Android Malware: A Look Behind The Fear Uncertainty Doubt

Andrei Matetic

Spring 2015

Table of Contents

Table of Figures

Table of Tables

Abstract

Mobile devices are a near constant companion and allow their owners to perform a variety of functions and tasks. Malware has spread to these devices and has acquired menacing capabilities over time. Android devices, comprising the majority of global devices, are a popular target for malicious software. Being such a high profile target has necessitated that Android in turn respond to the malware threat. Android has undergone architectural changes and existing defenses were strengthened to stem the spread of malicious software.

1. Introduction

No one carries a cellphone any longer. In terms of processing power, these devices have more in common with miniature laptop computers than mobile phones. The amount of time spent with digital devices has been increasing, up 24% to 60% in 2014. [1] These devices are being used for far more than making phone calls: email, messaging, web surfing, mobile banking, shopping, and media consumption. As mobile devices and their use have evolved and matured so has the nature of mobile malware.

Malware, malicious software, for mobile devices has been present for over 10 years. Over this time, its capabilities have expanded beyond using Bluetooth to spread from device to device and drain the battery. Today, mobile malware is able to: send and receive SMS messages, exfiltrate user data (e.g. contacts, application data, and local files), manipulate (e.g. record, redirect, and block) phone calls, lock a user out of their device or data, and erase data from the device. [2]

In a pattern similar to traditional desktop malware, mobile malware followed a similar shift in motivation. What had started out as driven by curiosity has shifted to be driven by profit. [3] Mobile malware can earn money for its authors through click fraud (luring the infected device to click on web links to influence search rankings) [4], subscribing the user to premium SMS services, and serving ads. [5] Another monetary avenue for mobile malware is ransomware. A portmanteau of ransom and software, ransomware extorts money from the user based on some threat. The threat could be real (blocking applications or encrypting files) [5] [6] or fictitious (being labeled a criminal under investigation by the FBI on child pornography allegations). [7]

Mobile malware has uses outside of monetary ones. AndroRAT (a combination of Android and RAT [Remote Access Tool or Trojan]) provides backdoor access into a user's phone allowing access to contacts, messages, call logs, and recording audio and capturing images. [8] Devices infected with NotCompatible.C end up joining a botnet that can be "rented" as a service. This botnet can be used for a variety of activities including: [9]

- Spam campaigns (Live, Aol, Yahoo, Comcast)
- Bulk ticket purchasing (Ticketmaster, Livenation, Eventshopper, Craigslist)
- Bruteforce attacks (WordPress)
- c99 shell control (observed logging into shells and performing different actions)

There are 173 million smartphone subscribers in the United States and nearly all run either Apple's iOS or Android. Domestically, a little over half (51.9%) of those devices run the Android mobile operating system [10], and globally Android's market share increases to 87%. [11] Android is developing the reputation of being the *Windows* of mobile operating systems—both in its adoption across a variety of devices and more notably in its perceived insecurity [3] with estimates of the Android malware share ranging from 97% to 99%. [11] [12]

Is this reputation deserved? What mechanisms does Android implement for device security? Are there peculiarities that make Android devices more vulnerable to malware? Some rudimentary analysis will be performed on a chosen piece of malware.

2. Android Architecture

Google is often considered synonymous with Android, but they are only one of the parties responsible for the mobile operating system known as Android. Viewed either as an operating system or software stack (Figure 1), Android is the product of the Open Handset Alliance and was intended to be an open platform supporting a variety of form factors, carriers, OEMs (Original Equipment Manufacturers) and developers. [13] [14] The Android Open Source Project (ASOP) is available for anyone to build and ship with an Android device. Products like Google Maps, Google+, Gmail, Google Play, are not part of Android proper. They are part of Google Mobile

Services (GMS), a collection of applications and API available from Google under a separate license. [15] Handset manufactures may choose not to include GMS on handsets due to either licensing costs [16] [17] or data privacy concerns [15].

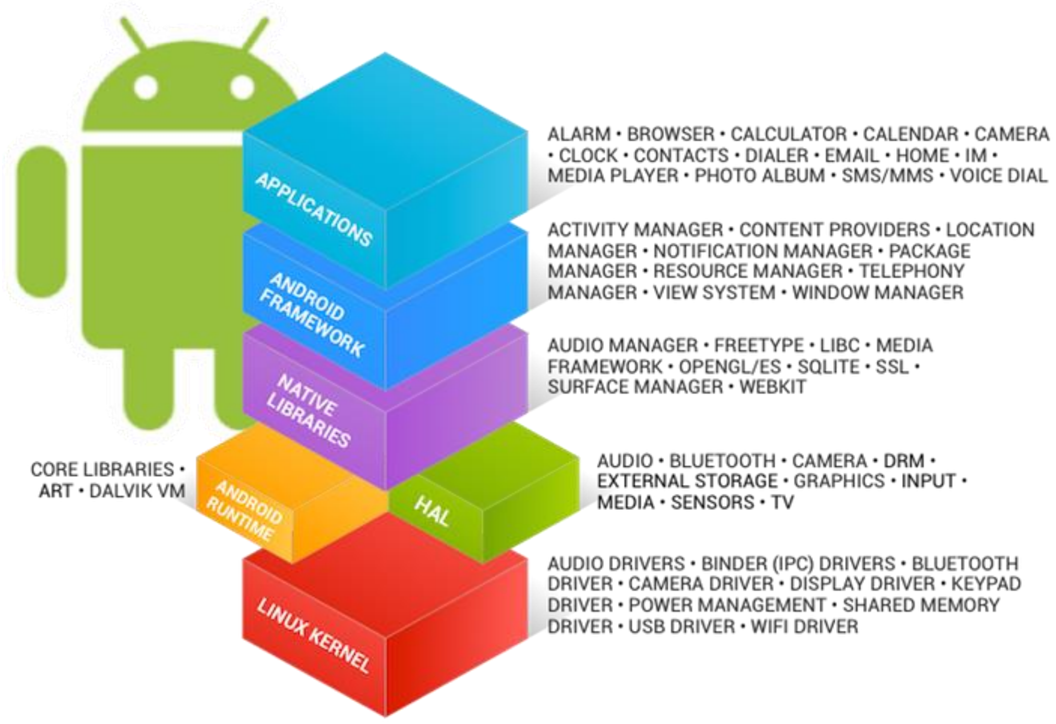

APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
ACTIVITY MANAGER • CONTENT PROVIDERS • LOCATION MANAGER • NOTIFICATION MANAGER • PACKAGE MANAGER • RESOURCE MANAGER • TELEPHONY MANAGER • VIEW SYSTEM • WINDOW MANAGER
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
CORE LIBRARIES • ART • DALVIK VM
ANDROID RUNTIME
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
AUDIO DRIVERS • BINDER (IPC) DRIVERS • BLUETOOTH DRIVER • CAMERA DRIVER • DISPLAY DRIVER • KEYPAD DRIVER • POWER MANAGEMENT • SHARED MEMORY DRIVER • USB DRIVER • WIFI DRIVER

Figure 1 The Android Stack

Applications are the user’s entry point into an operating system (desktop or mobile), and perform specific functions like: browsing the web, accessing social media, email and messaging, audio and video media consumption. The class of applications can be divided into two categories: system and user. System applications are considered “stock” being part of either Android or preinstalled by an OEM as part of their user experience skinning. User applications are installed by the owner of the device. As part of its flexibility and customizability, users can “replace” the functionality performed by stock applications with third party alternatives. [18]

The components of the Android Framework provide the building blocks for applications and allow access to a device’s basic functions and processing capabilities.

Android applications are written in Java. The Dalvik VM (Virtual Machine) that executes the bytecode runs in the Android Runtime layer and is architecturally different from Oracle's JVM (Java Virtual Machine). After compilation, the .class bytecode files are converted into a Dalvik executable (.dex) format. Beginning with Android 5.0, the Dalvik VM was replaced with the Android Runtime (ART) VM which is backwards compatible with Dalvik.

