various mobile devices.

In the meantime, the lack of deployed USB defenses or detection mechanisms empowers the attacks to remain stealthy. Currently, the only instance of USB-borne threats is flash drive viruses spreading from USB files. However, the new smart phones are capable

of accomplishing a much more powerful and widespread propagation of malfease. The propagation that can be caused by this new infection vector goes beyond viruses that are passively hidden in traditional USB storage devices. The above observations motivate our study of this new infection vector that is spurred by the new technology trends, as well as propose potential defenses.

Nowadays, most smart phones are equipped with a Mini USB or Micro USB interface for PC to phone connectivity. This USB interface provides the physical link for the synchronization of contacts and calendar data. Table 1 gives the different USB interfaces with different devices.From the operating system point of view, all Android driven devices contain more than one interface descriptor, which is known as a composite USB device. This physical link can be multiplexed: with a single physical USB interface, the device can act as multiple devices simultaneously as long as they comply with the USB specification.

For our experiments, the device is Google's Nexus One. The operating system is Android 2.1 (codename *eclair*). While Google's website [2] lists the specifications from a marketing point of view, Table 2 lists the hardware modules of the device from the operating system's point of view: the second column is the internal device driver names of the different modules. Table 3 provides the MTD (Memory Technology Device) device partition layout , whereas MTD is the Linux abstraction layer between the hardware-specific device drivers and higher-level applications. How fast we can flash the device depends on the size of the storage each specific device equipped with. In addition to the NAND device storage, Google's Nexus One uses a 4GB sd card as external storage. This works as separated device in the Android operating system and can be mounted as a USB mass storage device to the desktop system. We will leverage this hardware design to launch the Phone-to-Computer attacks. In the manufacture state, the Google's Nexus One has only two logical USB interfaces by default, one is the USB mass storage while the other is the Android ADB Interface. By modifying the kernel source code with corresponding kernel compilation options, we enabled other hidden USB interfaces in the kernel, show in Figure 1.

Modules	**Hardware**
CPU	Qualcomm QSX8250 1Ghz
Mother board	Qualcomm Mobile Station Modem (MSM) SoC
RAM	512 MB
ROM	512 MB , partitioned as boot/system/userdata/cache and radio
External Storage	4GB micro SD
Audio Processor	Msm_qdsp6 onboard processor
Camera	5 MegaPixels Sensor_s5k3e2fx
Wifi+BlueTooth+FM	Boardcom BCM 4329, 802.11a/b/g/n
Touch Screen Input	Msm_ts touchscreen controller, capella
Vibrator	Msm_vibrator on board vibrator
Digital Compass	AK8973 compass

Table 2: Google's Nexus One Hardware Modules.

2 Novel Infection Vectors

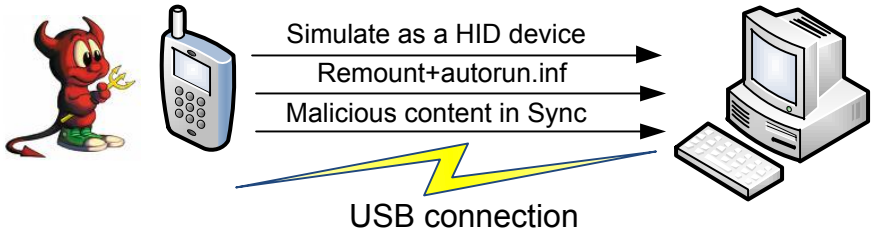


Figure 2: The Phone-to-Computer Attacks over the USB Connection.

2.1 Threat Model

To establish basic communication, the both end of the USB connection are connected via off-the-shelf USB cables. In our threat model, we assume an adversary that is already in control of one end of the USB connection. This is true for all our three attack scenarios. For instance, in the Phone-to-Computer attacking scenario, the phone is fully under the control of the adversary. Moreover, we assume that the attacker can manipulate any component of the device, ranging from applications to programmable hardware components. The victim, in this case the desktop system, is assumed to have a basic set of device drivers that come with the installation of the operating system and support Human Interface Device (HID) installation. Note that this is not an additional step required to be accomplished by the adversary. In the case of Computer-to-Phone infection, we assume the desktop system is compromised. Put it differently, we assume that the adversary has already placed malicious software that runs alongside with the regular legitimate software. The phone is considered intact and in the default manufacturer state. We only focus on how the compromised desktop system could infect the phone and propagate malware while connected through USB to the device. How the desktop system became comprised is beyond the scope of this paper. Such exploitation can be accomplished via traditional browser exploitation, email phishing, or buffer overflow.

