Fraunhofer
AISEC

FRAUNHOFER RESEARCH INSTITUTION FOR
APPLIED AND INTEGRATED SECURITY

ANDROID OS SECURITY: RISKS AND LIMITATIONS

A PRACTICAL EVALUATION

RAFAEL FEDLER, CHRISTIAN BANSE, CHRISTOPH KRAUSS, AND VOLKER FUSENIG

5/2012

Fraunhofer AISEC

Android OS Security: Risks and Limitations

A Practical Evaluation

Version 1.0

Rafael Fedler, Christian Banse, Christoph Krauß, and Volker Fusenig

Contact: *volker.fusenig@aisec.fraunhofer.de*

AISEC Technical Reports
AISEC-TR-2012-001
May 2012

Fraunhofer Research Institution AISEC
Parkring 4
85748 Garching

Abstract

The number of Android-based smartphones is growing rapidly. They are increasingly used for security-critical private and business applications, such as online banking or to access corporate networks. This makes them a very valuable target for an adversary. Up to date, significant or large-scale attacks have failed, but attacks are becoming more sophisticated and successful. Thus, security is of paramount importance for both private and corporate users. In this paper, we give an overview of the current state of the art of Android security and present our extensible automated exploit execution framework. First, we provide a summary of the Android platform, current attack techniques, and publicly known exploits. Then, we introduce our extensible exploit execution framework which is capable of performing automated vulnerability tests of Android smartphones. It incorporates currently known exploits, but can be easily extended to integrate future exploits. Finally, we discuss how malware can propagate to Android smartphones today and in the future, and which possible threats arise. For example, device-to-device infections are possible if physical access is given.

NES Research Department at Fraunhofer AISEC

The Fraunhofer Research Institution for Applied and Integrated Security AISEC[1] is one leading expert for applied IT security and develops solutions for immediate use, tailored to the customer's needs. Over 80 highly qualified employees covering all areas of IT security make such customized services possible. Fraunhofer AISEC is organized into three different research and development divisions. They focus on hardware security as well as the protection of complex services and networks. Clients of Fraunhofer AISEC operate in a variety of industrial sectors, such as the chip card industry, telecommunications, the automotive industry, and mechanical engineering, as well as the software and healthcare industries. Fraunhofer AISEC was founded in 2009 as an independent research organization within the Fraunhofer-Gesellschaft.

[1] http://www.aisec.fraunhofer.de/

Contents

1 Introduction

During recent years, the share of smartphones in overall handheld mobile communication device sales has drastically increased. Among them, the Android operating system by the Open Handset Alliance, prominently led by Google Inc., is market dominating. In Q3 2011, 52.5% of all devices sold were Android devices, followed by Symbian (16.9%) and Apple's iOS (15.0%), according to Gartner analysis [1].

With the widespread use of smartphones both in private and work-related areas, securing these devices has become of paramount importance. Owners use their smartphones to perform tasks ranging from everyday communication with friends and family to the management of banking accounts and accessing sensitive work-related data. These factors, combined with limitations in administrative device control through owners and security-critical applications like the Mobile TAN for banking transactions, make Android-based smartphones a very attractive target for attackers and malware authors of any kind and motivation. Up until recently, the Android Operating System's security model has succeeded in preventing any significant attacks by malware. This can be attributed to a lack of attack vectors which could be used for self-spreading infections and low sophistication of malicious applications.

However, emerging malware deploys advanced attacks on operating system components to assume full device control. We developed an extensible exploit execution framework to test existing and future exploits in a controlled environment. This framework can also serve to analyze exploitability of devices with specific test sets and payloads. Additionally, common malware behavior is emulated, such as dynamic configuration and exploit download from a remote web server.