The basis of the Android stack is built on top of a Linux kernel and provides the interfaces for the software to talk to the underlying hardware. There are Android specific customizations in the kernel: low memory killer, paranoid networking, and secure interprocess communications mechanism called Binder. [18]

3. Android Security

Android packages (APKs) are the result of the Android build process and are the container for all the resources (e.g. compiled byte-code) required for an application to function on a device or in an emulator. [19]

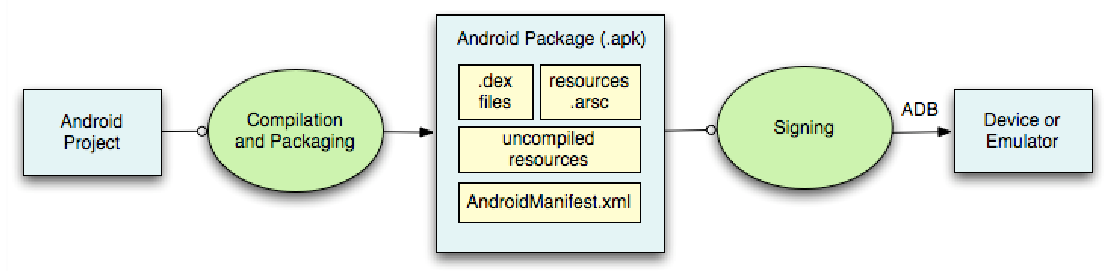
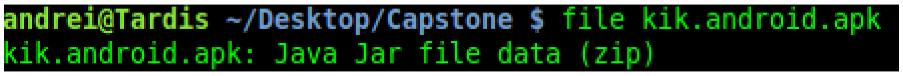


Figure 2 Android build process

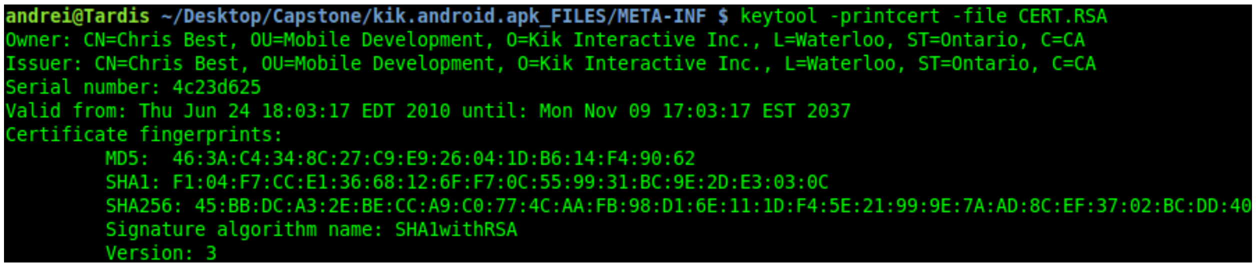
As shown with the Linux **`file`** command in Figure 3 an APK is a type of zip file and can be unpacked with any tool that supports that format.



```
andrei@Tardis ~/Desktop/Capstone $ file kik.android.apk
kik.android.apk: Java Jar file data (zip)
```

Figure 3 file command run against APK

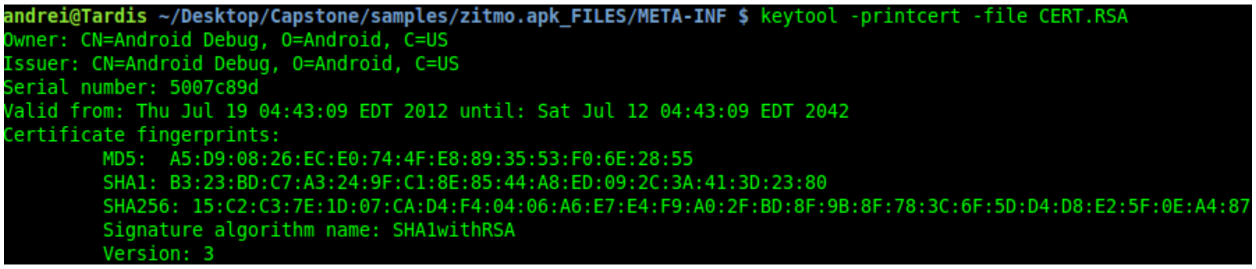
Before an application can be installed, it must be signed with a certificate. Certificates are used to identify the author of an application and are not required to be signed by a certificate authority. [20] Packaged as part of the APK is a RSA file (e.g. `CERT.RSA`) that can be inspected for certificate related information with the **`keytool`** command as shown in Figure 4.



```
andrei@Tardis ~/Desktop/Capstone/kik.android.apk_FILES/META-INF $ keytool -printcert -file CERT.RSA
Owner: CN=Chris Best, OU=Mobile Development, O=Kik Interactive Inc., L=Waterloo, ST=Ontario, C=CA
Issuer: CN=Chris Best, OU=Mobile Development, O=Kik Interactive Inc., L=Waterloo, ST=Ontario, C=CA
Serial number: 4c23d625
Valid from: Thu Jun 24 18:03:17 EDT 2010 until: Mon Nov 09 17:03:17 EST 2037
Certificate fingerprints:
         MD5:  46:3A:C4:34:8C:27:C9:E9:26:04:1D:B6:14:F4:90:62
         SHA1: F1:04:F7:CC:E1:36:68:12:6F:F7:0C:55:99:31:BC:9E:2D:E3:03:0C
         SHA256: 45:BB:DC:A3:2E:BE:CC:A9:C0:77:4C:AA:FB:98:D1:6E:11:1D:F4:5E:21:99:9E:7A:AD:8C:EF:37:02:BC:DD:40
         Signature algorithm name: SHA1withRSA
         Version: 3
```

Figure 4 Certificate information for an kik.android.apk

APKs can be signed in "debug mode" where the Android Software Development Kit (SDK) generates the certificate [20] with a known password. [19] Figure 5 shows an example of an APK signed in debug mode.

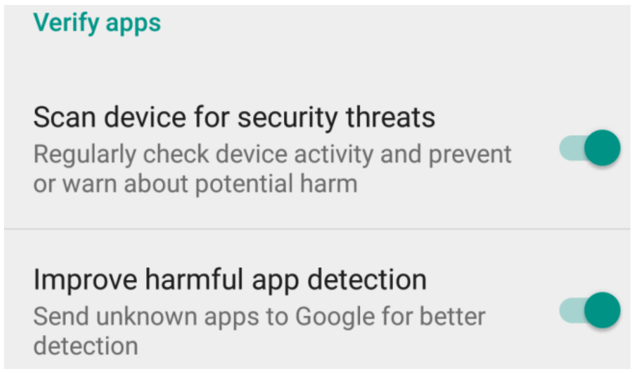


```
andrei@Tardis ~/Desktop/Capstone/samples/zitmo.apk_FILES/META-INF $ keytool -printcert -file CERT.RSA
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 5007c89d
Valid from: Thu Jul 19 04:43:09 EDT 2012 until: Sat Jul 12 04:43:09 EDT 2042
Certificate fingerprints:
         MD5:  A5:D9:08:26:EC:E0:74:4F:E8:89:35:53:F0:6E:28:55
         SHA1: B3:23:BD:C7:A3:24:9F:C1:8E:85:44:A8:ED:09:2C:3A:41:3D:23:80
         SHA256: 15:C2:C3:7E:1D:07:CA:D4:F4:04:06:A6:E7:E4:F9:A0:2F:BD:8F:9B:8F:78:3C:6F:5D:D4:D8:E2:5F:0E:A4:87
         Signature algorithm name: SHA1withRSA
         Version: 3
```

Figure 5 APK signed in debug mode

As part of the application install, the signature of the APK is verified to ensure that the application has not been tampered with and that certificate comes from the expected developer. [21] An error is displayed if signature verification fails.