For Phone-to-Phone attacks, the attacking device is manipulated to take over the innocent victim device. Beyond the full control of the mobile operating system of the attacking device, the adversary also has to craft a special USB cable. This cable is used to place the malicious device into USB host-mode and establish a connection to the the target phone device. We explain the necessary USB cable modifications in Section 2.4. Having established a thread model and listed our assumptions, we detail the steps to accomplish USB-borne attacks in the following sections.

2.2 Phone-to-Computer Attacks

Upon connection, USB becomes a bidirectional communication channel between the host (normally a desktop system) and the peripheral device. The established belief that only the master device (i.e the host computer) is potentially capable of taking over the slave

device (i.e. the smart phone) is incorrect. Indeed, an attacker can launch attacks and transfer malicious programs from a USB peripheral to the machine that acts as a host. Launching attacks against the connected desktop system is a new emerging avenue of exploitation that can be used to spread malware. We demonstrate this new infection vector by focusing on two general classes of attacks which have not been introduced previously.

The first class takes advantage of the fact that smart phones have open source operating systems and can pose as Human Interface Device (HID) peripherals (also called gadgets) and connect to the computer. This new functionality can be leveraged by an sophisticated adversary to cause more damage than traditional passive USB devices. The second class of attacks harnesses the capability of the phone to be automatically mounted as a USB device and automatically run content. The process of a USB device being mounted is not a threat on its own. Even having the possible malware hidden in sd card partition in the device and mounted on the computer as a USB stick is not a novel attack. However, being able to identify the operating system on the other side of the USB connection and prepare an attack payload *selectively* is a new attack capability. This is because the phone can arbitrarily control and repeat this mount and unmount operation within the device.

To demonstrate first class of attacks, we developed a special USB gadget driver in addition to existing USB composite interface on the Android Linux kernel using the *USB Gadget API for Linux*[4]. The UGAL framework helped us implement a simple USB Human Interface Driver (HID) functionality (i.e. device driver) and the glue code between the various kernel APIs. Using the code provided in:
"drivers/usb/gadget/composite.c", we created our own gadget driver as an additional composite USB interface. This driver simulates a USB keyboard device. We can also simulate a USB mouse device sending pre-programmed input command to the desktop system. Therefore, it is straightforward to pose as a normal USB mouse or keyboard device and send predefined command stealthily to simulate malicious interactive user activities. To verify this functionality, in our controlled experiments, we send keycode sequences to perform non-fatal operations and show how such a manipulated device can cause damages In particular, we simulated a Dell USB keyboard (vendorID=413C, productID=2105) sending "CTRL+ESC" key combination and "U" and "Enter" key sequence to reboot the machine. Notice that this only requires USB connection and can gain the "current user" privilege on the desktop system. With the additional local or remote exploit sent as payload, the malware can escalate the privilege and gain full access of the desktop system.

Another class of attacks are content exploitations. Such attacks take advantage of media content to exploit vulnerable softwares that exist in the victim system. These attacks are not new and have been known for quite some time (e.g. PDF and Flash exploits). However, we show a new way to accomplish these attacks using the USB connection.

(*Operating System Fingerprinting via USB*) For USB protocol, different operating system implementation will leave different footprint in the communication process, like any network protocol. With the computation power inside the gadget device, we developed operating system fingerprinting technique: in stable connectivity environment without unexpected packet loss due to signal fading, Linux (2.6.32 kernel) and MacOSX will start

the USB enumeration process by sending a get_device_descriptor control request as the ping packet. If the device replies this control request, the host controller identifies the gadget is alive and reset the bus immediately followed by the formal USB enumeration by resending a get_device_descriptor control request. All subsequent USB control requests are based on the gadget's device information. The host will continue enumerating every interface the gadget device reporting. However, for windows systems, the device state checking is more thoroughly: the ping request do not stop at get_device_descriptor only but will continue pull out all subsequent interface and configuration information. If everything works correctly, the windows host will reset the bus and start formal USB enumeration and setup the connection link.