This paper gives a short, yet comprehensive overview of the major Android security mechanisms. It also discusses possibilities to successfully infiltrate Android-based smartphones through recent attack means. Both pre-infection (propagation) and post-infection (threat) scenarios will be illuminated. We will also take a look at the implications by current developments for future propagation mechanisms. To provide theoretical background, we give a short explanation of fundamental Android security measures in Chapter 2. After outlining attack vectors, preconditions and state-of-the-art public exploits in Chapter 3, we introduce our exploit execution framework in Section 3.6. Subsequently, we elaborate on existing and prospective possibilities to successfully infect or infiltrate Android-based smartphones through recent attack means in Chapter 4. Chapter 5 provides an overview of current and emerging threat scenarios on smartphones both for private and corporate targets. Chapter 6 concludes the paper with a summary and an advisory.

2 Android and Android Security

Mobile operating systems pre-installed on all currently sold smartphones need to meet different criteria than desktop and server operating systems, both in functionality and security. Mobile platforms often contain strongly interconnected, small and less-well controlled applications from various single developers, whereas desktop and server platforms obtain largely independent software from trusted sources. Also, users typically have full access to administrative functions on non-mobile platforms. Mobile platforms, however, restrict administrative control through users. As a consequence, different approaches are deployed by the Android platform to maintain security.

This chapter briefly introduces the Android platform and its major security measures, also giving an overview on the measures' limits, weaknesses and even exploitability. Discussion will be limited to those functions related to threats elaborated on in this paper.

2.1 Android Platform

The Android operating system is illustrated in Figure 2.1. Apps for Android are developed in Java and executed in a virtual machine, called Dalvik VM. They are supported by the application framework, which provides frequently used functionality through a unified interface. Various libraries enable apps to implement graphics, encrypted communication or databases easily. The Standard Library ("bionic") is a BSD-derived libc for embedded devices. The respective Android releases' kernels are stripped down from Linux 2.6 versions. Basic services such as memory, process and user management are all provided by the Linux kernel in a mostly unmodified form.

However, for several Android versions, the deployed kernel's version was already out of date at the time of release. This has lead to a strong increase in vulnerability, as exploits were long publicly available before the respective Android version's release.

Detailed information on all security features can be retrieved from the Android Security Overview [2] on the official Android Open Source Project website.

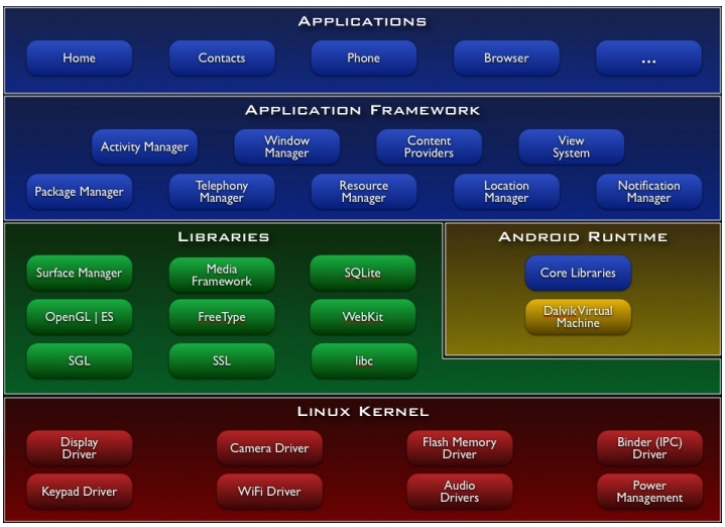

Figure 2.1: Android System Architecture [3]

2.2 File System and User/Group Permissions

As in any Unix/Linux-like operating system, basic access control is implemented through a three-class permission model. It distinguishes between the *owner* of a file system resource, the owner's *group* and *others*. For each of these three entities, distinct permissions can be set to read, write or execute. This system provides a means of controlling access to files and resources. For example, only a file's owner may write (alter) a document, while members of the owner's group may read it and others may not even view it at all.