Application verification was added to Android version 4.2 and brought back to version 2.3 via updates. It is performed during application install and periodically afterwards (Figure 6), and both functions are enabled by default. Android provides the framework for verification but ships with no verifiers. Application verification is performed as part of applications installed through the Google Play Store. [18]


Verify apps

Scan device for security threats
Regularly check device activity and prevent or warn about potential harm

Improve harmful app detection
Send unknown apps to Google for better detection

Figure 6 Verify apps settings

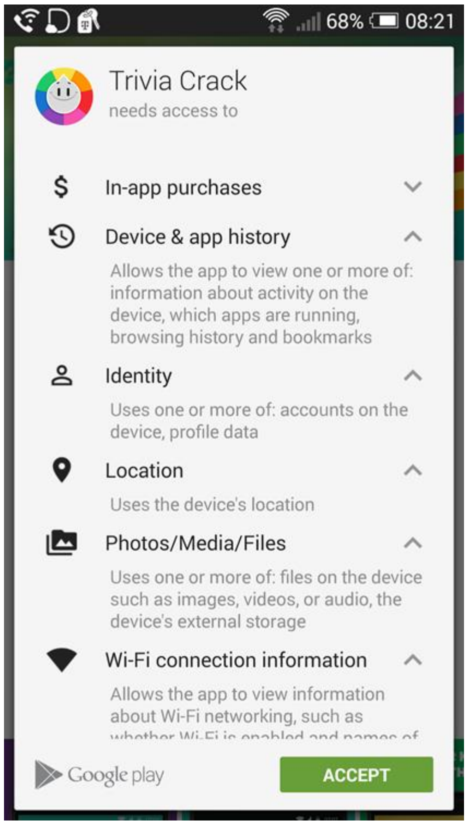
Android draws from its Linux heritage as a multiuser and multiprocess system. The kernel enforces segregation between individual users as well as the processes (jobs) that users may run. Without explicit permissions, one user can not touch another's files and one process cannot interfere with another. In a physical system, Linux uses a user ID (UID) to distinguish between different users that can logon and interact with it. As a mobile device, smartphones have one physical user. Android assigns a UID to each application instead. This assignment forms the basis of sandboxing. [18]

Applications run as processes and are sandboxed. Sandboxing prevents an application from "perform[ing] any operations that would adversely impact other applications, the operating system, or the user." [22] For functions that lie outside of the application sandbox, permission needs to be obtained from the user of the device. Included as part of the APK is the `AndroidManifest.xml` file where the application's permissions are specified as shown in Table 1.

Table 1 Sample AndroidManifest.xml permissions listing

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="kik.android.permission.CONTACT"/>
<uses-permission android:name="com.android.vending.BILLING"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

At install time, the user is presented a dialog listing the permissions that the application is requesting (Figure 7). If an application's permissions change during the course of an update, the user is presented with a prompt to accept the newly requested permissions (Figure 8). Permissions must be accepted in total or not at all (i.e. selecting not to install the application or the update). Well intentioned, the permissions dialogs are often a speed bump for users. Apps do a good job of explaining what a permission is, but seldom is the *why* of a permission being needed explained. Why does Trivia Crack need to know my device's location (Figure 7) and why is Kik, a messaging application, now interested in using my microphone (Figure 8)?


68% 08:21

Trivia Crack
needs access to

$ In-app purchases

Device & app history
Allows the app to view one or more of: information about activity on the device, which apps are running, browsing history and bookmarks

Identity
Uses one or more of: accounts on the device, profile data

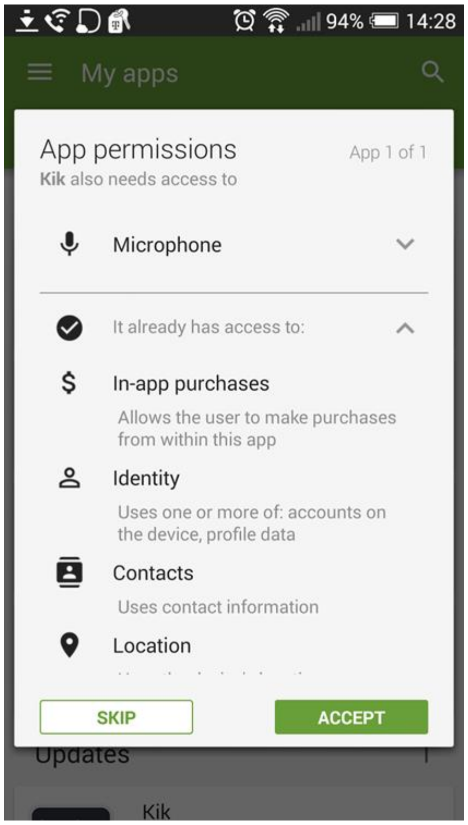
Location
Uses the device's location

Photos/Media/Files
Uses one or more of: files on the device such as images, videos, or audio, the device's external storage

Wi-Fi connection information
Allows the app to view information about Wi-Fi networking, such as whether Wi-Fi is enabled and names of

Google play ACCEPT

Figure 7 Permissions prompt for application install


94% 14:28

My apps

App permissions App 1 of 1
Kik also needs access to

Microphone

It already has access to:

$ In-app purchases
Allows the user to make purchases from within this app

Identity
Uses one or more of: accounts on the device, profile data

Contacts
Uses contact information

Location

SKIP ACCEPT

Updates

Kik

Figure 8 Permissions prompt during an application update

Permissions represent a way out of the application's sandbox and should be treated with caution. The principle of least privilege should be employed and applications should only request the permissions that they need or design the application in a way as to avoid having to request a permission. [23] Over permissioning occurs when an application requests more permissions than are needed. In the case of repackaged APK, malware would inherit all the permissions of the original application. A 2011 survey of almost 1000 applications found about one-third were over permissioned. Of the over permissioned applications, roughly half requested only one additional permission. [24] In their analysis, [24] identified developer confusion as the prime cause for over permissioning in applications:

- Permission names may sound related to the function the developer wants to use

- Handing off a function (e.g using the camera) to a deputy or proxy requires the deputy to have the required permission and not the original application
- In a mixed permission class, requesting a write permission when only get or read methods are called.
- Permissions that were required in earlier releases that are no longer required
- Testing or debug artifacts left in production code

In addition to application permissions, the `AndroidManifest.xml` file specifies the component structure of the application. Android applications contain one or more of the following: [18]

Activity	Activities are the building blocks of an application's GUI. A single activity is a screen that provides a user interface to the user.
Services	A service runs in the background and does not require a user interface. Services generally handle long running processes (e.g. downloads) so that the main activity does need to block.
Broadcast receiver	A broadcast receiver is a handler for system events (i.e. broadcasts).
Content provider	A content provider is the mechanism used to access an application's data.

Intents are messaging objects that application components use to communicate with each other and request actions to be performed. Inside the application's manifest, an intent-filter can be specified that only the specified intents are listened to. [25]

4. Infection

There are several infection vectors for mobile devices. The utility of any mobile platform is driven in part by its app ecosystem – what's in its "app store". A source of infection are applications from nontrusted sources – "The majority of problems arise from the installation of 'cracked' applications from 3rd party market places which are often bundled with malicious software." [26]

Malware does exist on Apple's iOS mobile platform. [27] [28] Quoting Lookout's CTO, Kevin Mahaffey, [29] notes "Bad guys are rational economic actors. Because Android is so much more popular in the world they're targeting the largest platforms first." Part of Android's success is due to being available on a variety of devices at varying price points. The AOSP source is open and can be downloaded, studied, and built by both enthusiasts and malicious actors. Apple's official App Store is the only mechanism without a jailbreak for iOS users to purchase applications. If the handset manufacturer includes Google Mobile Services (GMS), the Google Play Store is the default source for applications on Android devices. It also allows for a user to opt into (Figure 9) installing software from an unknown source. Enabling this option presents the user with the warning dialog in Figure 10 that needs to be accepted.