In Android devices, in addition to the NAND device, an sd card works as external storage. This separated device can be mounted as a USB mass storage device to the desktop system. There are system-wide options for the user to set:1, connecting only for battery charging;2, allowing NAND ROM device available to the desktop system via USB Android Debugging Bridge driver (*adb*);3, allowing sd card device available to the desktop system as a USB mass-storage device. If the last option is set, the sd card device is *automatically* mounted by generic USB mass-storage driver in major commodity operating systems by default bypassing any restrictions. We leverage this platform-specific observation to implement the basic attack against the desktop system. Our malicious program drops an *autorun.inf* and the *calc.exe* to the sd card partition. The next time when the user want to transfer files (e.g. movie, photo, mp3 file etc), once the sd card is mounted as a partition, the calc.exe will be executed in our default configuration Windows XP system [1].

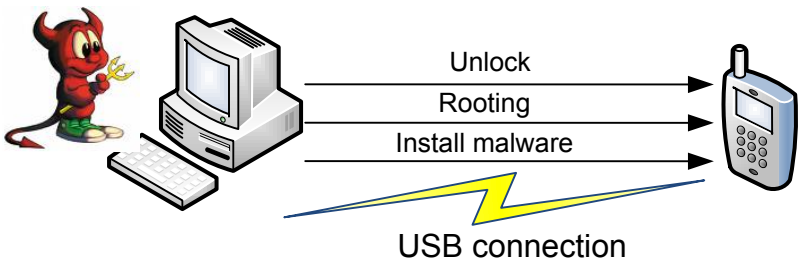


Figure 3: The Computer-to-Phone Attacks over the USB Connection.

Moreover, unlike the traditional passive USB stick devices, the CPU powered phone as a USB peripheral device promotes the attacks in a more intelligent manner. As a starting point, we (the attacker) wrote the malware on the phone monitoring the USB connectivity. Once the phone is connected to a desktop system, we probe and identify the operating system by looking at the URB (USB Requesting Block) ID in the USB packets. By doing this, we differentiate the targeted system and avoid brute force approaches. After the target system is being identified, using the computational power on the phone, we enumerate the available vulnerabilities and change the attacking payload with multiple runs with different content. For example, in our controlled experiments, the targeted desktop system is a Windows XP SP3 with a vulnerable version Adobe PDF software and fully updated JPG parse engine. Our proof-of-concept malware on the phone will compose the

autorun.inf upon detecting it is a Windows, and launch Windows Picture and Fax Viewer program to view the special crafted JPG file and the PDF program to view the malicious PDF file we dropped. We observed the expected result that the malicious logic in the crafted PDF file was executed and the Windows system is compromised. We acknowledge that this depends on malware-writer's knowledge on contemporary vulnerabilities. However, the CPU equipped phone device as a gadget can help malware-writers generate composite malware and highly infectious code, to achieve higher successful ratio.

For iPhone devices, the strong coupling between iTunes software and iPhone devices makes such Phone-to-Computer attacks even simpler. Once the iPhone connected to the desktop system, the iPhone/iPod Service installed by iTunes will detect the device and launch iTunes. iTunes will scan the media content on the device and make them available in the iTunes. Since the attacker has the full control of the device, it can drop any specially crafted media file (e.g. jpg, pdf, mp3, mov etc) to exploit the corresponding processing engine.

2.3 Computer-to-Phone Attacks

In this section, we detail the steps required to take over a smart phone device when its connected via the USB port to a computer. A closer look into the attacking process reveals that it can be decomposed into a sequence of operations. The phone is not unlocked and in manufacture out-of-box state in terms of installed software. This is usually true for most of the end-users. To mount the attack, we take advantage of the open source program *fastboot* which can manipulate the boot-loader of the Android phone devices. By issuing the command *fastboot oem unlock*, the device will display a warning page and once we click "yes", it is officially unlocked and the manufacture warranty also is voided. However, this is far from being inconspicuous and requires user input. To achieve fully automation, we crafted a small program to simulate the clicking of yes action. We do so by sending the touchscreen input event with the corresponding touchscreen coordinators need be pressed directly via the USB connection. Upon completion of the unlocking process, we can replace the system images. This means that all software including kernel, libraries, utility binaries, and applications are now under our control. The second step is to do a full system dump from device, so that we can ex-filtrate all the programs and user information. This can be used for phishing purposes in addition to creating a backup of the applications to prevent the user from noticing any changes in the device.