In traditional desktop and server environments, many processes often share the same group or even user ID (namely the user ID of the user who started a program). As a result, they are granted access to all files belonging to the other programs started by that same ID. In traditional environments with largely trusted software sources this may suffice, though Mandatory Access Control approaches trying to establish a more fine-grained model do exist, such as AppArmor and SELinux [4].

However, for mobile operating systems this is not sufficient. Finer permission distinction is needed, as an open app market is not a strongly monitored and trustworthy software source. With the traditional approach, any app executed under the device owner's user ID would be able to access any other app's data.

Hence, the Android kernel assigns each app its own user ID on installation. This ensures that an app can only access its own files, the temporary directory and nothing else – system resources are available through API calls. This establishes a permission-based file system sandbox.

2.3 Android API Permission Model and Manifest File

On installation, the user is presented with a dialog listing all permissions requested by the app to be installed. These permission requests are defined in the Manifest File `AndroidManifest.xml`, which is obligatory for shipping with every Android app.

However, this system has a few flaws:

- *All or none-policy*: A user cannot decide to grant single permissions, while denying others. Many users, although an app might request a suspicious permission among many seemingly legitimate permissions, will still confirm the installation.
- Often, users cannot judge the appropriateness of permissions for the app in question. In some cases it may be obvious, for example when a game app requests the privilege to reboot the smartphone or to send text messages. In many cases, however, users will simply be incapable of assessing permission appropriateness.
- Circumvention: Functionality, which is supposed to be executable only given the appropriate permissions, can still be accessed with fewer permissions or even with none at all. This point will be explained in detail in the chapter on attack vectors.

2.4 Android Market ("Google Play")

As anyone can publish an app after registration as a developer for USD25 and due to its availability to all Android users, the Android Market, recently renamed "Google Play", was and is the main channel of malware distribution.

The majority of all infections is conducted through free illegitimate copies of paid content. Users unwilling to pay for such content turn to pirated copies, which are often altered to deliver malicious code. This process, known as "repackaging", is illustrated in Figure 2.2. An Android botnet uncovered recently employed this technique to infect several ten thousand devices, generating a revenue of multiple million dollars annually through premium communication services [5].

As became recently known, Google tests apps for possibly malicious behavior through a service called "Bouncer" [6]. Bouncer examines apps submitted to the Android Market automatically by execution inside a virtual Android environment in Google's cloud infrastructure. In case of malicious code detection, manual analysis is performed to prevent false alarms.
Although malware download numbers decreased since the installation of Bouncer, this system does not provide security against modern attack approaches. Dynamically provided exploits which were not initially shipped with the original app will