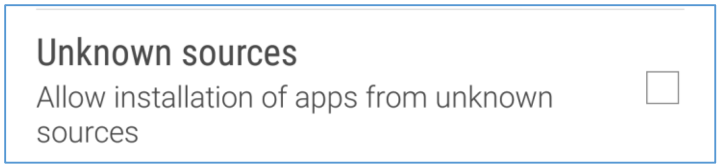
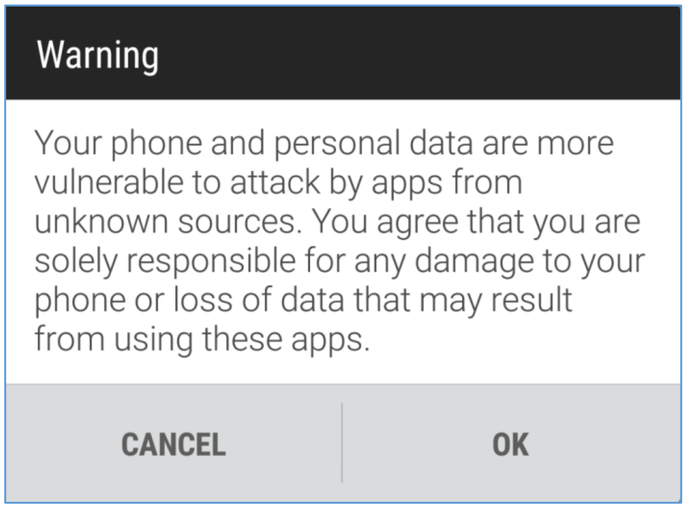

Unknown sources
Allow installation of apps from unknown sources

Figure 9 Install apps from unknown sources

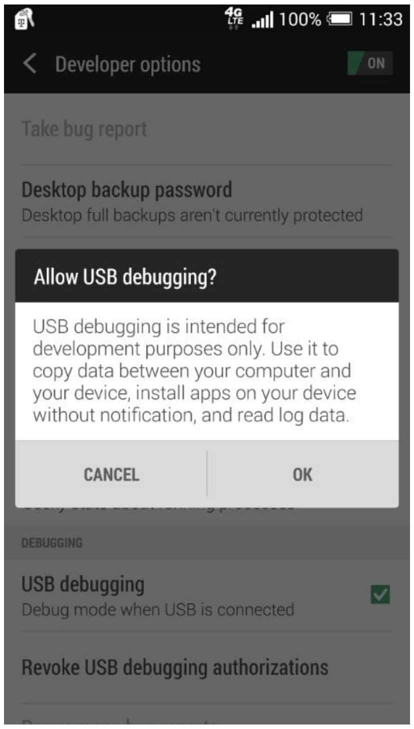

Warning

Your phone and personal data are more vulnerable to attack by apps from unknown sources. You agree that you are solely responsible for any damage to your phone or loss of data that may result from using these apps.

Figure 10 Unknown sources warning

Visiting a compromised website, as with the NotCompatible variants, may download malware to the device. With some social engineering, the user is tricked into installing the malicious application.

Either to charge or transfer data, mobile devices spend some part of the day connected to a computer. After infecting the host computer, Windows malware Trojan.Droidpak downloads the necessary pieces, ADB (Android Debug Bridge) and AV-cdk.apk, to infect an attached Android device. [30] ADB is a command line tool that allows for communication to an emulator session or attached device and can be used to install an APK to the connected device (e.g. **`adb install <path-to-apk>`**). [31] Installing an APK in this way does not prompt the user to accept any application permissions. For ADB to work, USB debugging needs to be enabled on the device, and without USB debugging, Trojan.Droidpak is not able to infect the device. Enabling USB debugging is hidden away as a developer option, and a warning dialog (Figure 11) is displayed that needs to be accepted. Connecting with ADB now prompts the user for permission (Figure 12).



Desktop backup password
Desktop full backups aren't currently protected

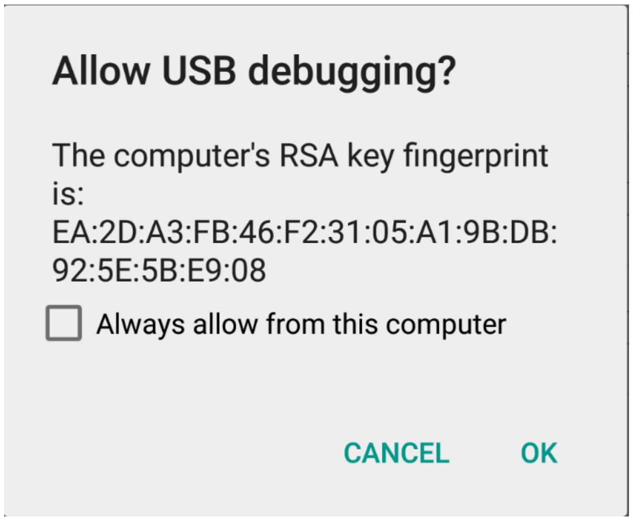
Allow USB debugging?

USB debugging is intended for development purposes only. Use it to copy data between your computer and your device, install apps on your device without notification, and read log data.

USB debugging ✓
Debug mode when USB is connected

Revoke USB debugging authorizations

Figure 11 Enabling USB debugging



Allow USB debugging?

The computer's RSA key fingerprint is:
EA:2D:A3:FB:46:F2:31:05:A1:9B:DB:92:5E:5B:E9:08

☐ Always allow from this computer

Figure 12 Prompt to Allow USB Debugging

5. Mitigations

Updating an Android device to a new release (e.g. 5.0 Lollipop) is an involved process that is complicated with multiple players and device types. The source code of an update is released to both chipset manufacturers and device OEMs (e.g. HTC) for their evaluation. Assuming the chipset manufacturers are willing and able to support the new release, they create the required drivers and optimizations for the OEMs. If the OEM is able to support the release, the next path depends on device branding: carrier, developer/unlocked, or Google Play edition. Carrier (e.g. AT&T, Verizon, Sprint, or T-Mobile) devices are modified with that carrier's selection of UI skin and applications. A developer/unlocked phone may receive minimal customization from the OEM, and a Google Play edition phone receives no carrier or OEM customizations at all. Devices are tested and bugs addressed before being sent for technical acceptance from both the carrier if required and Google. Once technical acceptance is granted an OTA (Over the Air) update is prepared to be pushed to customer's devices. [32] [33] Apple's apparent "speed" is an "illusion"

– they have a similar set of steps to go through, and the amount of time is about the same. Apple's tighter control over the hardware platform simplifies some of their testing. Being their own OEM, Apple announces a new iOS release when it is ready for end-user consumption. New Android releases are announced at the midpoint, when they are ready for device OEMs use. [33]

Somewhere in a device's lifecycle, it is no longer able to run new versions of the Android OS and becomes unsupported. Being unsupported garners a device no new features or fixes. Any vulnerabilities discovered would persist unpatched and potentially exploitable by an attacker. Google announced in late January 2015 that a vulnerability discovered in the Webkit browser engine affecting the stock web browser application and has been corrected in Android version 4.4 and later. [34] Versions of Android (4.3 and earlier) vulnerable to the flaw account for about 60% of devices, ranging from version 2.2 to 4.3. [35]

The question of antivirus for Android devices is contentious. The first page of search results for "android antivirus" [36] has the following results:

- *14 best antivirus Android apps - Android Authority*
- *Best Android Antivirus Apps 2015 - Mobile Security Software*
- *Test antivirus software for Android - January 2015 | AV-TEST*
- *AntiVirus Security - FREE - Android Apps on Google Play*
- *Mobile Security & Antivirus - Android Apps on Google Play*
- *Best Android antivirus 2015 UK - PC Advisor*
- *Do you need antivirus on Android? We ask the experts ...*
- *Free Antivirus for Android - Download the best ... - Avira*
- *Free Antivirus for Android™ | Antivirus App for Smartphones ...*
- *AVG AntiVirus FREE for Android™ Mobiles Overview | AVG ...*