Dev	**Size**	**Name**	**Range**	**Erasesize**
mtd0:	0x000e0000 896KB	misc	0x000003ee0000-0x000003fc0000	0x00020000
mtd1:	0x00500000 5MB	recovery	0x000004240000-0x000004740000	0x00020000
mtd2:	0x00280000 2.5MB	boot	0x000004740000-0x0000049c0000	0x00020000
mtd3:	0x09100000 145MB	system	0x0000049c0000-0x00000dac0000	0x00020000
mtd4:	0x05f00000 95MB	cache	0x00000dac0000-0x0000139c0000	0x00020000
mtd5:	0x0c440000 196.24MB	userdata	0x0000139c0000-0x00001fe00000	0x00020000

Table 3: Google's Nexus One NAND Partition Layout.

The entire unlocking and flashing process takes 4 mins 5 seconds on our device and may vary for different devices due to different content sizes. To be more specific, we flash the recovery partition using a third party modified recovery image which provide the functionality that can do a whole NAND file system backup based on the partition information in Table 3. Such backup covers boot partition, system partition, userdata partition, and a hash checksum. We disassemble this boot partition dump *boot.img* to a raw kernel zimage binary file and corresponding ram-disk file. The boot.img file is composed with the kernel in zimage format, the compressed ram-disk in gzip format, and the paddings. The overall layout of the boot.img file is listed as follows: 0x0-0x7ff: File Magic:"Android!",kernel size in bytes, kernel physical loading address, ram-disk size in bytes, ram-disk physical loading address, product name, kernel command line options (512bytes), timestamp, sha1 hash. 0x800:4K page aligned kernel zimage with zero trailing paddings after that is the ram-disk which also 4K page aligned and zero padded. The last part is a second optional kernel for testing and do not normally appear in device. We use such knowledge to repack the boot.img file which includes malicious code.

Google maintains regular release and updates for Android system, and all the boot.img files are publicly available as well as other system files. The user may update the boot.img on it's own and we can not assume it has the same boot.img as Google's released standard ones. For a particular victim device, we do not have the prior knowledge about this boundary information between the kernel and the ram-disk. Since the magic string of gzip file is 0x1F8B, we use 0x000000001F8B which is the trailing padding zeroes plus the gzip magic string as the identification of the start ram-disk content, and rewrite them to separate files. After we get the ram-disk file, we unpack it and get direct access to *init.rc* file. This file is parsed by *init* program which is also the first process of the system. It sets up the basic environment for the system and then launches critical system daemon processes and services. The *init* binary and *init.rc* include Android specific system features (e.g some global system properties are defined and parsed here) and are critical to the entire system. Until now, we assumed direct access to all the resources to insert our malicious logic into the system. Initially, we bind the *adbd* daemon process with root permission by changing the *adbd* parameters *init.rc* file. This will provide root shell access to the whole system when we launch *adb* connection from our desktop system as a attack vector. Afterwards, we use the command in *init.rc* to remount system partition as read-only or we can run"(mount yaffs2 mtd@system /system ro remount,mount rootfs rootfs / ro remount)" to achive full filesystem privileges regardless of the system settings. Then, we add new command in *init.rc* file to launch the malicious program as a system service which will be pushed into the system as a separate step so that it is persistent and still running after phone reboot or battery outage. It is worth mentioning that this makes the malicious program persistent at bootup and is agnostic to the malware code itself. If the malicious binary is removed, such automated initialization will fail. The path need to match the corresponding path of the binary.

After performing the aforementioned modifications, we repack the *boot.img* from the modified sources and flash it back to boot partition on the device. The repack process is straightforward: we compress the modified ram-disk files and directory structures into a single *ramdisk.cpio.gz* file. We then combine it with the kernel and kernel command line options by *mkbootimg* program which is available in Android repository. The flashing pro-

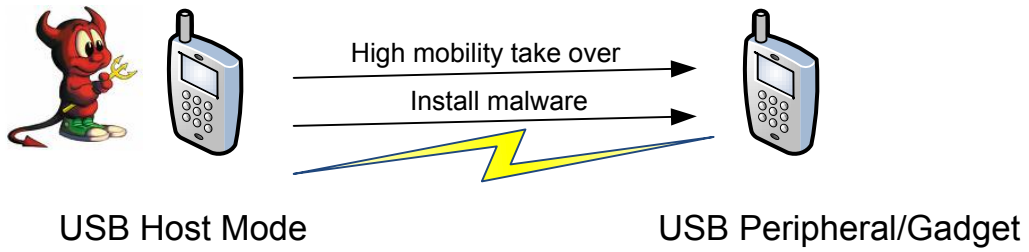


Figure 4: The Phone-to-Phone Attacks over the USB Connection.