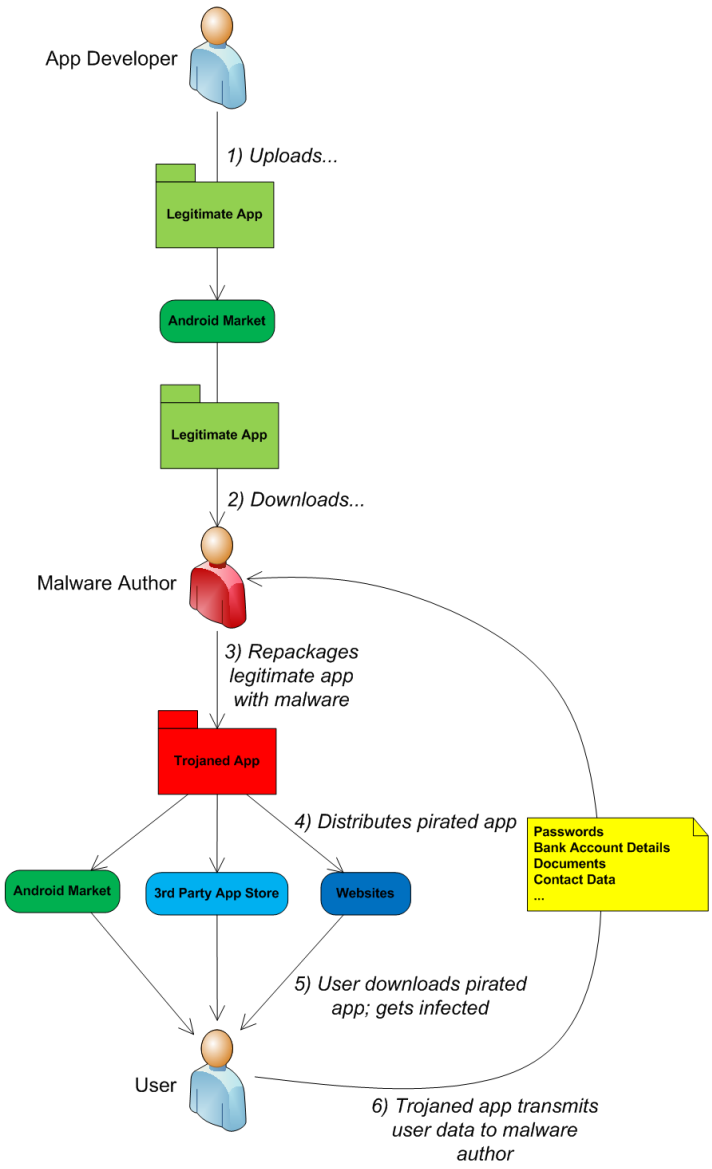

Figure 2.2: Repackaging Process

successfully avoid detection, especially if only downloaded at the malware author's command. For this objective, malware authors may delay downloading of exploits until publishing to the Android Market is completed. It is also questionable whether Bouncer is able to identify unknown or altered exploits.
In conclusion, Bouncer may aid against obviously malicious app. This includes apps which do not rely on dynamic and platform-specific privilege escalation exploits or implement all spying or remote control functionality in Java. Against more sophisticated attacks, Bouncer does not prove effective.

2.5 Remote Installation and Uninstallation

Google possesses the ability to remotely remove or install any app from/to all Android devices. It is used for the removal of malicious applications and the addition of new services to phones which may require installation of new apps.

This is accomplished through the Android Market protocol's `REMOVE_ASSET` and `INSTALL_ASSET` commands. The latter is in fact used for normal installation of app package (APK) files through the Android Market as well. Functionality is provided by the GTalk service which is always active on an Android device and listening for protocol commands. When installing an app from the Android market, the "Install" button triggers Google servers to push an `INSTALL_ASSET` command to the phone, forcing it to download and install the specified app [7, 8].

However, man-in-the-middle attacks between an Android device and Google servers may grant an attacker the possibility to push malware to an Android device. These attacks may be accomplished through different methods, such as successful compromisation and manipulation of SSL-secured communication of end-user devices with Google servers. Rogue wireless access points may impersonate Google servers or redirect traffic destined for Google to servers under an attacker's control.

2.6 Patch Process

The Android OS patch process is a complex and time-consuming combined effort with many parties involved. After a vulnerability has been discovered in an Android OS component, Google has to develop and push out a patch. This patch is merged with the Android OS source code, which is then provided to device manufacturers. They adjust the new code base to their devices, affecting not only original equipment, but also handsets specifically produced for carriers or geographical regions. This may include branding, software-enforced usage limitations (e.g., prohibition of tethered network access) or even special hardware features. Finally, a system image ("ROM") has to be generated and distributed over the air to customer devices for installation. Erroneous installation images may immediately render customer devices unusable.

The multiplicity of existing devices and the need for adjustments for carriers and regions make the Android patch process a slow matter. It requires strong precision and much manpower and finances. Until a security patch reaches customers, weeks or even months may pass. Due to the aforementioned reasons, many manufacturers even decide to not provide older devices with updates at all. Thus, many devices remain unpatched for a long time, or even forever. Even though newer devices may not be vulnerable to attacks, many older devices continue to be in use by customers and can be compromised easily [9].