Of these, only one asks if antivirus is truly necessary. Vendors want consumers to buy their products. As one of their best practices, Symantec recommends to "Install reputable security software, such as Norton Mobile Security". [30] In the press, opinions on the need for antivirus ranges from lukewarm “just in case” to it being part of an overall solution to leaving an individual’s device at risk if not installed. [26] [37] [38] At the other end is the opinion that antivirus is useless and that “virus companies are playing on your fears to try to sell you bs protection software for Android, RIM and IOS.” [39]

Research by [40] gives credence to the notion that antivirus on Android is not as effective as vendors claim. The problem is twofold: “retrospective analysis” as termed by [40] and Android’s application sandboxing. Retrospective analysis is based primarily on signatures of known malware samples. These signatures are based on cryptographic hashes (e.g. SHA1 or MD5). Minor changes in the malware package, without having to alter the DEX or ART bytecode, would result in a different hash and produce what looks to be a new instance of malware. In their testcases [40] showed a precipitous decline in detection rates of altered malware samples among their suite of Android antivirus products. The application sandbox is intended to make sure applications do not run amok and create an undesirable user experience on the Android platform. Unlike on a desktop, Android antivirus is treated and restricted like any other mobile application: it can only access its own data and process information. A scan of the filesystem is not currently possible. Not able to see what other processes are doing, malware that initially looks benign before downloading code (known as a dropper) could not be detected either. As a remedy, [40] proposes adding a system interface that would allow antivirus products these types of capabilities in exchange for stricter certificate verification requirements.

Where antivirus can be useful is at the front gate – controlling access to the app store. Referred to as “Bouncer”, submissions are scanned by Google for known malware. A submitted application’s behavior is analyzed by running the application looking for malicious signs and patterns. From their data, Google saw a 40% decrease in the number of malicious downloads between the first and second halves of 2011. [41] Bouncer is not without its flaws. As [42] notes, malware could evade being detected by Bouncer by not performing any malicious activity when being evaluated (i.e. behave for the 5 minutes of Google's runtime) or downloading malicious code when running on actual hardware.

Permissions are how applications access resources outside of its sandbox. As shown previously in Figure 7 and Figure 8, the user is not able to pick and choose which permissions to accept during application install or update. Applications like LBE Privacy Guard[1] can provide a middle ground and act as sort of permissions firewall. Users can select which permissions to allow on an app by app basis (e.g. allow a game internet access but block sending SMS messages or reading the device's phone number). [43] When an application breaks due to a missing permission, it can be added back. For this type of permission granularity, root access to the device is required.

From its Linux background, Android carries with it the root user, a user that has complete access to the system. Normally, a device's owner does not have access to the root user. Rooting is the process the owner uses to obtain root (sometimes called superuser) access to the device. The process is device and release dependent. In general terms, the device is placed in an S-Off (Security Off) state and a custom image is loaded containing root access. The S-Off state is required to gain write access to the system partition which is normally read only. Great care is required, but gaining root access gives a user even greater flexibility over their device:

- Tweak CPU and GPU frequencies along with other performance settings[2]
- Uninstall unwanted system applications[3]

[1] https://play.google.com/store/apps/details?id=com.lbe.security.lite
[2] Device Control [root] https://play.google.com/store/apps/details?id=org.namelessrom.devicecontrol
Kernel Tuner **root** https://play.google.com/store/apps/details?id=rs.pedjaapps.KernelTuner
[3] System app remover (ROOT) https://play.google.com/store/apps/details?id=com.jumobile.manager.systemapp

- Manage privacy and what data an application is allowed to access[4]

As the benefits of root access could also be leveraged by malware, the decision to root a device needs to be weighed carefully. However, [43] argues the opposite position that rooting can be used improve device security. As part of the rooting process or recommended install, the SuperSU[5] application allows for the user to manage which applications can have root access. Prompting the user for root access makes "threats such as DroidKungFu [...] less intimidating, [as] it can do very little at a root level if that root [access] is controlled and monitored." [43] Over time, Android has limited the number of processes that need to run as root. Android version 4.4 is the first version with SELinux in enforcing mode and included enforcing domains for core system daemons. Should a privilege escalation attack gain root access it is not guaranteed that full unrestricted access to the device is available. [18]

6. Case Study: AdobeFlashPlayerUpdate.apk

The `AdobeFlashPlayerUpdate.apk` sample for this case study was acquired from androidsandbox.net, an online tool for analyzing Android applications. [44] There are three forms of analysis:

- Static Analysis
- Dynamic (Behavioral) Analysis
- Reverse Engineering

No code is executed during static analysis. Instead the components of the APK are reviewed (e.g. permissions from the manifest), extract the certificate information, cryptographic hashes of the components are computed, string data is extracted and other metadata is collected and reviewed. [5] A site like virustotal[6] can automate many of these steps. Uploading a sample to

[4] [ROOT] X Privacy Installer https://play.google.com/store/apps/details?id=biz.bokhorst.xprivacy.installer
[5] https://play.google.com/store/apps/details?id=eu.chainfire.supersu
[6] https://www.virustotal.com/

such a site can be a mixed blessing. For new malware it can be a signal to the creator that it was found. Stealth is an important factor in aiding in the spread of a piece of malicious software. As part of pay-per-install schemes, providers ask that their affiliates do not upload their code to popular malware scanners like virustotal. Data submitted to these services is shared with antivirus vendors. [45]

Dynamic or behavioral analysis collects data about a piece of malware as it runs. The malware is ran either inside an emulator or on an actual device. Function calls are traced and mapped out. File (e.g. open, write, and close) and network activity can also be monitored and what is displayed to the user can be observed.

Reverse engineering involves examining the DEX or ART bytecode along with the disassembled instructions. The bytecode can be converted back into a regular java .class file which can be decompiled into java source.

Having been found on one Android malware sandbox, it is unlikely that `AdobeFlashPlayerUpdate.apk` is a piece of new malware so there is little risk in submitting it to virustotal. Table 2 summarizes the permissions requested by the application. Some of these permissions like location and contact information could be reasonable for a normal application, but disabling the lock screen (keyguard), retrieving information about running tasks, and manipulating phone calls are not needed as part of an Adobe Flash update. There is potential for misuse with `SYSTEM_ALERT_WINDOW` as malware can present a fake dialog to the user prompted further downloads. The use of `WAKE_LOCK` keeps the processor and screen of a device active and could negatively impact battery life.

Table 2 AdobeFlashPlayerUpdate.apk requested permissions

ACCESS_COARSE_LOCATION	Allows an app to access approximate location derived from network location sources such as cell towers and Wi-Fi.
ACCESS_FINE_LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
ACCESS_NETWORK_STATE	Allows applications to access information about networks
ACCESS_WIFI_STATE	Allows applications to access information about Wi-Fi networks
DISABLE_KEYGUARD	Allows applications to disable the keyguard
GET_TASKS	Allows an application to get information about the currently or recently running tasks.
INTERNET	Allows applications to open network sockets.
PROCESS_OUTGOING_CALLS	Allows an application to monitor, modify, or abort outgoing calls.
READ_CONTACTS	Allows an application to read the user's contacts data.
READ_PHONE_STATE	Allows read only access to phone state.
SYSTEM_ALERT_WINDOW	Allows an application to open windows using the type TYPE_SYSTEM_ALERT, shown on top of all other applications.
WAKE_LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.

Extracting the certificate information (Table 3) shows that incorrect information was provided to create the certificate that signed the application. No identifying information is provided, and it is clear that this application was not signed by anyone at Adobe.