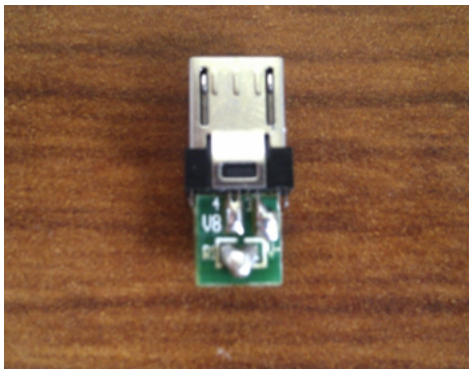


Figure 5: The Micro B USB Connector Dongle.

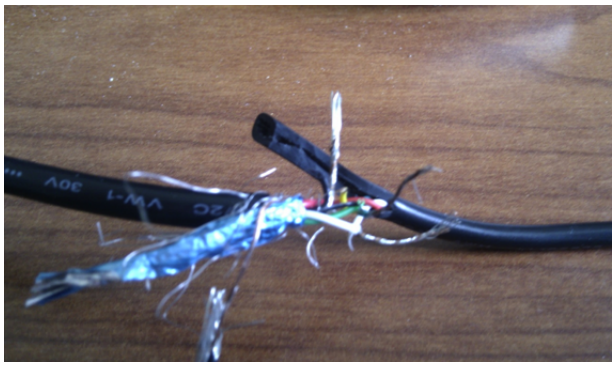


Figure 6: The Crafted USB Cable for Phone-to-Phone Attacks.

cess merely takes 2 seconds for a 2560KB *boot.img* file by issuing command *fastboot flash boot boot.img* where *fastboot* is a program having the minimal functionality of maintaining the device in boot-loader mode (e.g. updating partitions of the device). This program is available for Windows, Linux, and Mac OSX. After all the above steps, we have gained full control of the victim device and prepared automated launching of the malicious code. We reboot the phone back to normal mode from boot-loader mode and push our malicious binary to the system partition by *adb push evilprog /system/xbin* and change the permission for execution. The detailed malicious action that this evil binary can do is beyond the scope of this paper. For proof-of-concept demonstration purposes, we wrote a program for collecting the device information and send them to a pre-configured internal collection server stealthily over TCP/IP via cellular data network or wireless network whichever available. This program is cross-compiled against Android's bionic C libraries with arm-eabi toolchains. Some more developed and foreseen real attacks are discussed in Section 3. Note that this program is written in C and executed as the ARM ELF binary at the system utility level which is lower than Davik Java virtual machine and bypass all Android's permission checks for application at JVM [6]. Our server successfully collected the device information sent by the program, which includes the serial number of the device, the kernel version and a list of installed applications.

As we mentioned earlier in this section, all the above logic and operation sequences are programmed as a malicious daemon running on the desktop system. The complete process takes 300 seconds, which corresponds to the sum of every steps.

2.4 Phone-to-Phone Attacks

The inherent mobility and programmability of the third-generation smart phones gave rise to a new type of insider attack. The phone is fully capable of assuming the role of a computer host by setting its USB port to be a USB Hub. This type of attack is similar to the attacks described in Section 2.3. For phone-to-phone attacks, a malicious user connects a subverted device to a victim device and then take over it stealthily. This can happen, for instance, when the victim device is left unattended. In this section, we show how to perform a phone-to-phone attack via a single USB interface as the infection vector. The key capability is to enable the USB host mode on one device, a Motorola Droid in our case, which first time provides the ability of controlling a Android device from another Android device. The rest of the attack is similar to the one described in Section 2.3. When the manipulated Motorola Droid device connected to another device, the malicious daemon will send pre-programmed command and the victim device will treat it as from a normal desktop system.