2.7 SEAndroid

The United States' National Security Agency has recently announced the commencement of the SEAndroid (Security Enhanced Android) project as an addition to the Android kernel [10]. Similarly to the well-known and widely deployed SELinux Linux kernel patch, the SEAndroid project aims to establish a fine-grained Mandatory Access Control model. It is further adjusted and extended to meet the requirements which arise on the Android platform, e.g., to secure inter-process communication [11].

Once integrated into Android, SEAndroid may indeed prevent some of the attacks presented in Chapter 4. The most useful feature is the monitoring of executability of binaries in the root file system. If execution of non-system binaries or of files saved after the installation of any specific app is prohibited, any currently known root exploit would be successfully prevented. Thus, exploitability of Android devices would be reduced significantly. This has already been proven to prevent many publicly known exploits [12]. Other means of privilege escalation and data leakage, such as unwanted inter-process communication, can also be prevented.

SEAndroid is still in a very early development stage. It is unknown when or if SEAndroid will be integrated into the default code base of the operating system.

2.8 Apps and Native Code

Android apps are developed in Java and executed inside the Dalvik Virtual Machine, a register-based Java Virtual Machine. Each app receives its own Dalvik VM instance and user ID, hence separating process memory and files through means of the Linux kernel. Apps authored by the same developer may share the same user ID, if chosen by the developer. System resources may only be accessed by apps through API methods provided by the Dalvik VM.

These limitations make attacks on the underlying operating system from Android apps nearly impossible. However, native ARM architecture code can be compiled for Android devices; both the Linux kernel and essential libraries are implemented in C (cf. Figure 2.1). Native code may even be called from within Android apps, be it for user-desired or malicious usage. As native code is executed outside the Dalvik VM, it is not limited to API methods, thus receiving more direct and uncontrolled access to the operating system core.

3 Exploitability and Attack Vectors

Multiple security holes have been found in different components of the Android operating system. Up until now, they have primarily served to grant device owners administrative privileges on their devices ("rooting"). Only recently malware authors have begun utilizing such holes and publicly available exploits for malicious code. In this chapter, we introduce those exploits and further attack vectors. An assessment of malware utility of these exploits was performed using our exploit execution framework, presented in Section 3.6.

3.1 Native Executable Control

All current exploit-based attacks depend on the ability to execute native code. It is the most vital condition for attacks which let an attacker assume full control over a device. As native code is not executed inside the Dalvik VM, it poses a much higher risk to the Android OS core than usual apps.

Restriction of the privilege to set the executable bit for files, e.g., to the root user and the Market app, would significantly reduce exploitability of Android devices. As a result, apps would no longer be able to download and execute native binaries dynamically. However, setting the executable bit through user-mode options such as `chmod` is currently not monitored or controlled at all. SEAndroid, although in an early development stage, has demonstrated the high effectiveness of this measure (cf. Section 2.7). As integration of SEAndroid into the default Android code base is uncertain and not to be expected soon, native code exploits will remain the biggest attack vector.

3.2 Public Exploits

In the following, an overview of publicly known exploits will be given, including their usability for malicious intent and target operating system versions. These exploits have been tested in practice through our malware execution framework, explained in detail in Section 3.6.

Since some exploits depend on permissions available only to the shell user, these cannot be utilized for in-app privilege escalation to gain root access. Up to date, they are only used for granting users full administrative access ("rooting"). However, as these exploits can be executed by the shell user via the Android Debug Bridge, they can be used for device-to-device or PC-based infection. The following table names all publicly known exploits, their Common Vulnerabilities and

Exposures (CVE) number and affected Android versions. Basic behavior is described and usability for in-app exploitation, i.e. without USB access, is denoted.