Table 3 AdobeFlashPlayerUpdate.apk certificate information

```
…/AdobeFlashPlayerUpdate.apk_FILES/META-INF $ keytool -printcert -v -file CERT.RSA
Owner: CN=fp, OU=fp, O=fp, L=fp, ST=fp, C=120
Issuer: CN=fp, OU=fp, O=fp, L=fp, ST=fp, C=120
```

From the manifest file, Table 4 shows an activity and some receivers of interest. The main activity is called when the application is launched. In this manifest handlers are declared to receive the `PHONE_STATE`, `NEW_OUTGOING_CALL`, and `PACKAGE_REMOVED` broadcast messages.

Table 4 AdobeFlashPlayerUpdate.apk activity and receivers of note

```
<activity android:label="@string/app_name"
android:name="com.melina.fp.MainActivity">
 <intent-filter>
  <action android:name="android.intent.action.MAIN"/>
  <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
</activity>

<receiver android:enabled="true" android:exported="true"
android:name="net.mz.callflakessdk.core.ReceiverCall">
 <intent-filter android:priority="999">
  <action android:name="android.intent.action.PHONE_STATE"/>
 </intent-filter>
 <intent-filter android:priority="999">
  <action android:name="android.intent.action.NEW_OUTGOING_CALL"/>
 </intent-filter>
</receiver>

<receiver android:enabled="true" android:exported="true"
android:name="net.mz.callflakessdk.core.ReceiverPackageRemoved">
 <intent-filter android:priority="999">
  <action android:name="android.intent.action.PACKAGE_REMOVED"/>
  <data android:scheme="package"/>
 </intent-filter>
</receiver>
```

Dynamic analysis involves running the sample inside a virtual machine or on actual hardware and observing the application's behavior (screens presented, files accessed, network connections made). If no physical hardware is available, an emulator or online sandboxes like NVISO ApkSan[7] can be utilized to run the application and report back on its behavior. The package was installed to an emulated device, and the icon as seen in Figure 13 was added to the list of installed applications. In all appearances, it does look like a legitimate Adobe application.

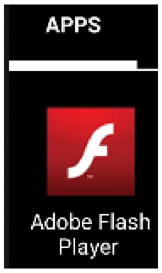
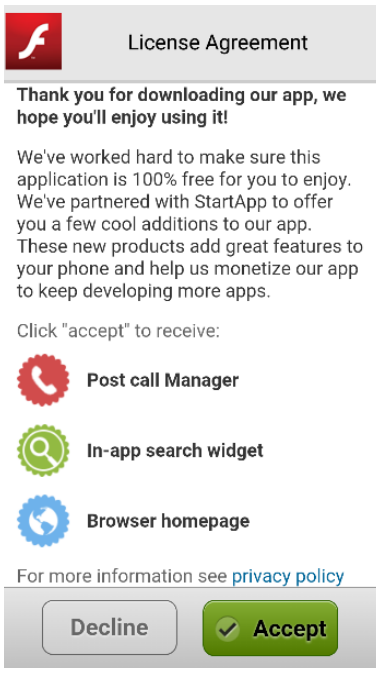


Figure 13 AdobeFlashPlayerUpdate.apk icon

Running the application presents the user with a license agreement as shown in Figure 14. The Adobe Flash icon in the upper left adds a further air of legitimacy

[7] http://apkscan.nviso.be/


License Agreement

Thank you for downloading our app, we hope you'll enjoy using it!

We've worked hard to make sure this application is 100% free for you to enjoy. We've partnered with StartApp to offer you a few cool additions to our app. These new products add great features to your phone and help us monetize our app to keep developing more apps.

Click "accept" to receive:

Post call Manager

In-app search widget

Browser homepage

For more information see privacy policy

Figure 14 License agreement

Selecting Accept displays the ad shown in Figure 15. Trying to exit the application displays more ads similar to those in Figure 16. Declining the license agreement has no effect. A startup ad is still displayed along with ads when the user tries to exit the application.

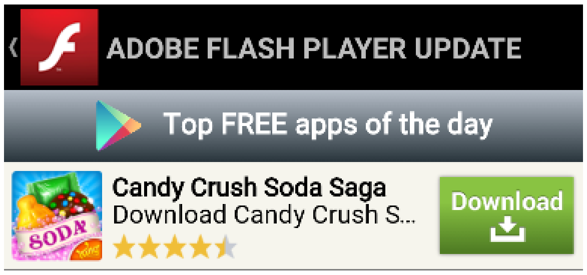


Figure 15 Startup Ad

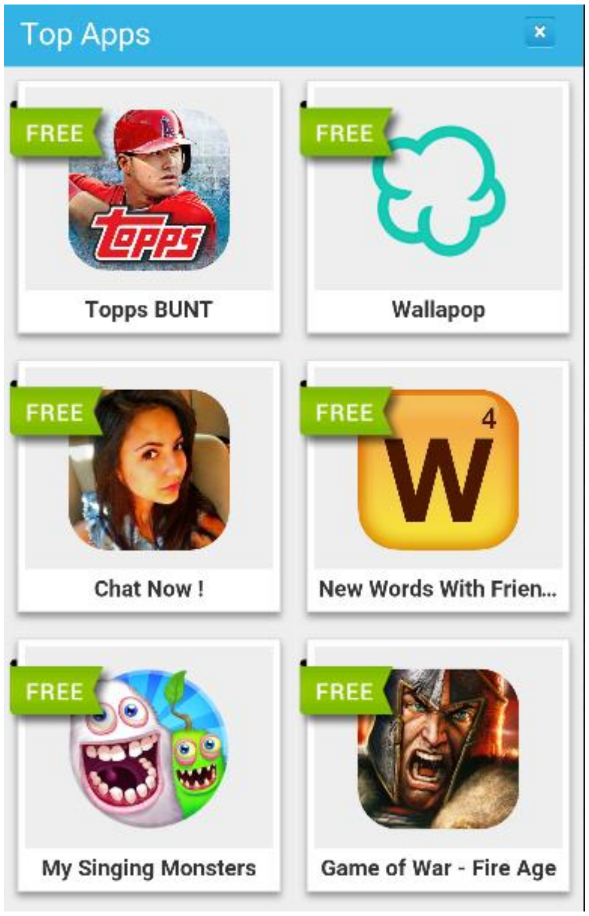


Figure 16 Ads displayed on exit

Reviewing the sandbox analysis shows outbound network connections to IP addresses and ports as shown in Table 5 as well as web connections to the domains listed in Table 6.

Table 5 Network connections made

IP Address	Name Resolution
54.72.14.222:80	ec2-54-72-14-222.eu-west-1.compute.amazonaws.com
54.215.231.168:80	ec2-54-215-231-168.us-west-1.compute.amazonaws.com
46.165.221.16:80	ads.mobilecore.com
23.21.116.57:443	ec2-23-21-116-57.compute-1.amazonaws.com

Table 6 Web domain connections

ads.mobilecore.com
static.mobilecore.com
ads.appia.com
poseidon.mobilecore.com
developer.mobilecore.com

```
www.startappexchange.com
eula.ad-market.mobi
```

None of the IP address or web domains belong to or are associated with Adobe. Reviewing a wireshark trace while running the malware in an emulator does show an HTTP GET request to an Adobe system for the URL `http://helpx.adobe.com/flash-player/kb/archived-flash-player-versions.html`. An Adobe knowledge base document is retrieved to maintain the appearance of legitimacy.

Android applications run in a Java VM that is different from Oracle's VM. After an application is written, the compiled Java bytecode is converted to DEX (or ART for Android 5.0 and beyond) bytecode. Reverse engineering can involve either examining the disassembled byte code or converting it (e.g. `d2j-dex2jar.sh classes.dex`) into a form that can be examined by a Java decompiler. The activities and receivers declared in the manifest provide a starting point in this type of analysis. Like a `main()` method, this application starts by calling `onCreate()` for the `com.melina.fp.MainActivity` activity (Table 7). A connection is initialized and made to Countly[8], a mobile analytics platform, for a specific app key.