For our purposes, we leverage the advanced USB chip in recent released Google Nexus One by HTC and Motorola Droid devices and enable the device's USB host mode capabilities. In regular operation, the phone devices only act as peripheral devices at the USB protocol level. The desktop system will send the first USB packet and initiate the USB connection link. We instead enable the USB OTG (On-the-Go) driver in the device with such hardware support, and flip a normal smart phone device as the USB host. To be more specific, both Nexus One's Qualcomm QSX8250 chipset and Motorola Droid's Texas Instruments OMAP3430 chipset support USB OTG specification [5]. Our experiment on Google Nexus One device failed due to limited SoC depended kernel code support for Qualcomm QSX8250 chipset. However, the OMAP series chipset integrated with the Philips ISP1301 USB OTG transceiver has more mature code in the kernel source. By checking the following kernel compilation options, we can enable the OTG software.

```
CONFIG_ARCH_OMAP_OTG=y
CONFIG_USB_OTG=y
CONFIG_USB_MUSB_OTG=y
CONFIG_USB_OTG_UTILS=y
```

After we activate the kernel driver, we need the specially crafted USB connectors and cable to trigger the USB host mode of the USB OTG device and connect other peripheral devices. By soldering the 4th pin and 5pin of the micro USB connector from a car charger, we changed a micro B connector to a micro A connector, to identify itself as a host side connector. Unfortunately, most off- the-shelf product do not specify it is a A connector or a B connector. Figure 5 shows the micro B dongle we had to solder to achieve our goal. To place the device in the USB hub mode, we have to perform a hard reboot while the micro B connector is inserted in the Droid USB interface. Moreover, we have to unplug the micro-dongle as soon as the Motorola logo disappears as the Droid logo appears. This forces the hardware initialization process to identify the USB hardware in the host mode. After the system boots up, we can verify that the USB is in host mode by running the following command "cat /sys/devices/platform/musb_hdrc/mode". If the output of the command is "a_host" then we are in host mode. Notice that we need to enable the wireless

connectivity and use secure shell connection for shell access because the USB interface is in host mode and thus traditional *adb* shell access over USB is disabled.

To connect other peripheral devices, in our case a victim phone, we make the special USB cable with both end micro USB by cutting two cables and put two micro connector in a single cable by soldering the same color together. Our additional experiments shows the device can support additional USB-to-Serial converter but for USB flash driver devices, we have to use external USB power hub to supply additional power to the Vcc line. Figure 6 depicts a snapshot of the cable we made with the micro USB connectors at both ends. It is worth mentioning here that due to the requirement that the D+ and D- must be twisted for synchronization purposes, we can only break the cable within a limited distance for soldering.

Another important aspect of the attack is that the peripheral device driver must be compiled in the host mode device. To limit unnecessary code, most of the non-required kernel options and device drivers are turned off by manufacture configuration. We performed our experiments using a Motorola Droid to attack a Nexus One phone. The generic USB hub driver on the Droid kernel is compiled as part of the Linux Kernel. The final step is compiling the user level program against the Android system libraries. *adb* provides the ability of controlling a Android device from another Android device. The rest of the attack is similar to the one described in Section 2.3 where the host is replaced with the Droid device. When the malicious Motorola Droid device connects to the victim device, the malicious daemon will send the pre-programmed command over the USB and the victim device will treat it similarly as it did for the host computer.

3 Discussion

Our attacks are primarily implemented on the Android framework because of its open source nature and the ease that we can demonstrate and detail our results making them reproducible. However, we posit that attacks that abuse the USB physical link and hardware programmability exist also for other mobile phone platforms such as the Apple iPhone OS, Microsoft Windows CE and Symbian OS. Moreover, there are scenarios where the described classes of attacks are easier to be accomplished on other platforms. Taking iPhone OS as an example, an adversary can take advantage of the default music play functionality that iTunes software offers to craft malware media files and "synchronize" them with the connected computer. In addition, antivirus products normally scan the external storage in the device which appears as a flash drive from the operating system's view. However, such scans are based on well-known file formats and none of them can scan the internal ROM or raw data stored in the hand-held devices, to the best knowledge of the authors. This represents a clear defense gap.

The common theme behind the USB attacks is the established belief that physical cable connectivity can be inherently trusted and that peripherals are not capable of abusing the USB connection. To protect the end-point devices, there is a need to shed that belief. Instead we have to focus on how to establish trust that is not implicit but explicit and puts the human on the loop. Therefore, a possible defense strategy is to authenticate the USB connection establishment phase and communications using similar techniques that

were developed for Bluetooth devices. This will give a visual input to the user and will allow her to verify that a device that attempts to connect as a peripheral is indeed allowed to connect. Moreover, there is a need to identify and communicate to the user the type of the USB device that attempts to connect as a peripheral. This will prevent attacks that pretend to be HID devices and connect without any user interaction.