Table 7 com.melina.fp.MainActivity::onCreate(Bundle paramBundle) code snippet

```
OnCreate(Bundle paramBundle)
{
   ...
Countly.sharedInstance().init(this, "https://cloud.count.ly",
                              "7f26eaffaf2152e6eca079217142a4432724226a");
MobileCore.init(this, getText(2131034115).toString(),
                MobileCore.LOG_TYPE.DEBUG);
MobileCore.showOfferWall(this, null);
MobileCore.getSlider().setContentViewWithSlider(this, 2130903040);
StartAppAd.init(this, "112063477", "201585241");
StartAppSearch.init(this, "112063477", "201585241");
  ...
}
```

[8] https://count.ly/

`AdobeFlashPlayerUpdate.apk` also makes use of mobileCore[9], a mobile ad network. After that is initialized, it starts to display ads.

7. Conclusion

With titles like *99% of All Mobile Malware Targets Android Devices* and *Malware infected as many Android devices as Windows laptops in 2014* the situation does indeed look grim. Android security has improved over its release cycles and will continue as the malware cycle evolves. Risk cannot be reduced to zero without giving up the conveniences afforded by smartphones. As summarized in Figure 17 and Figure 18 [46], there are many layers of defense between the user and malicious software like `AdobeFlashPlayerUpdate.apk`. The Google Play Store is not perfect, and malware will continue to find a way in. At each layer, it still falls on the user to exercise caution. With each layer bypassed, the risk of malware infecting a device is increased. Android malware cannot function on its own. It needs to gain entry and be installed on to the device. Ultimately, that decision rests with the owner of the device. They are the last line of defense.

[9] https://www.mobilecore.com/

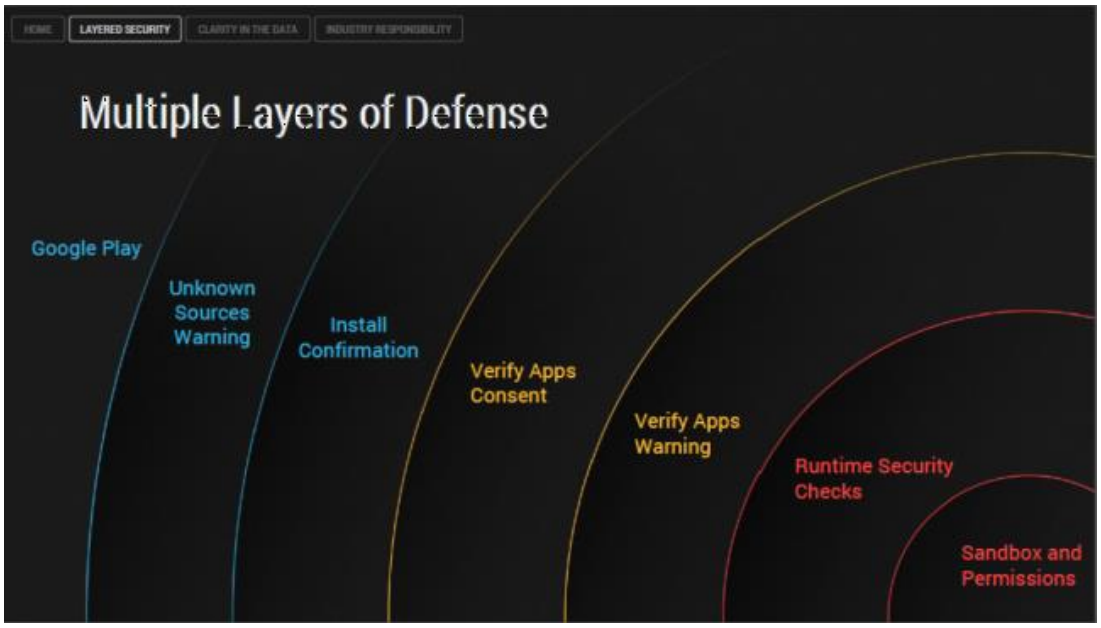


Figure 17 Android layers of defense

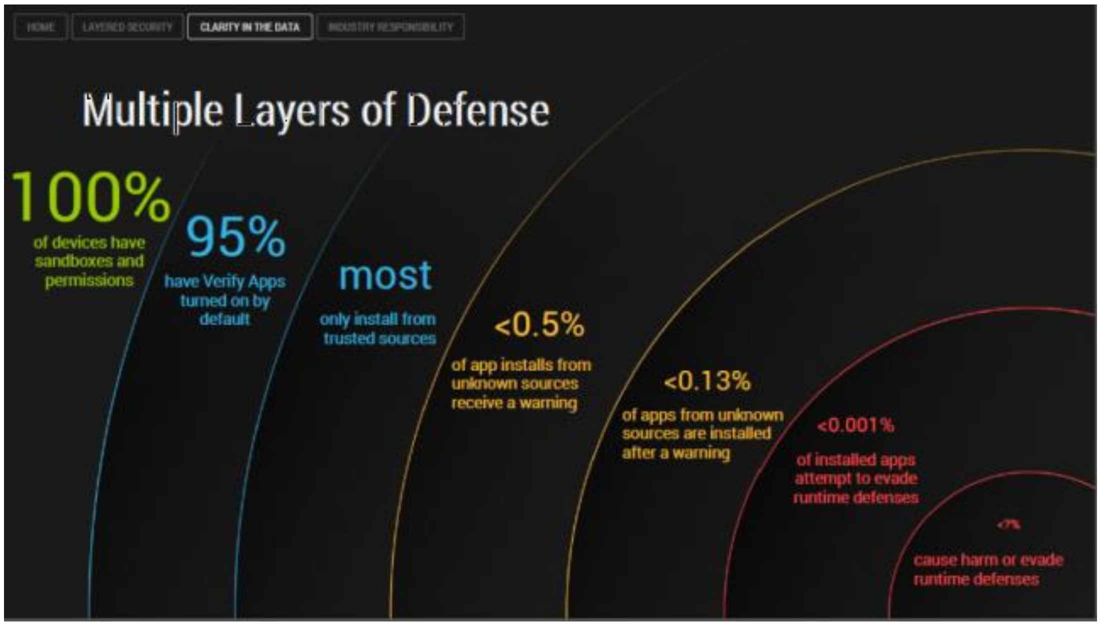


Figure 18 Usage of the layers

References

[1] comScore, Inc., "The_US_Mobile_App_Report.pdf," [Online]. Available: http://mat1.gtimg.com/tech/2014/pdf/The_US_Mobile_App_Report.pdf. [Accessed 31 January 2015].

[2] E. Kaspersky, "10 years since the first smartphone malware – to the day.," 15 June 2014. [Online]. Available: http://eugene.kaspersky.com/2014/06/15/10-years-since-the-first-smartphone-malware-to-the-minute/. [Accessed 31 January 2015].

[3] L. Cavallaro, *Class Lecutre, Topic: "Mobile malware: introduction",* Information Security Group Royal Holloway, University of London, 2013.

[4] A. Nayak, T. Prieto, M. Alsshamlan and K. Yen, "Android Mobile Platform Security and Malware Survey," *International Journal of Research in Engineering and Technology,* vol. 02, no. 11, pp. 764-774, 2013.

[5] K. Dunham, S. Hartman, J. A. Morales, M. Quintans and T. Strazzere, Android Malware and Analysis, Boca Raton: Auerbach Publications, 2014.

[6] S. Khandelwal, "First Android Ransomware that Encrypts SD Card Files," 4 June 2014. [Online]. Available: http://thehackernews.com/2014/06/first-android-ransomware-that-encrypts.html. [Accessed 15 February 2015].

[7] N. Perlroth, "Android Phones Hit by ‘Ransomware’," 22 August 2014. [Online]. Available: http://bits.blogs.nytimes.com/2014/08/22/android-phones-hit-by-ransomware/. [Accessed 13 February 2015].

[8] Vulnerability Research Team (VRT), "Androrat - Android Remote Access Tool," 16 July 2013. [Online]. Available: http://vrt-blog.snort.org/2013/07/androrat-android-remote-access-tool.html. [Accessed 1 March 2015].

[9] Lookout Inc., " November 19, 2014 The new NotCompatible: Sophisticated and evasive threat harbors the potential to compromise enterprise networks," 19 November 2014. [Online]. Available: https://blog.lookout.com/blog/2014/11/19/notcompatible/. [Accessed 1 March 2015].

[10] comScore, Inc., "comScore Reports June 2014 U.S. Smartphone Subscriber Market Share," 6 August 2014. [Online]. Available: http://www.comscore.com/Insights/Market-Rankings/comScore-Reports-June-2014-US-Smartphone-Subscriber-Market-Share?. [Accessed 31 January 2105].

[11] G. Kelly, "Report: 97% Of Mobile Malware Is On Android. This Is The Easy Way You Stay Safe," 24 March 2014. [Online]. Available: http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/. [Accessed 1 February 2015].

[12] F. Van Allen, "99% of All Mobile Malware Targets Android Devices," 22 January 2014. [Online]. Available: http://www.techlicious.com/blog/99-of-all-mobile-malware-targets-android-devices/. [Accessed 1 February 2015].

[13] Google, "The Android Source Code," [Online]. Available: https://source.android.com/source/index.html. [Accessed 1 March 2015].

[14] Open Handset Alliance, "Android Overview," [Online]. Available: http://www.openhandsetalliance.com/android_overview.html. [Accessed 1 March 2015].

[15] Motorola Solutions, "Google Mobile Services," 21 May 2014. [Online]. Available: https://developer.motorolasolutions.com/servlet/JiveServlet/previewBody/2210-102-2-2535/Google%20Mobile%20Services%20vs%20AOSP.pdf. [Accessed 1 March 2015].

[16] C. Arthur and S. Gibbs, "The hidden costs of building an Android device," 23 January 2014. [Online]. Available: http://www.theguardian.com/technology/2014/jan/23/how-google-controls-androids-open-source. [Accessed 22 April 2015].

[17] J. Kahn, "Google: We do not charge licensing fees for Android's Google Mobile Services," 23 January 2014. [Online]. Available: http://9to5google.com/2014/01/23/google-we-do-not-charge-licensing-fees-for-androids-google-mobile-services/. [Accessed 22 April 2015].

[18] N. Elenkov, Android Security Internals, San Francisco: No Starch Press, Inc., 2014.

[19] Google, "Building and Running Overview," [Online]. Available: https://developer.android.com/tools/building/index.html. [Accessed 1 March 2015].

[20] Google, "Signing Your Applications," [Online]. Available: https://developer.android.com/tools/publishing/app-signing.html. [Accessed 23 March 2015].

[21] W. Verduzco, "Application Signature Verification: How It Works, How to Disable It with Xposed, and Why You Shouldn't," 16 June 2014. [Online]. Available: http://www.xda-developers.com/application-signature-verification-how-it-works-how-to-disable-it-with-xposed-and-why-you-shouldnt/. [Accessed 20 April 2015].

[22] Google, "System Permissions," [Online]. Available: https://developer.android.com/guide/topics/security/permissions.html. [Accessed 1 March 2015].

[23] Google, "Security Tips: Using Permissions," [Online]. Available: https://developer.android.com/training/articles/security-tips.html#Permissions. [Accessed 24 March 2015].

[24] A. P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, "Android Permissions Demystified," October 2011. [Online]. Available: http://www.cs.berkeley.edu/~dawnsong/papers/2011%20Android%20permissions%20demystified.pdf. [Accessed 24 March 2015].

[25] Google, "Intents and Intent Filters," [Online]. Available: https://developer.android.com/guide/components/intents-filters.html. [Accessed 11 April 2015].

[26] S. Hill, "Do you need antivirus on Android? We asked an expert," 15 February 2015. [Online]. Available: http://www.digitaltrends.com/mobile/do-you-need-antivirus-on-android/. [Accessed 15 March 2015].

[27] A. Apvrille, "iOS Malware Does Exist," 9 June 2014. [Online]. Available: https://blog.fortinet.com/post/ios-malware-does-exist. [Accessed 1 April 2015].

[28] L. Sun, B. Hong and F. Hacquebord, "Pawn Storm Update: iOS Espionage App Found," 15 February 2015. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/pawn-storm-update-ios-espionage-app-found/. [Accessed 20 March 2015].

[29] S. Fadilpašić, "Beware, iOS malware is on the horizon," 5 March 2015. [Online]. Available: http://www.itproportal.com/2015/03/05/beware-ios-malware-horizon/. [Accessed 1 April 2015].

[30] Symantec, "Windows Malware Attempts to Infect Android Devices," 23 January 2014. [Online]. Available: http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices. [Accessed 15 March 2015].

[31] Google, "Android Debug Bridge," [Online]. Available: https://developer.android.com/tools/help/adb.html. [Accessed 24 March 2015].

[32] HTC, "The Anatomy of an Android OS Update," 2013. [Online]. Available: http://www.htc.com/us/go/htc-software-updates-process/. [Accessed 15 February 2015].

[33] B. Rose, "Why Android Updates Are So Slow," 19 March 2013. [Online]. Available: http://gizmodo.com/5987508/why-android-updates-are-so-slow. [Accessed 15 February 2015].

[34] S. Rosenblatt, "Google leaves most Android users exposed to hackers," 24 January 2015. [Online]. Available: http://www.cnet.com/news/google-leaves-most-android-users-exposed-to-hackers/. [Accessed 12 February 2015].

[35] Google, "Dashboards: Platform Versions," [Online]. Available: https://developer.android.com/about/dashboards/index.html#Platform. [Accessed 12 February 2015].

[36] "Google Search: android antivirus," [Online]. Available: https://www.google.com/#q=android+antivirus. [Accessed 15 March 2015].

[37] Tom's Guide Staff, "Best Antivirus Software and Apps 2015," 4 February 2015. [Online]. Available: http://www.tomsguide.com/us/best-antivirus,review-2588-7.html. [Accessed 15 March 2015].

[38] J. Hindy, "15 best antivirus Android apps and anti-malware Android apps," 6 January 2015. [Online]. Available: http://www.androidauthority.com/best-antivirus-android-apps-269696/. [Accessed 15 March 2015].

[39] C. DiBona, 16 November 2011. [Online]. Available: https://plus.google.com/+cdibona/posts/ZqPvFwdDLPv. [Accessed 15 March 2015].

[40] R. Fedler, J. Schütte and M. Kullicke, "On the Effectiveness of Malware Protection on Android An evalution of Android antivirus apps," Faunhofer Research Institution Fore Applied And Integrated Security, 2013.

[41] H. Lockheimer, "Android and Security," 2 February 2012. [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html. [Accessed 19 April 2015].

[42] O. Hou, "A Look at Google Bouncer," 20 July 2012. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/. [Accessed 19 April 2015].

[43] R. Farmer, "A Brief Guide to Android Security," Acumin Consulting, 2011.

[44] "Index of /samples," 6 January 2015. [Online]. Available: http://androidsandbox.net/samples/. [Accessed 12 February 2015].

[45] L. Cavallaro, *Class Lecutre, Topic: "Specialized cybercrime: Pay-per-Install",* Information Security Group Royal Holloway, University of London, 2013.

[46] B. Petrovan, "Google data shows threat of Android malware is massively overblown," 3 October 2013. [Online]. Available: http://www.androidauthority.com/android-malware-threat-report-279702/. [Accessed 25 April 2015].