Unfortunately, attacks that exploit the USB while the victim device is in "slave" mode are more difficult to thwart because some of the functionality is required to control the "slave" device. However, smart phone vendors can try to filter and vet the USB communications using a USB firewall. Similar to network firewall, this USB firewall will inspect all USB packets coming to the device and check the content based on platform-specific rules preventing attacks that replay key-strokes via the USB bypassing the user-input.

In the meantime, we can protect the smart phone system by performing a full backup. This is an easy solution and feasible for most mobile devices. Indeed, the internal ROM storage is relatively limited on smart phones, 512 MB in our case. Using a program that runs on the phone, we can easily dump the entire filesystem using prior knowledge about the partition information to a back-end desktop systems or even external sdcard storage. Note that such backup is the complete filesystem, which includes boot partition and kernel binaries. If the backup is performed from a clean state, a simple revert can defeat all persistent malware even rootkits. However, restoring the phone to a pristine state might lead to loss of user personalization data and thus, it can only act as an emergency measure and not a full-proof or even user friendly solution.

4 Acknowledgements

We would like to thank Nelson Nazzica, Quan Jia, Meixing Le and Jiang Wang from the Center for Secure Information Systems at George Mason University for their comments on our early draft. We also thank the anonymous ACSAC reviewers for their constructive comments. This work was supported in part by US National Science Foundation (NSF) grant CNS-TC 0915291 and a research fund from Google Inc. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Autoplay in windows xp: Automatically detect and react to new devices on a system. `http://msdn.microsoft.com/en-us/magazine/cc301341.aspx`.

[2] Nexus one features and specifications. `http://www.google.com/phone/static/en_US-nexusone_tech_specs.html`.

[3] Usb 2.0 specification. `http://www.usb.org`.

[4] Usb gadget api for linux. `http://www.kernel.org/doc/htmldocs/gadget.html`.

[5] Usb on-the-go. http://www.usb.org/developers/onthego/.

[6] Enck, W., and McDaniel, P. Understanding android's security framework. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 552–561.

Technical Details for the USB Connectivity

Detailed Instruction to craft the USB host cable:

- We start with the car charging cable from Version Store. Break open the micro-usb connector (it comes apart fairly easily) and look at the little PCB inside there should be a single tiny surface-mount resistor and two wires from the charger cable. Unsolder both wires and the resistor, and then bridge the pads where the resistor used to be so that its completely shorted.
- . Connector cable. Cut the end of the USB extender cable, you want to keep the MicroUSB B socket end and USB A Female socket, wire and solder the same color together. When its finished it should look something like the picture.
- External power supply, use a

  USB hub The connectivity should be like the following:

  — USB stick (or USB keyboard) s A/Male ⟷ USB HUBs A/Female
  — USB MiniA/Male ⟷ USB A/Male
  — USB A/Female ⟷ 4 color matching wires
  — USB MicroB ⟷ Android Phone

  If you can get stock cable like this from store, you can bypass this step. Directly supply power to Vcc(Red) and GND(Black) in the middle of the cable does not work. My guess is: there is some sync signal in power lines for Vbus to be identified and working. You cannot break the white and blue data lines completely because they are wired and the signals inside these 2 lines are synchronized. If you separated them to long, it will break the synchronization and result no USB protocol communication.
- You dont even need to root your droid in order to verify it works (although I rooted mine), just do the following: - Turn your Droid off - Plug the micro-dongle into the USB port - Turn the droid on - Unplug the micro-dongle as soon as the Motorola logo disappears (as the Droid logo is appearing).

Once your Droid is booted, pull up a terminal and look at dmesg
"cat /sys/devices/platform/musb_hdrc/mode" will give you b_idle After plugging in your

USB peripheral using the cable you made earlier you should see the usual kernel notifications about new USB devices being connected; lsusb utility will show the vendorID and productID of the device. Due to lack of scis driver in the kernel(you still can add it.), you may not get your USB stick filesystem working, but for USB keyboard, it works perfectly, and you can type like a normal computer.

"cat /sys/devices/platform/musbi_hdrc/mode" will give you the response a_host Theyll also turn on (or start charging) if theyre powered by USB. Youll only be able to plug in one peripheral before the port reverts to peripheral mode; youll have to reboot with the micro-dongle if you want to go back into host mode. Also, if you leave the micro-dongle plugged in too long it triggers another bug; the port gets stuck supplying power to devices but not actually recognizing them.