Vulnerable Android Version	Name	Description	CVE
< 2.2	Use-After-Free Webkit Vulnerability	Remote arbitrary code execution through NaN handling due to lacking floating point number validation	CVE-2010-1807
2.1, 2.2, 2.3	Focus Stealing Vulnerability	Any app may steal another app's focus and display its own Activity windows above the other app's, effectively making it impossible for a user to determine that this Activity origins from another app. This allows for displaying login dialogs for stealing user credentials.	TWSL2011-008
$\leq$ 2.1	Exploid	Hotplug Exploit, usable for malware	CVE-2009-1185
2.2.0, some 2.2.1/2.2.2	RageAgainstTheCage, Zimperlich	RLIMIT_NPROC exhaustion, causing dropping privileges through suid() to fail because suid() cannot be called anymore – Zimperlich usable for malware	"CVE-2010-EASY"[1]
$\leq$ 2.2.3, 2.3.[0–3], 3.0	GingerBreak	mPartMinors[] (NPARTS) out of bounds write – can be used for malware	CVE-2011-1823
$\leq$ 2.2.2	psneuter, KillingInTheNameOf	Neuters the Android property Service, which is checked by adb when starting up to determine whether it should run as root or with low privileges. Variant "KillingMeSoftly" usable for malware.	CVE-2011-1149
2.2, 2.3	ZergRush	libsysutils root exploit use-after-free, not usable for in-app malware	CVE-2011-3874
4.0.[0–3]	mempodroid	Unauthorized write access to other processes' memory; not usable for in-app malware: adb access required for exploitation	CVE-2012-0056

[1] no official CVE

3.3 Android Permission Model

As shown in [13], the permissions formally requested by an app at installation time through its manifest file do not have to match those actually made use of. Hence, it is completely impossible for a user to determine what activities an app performs with the permissions it has requested.

There are several examples of actions which can be performed without the proper permissions or without any permissions at all. Some of these actions are particularly valuable for malware purposes:

- `RECEIVE_BOOT_COMPLETED`: This is a permission enabling an app to start at boot – malware may start automatically and run unnoticed.
- `START_ON_INSTALL`: Enables an app to start up automatically right after installation.

These two permissions – especially when not formally requested and presented to the user – significantly simplify malware infection and camouflage. Any app on the Android market may serve as a downloader and installer for any other malicious app. This functionality requires only very few lines of code. The payload can easily be camouflaged as temporary data or even hidden inside seemingly legitimate program data. For example, a binary or APK file can be stored behind an image file's actual data section.

After installation, the two aforementioned capabilities suffice to immediately create a service which will continue to run until removed, if detected at all. This is possible although the permissions `RECEIVE_BOOT_COMPLETED` and `START_ON_INSTALL` had not been requested. More permissions can be made use of as well without a formal request, such as uploading or downloading data from the Internet by pointing the default browser to certain URLs.

Furthermore, the Android logging service has proven very effective for access to all kinds of data. The `READ_LOGS` permission can substitute the following on many devices, depending on the Android version and thus the standard apps' versions installed:

- `READ_CONTACTS`
- `GET_TASKS` – every started Activity[1] is listed in the system's logs
- `READ_HISTORY_BOOKMARKS` – opening new web pages is a browser Activity and thus logged
- `READ_SMS`

[1] An Activity is an Android app component displaying visible information or an interactive dialog to the user.

This does not work on most current Android versions anymore. However, many apps still log much more data than actually needed, including invoked URLs, message bodies, authentication data, geographical coordinates and much more. In general, users should be very cautious when apps request `READ_LOGS` permission on installation to prevent data leakage. Most apps – except for log viewers – do not need this permission.

A special instance of permission model shortcomings is introduced in the following section.

3.4 Zero Permission URI Handler Remote Shell

An approach found very recently [14] does not require any exploit code to be executed on a target device. Instead, it combines Android API features cleverly to install a remote shell on an Android device. It builds on the aforementioned inefficiencies of the permissions model.

Communication from an Android device to a host acting as command-and-control server is accomplished through a web browser. Any Android app may, without any formal API privileges required, launch the default web browser and direct it to any URL through means of an Intent[2]. URLs, however, may also contain GET-parameters, which can be used to transmit data from an Android device to a web server. Issuing such web browser commands can be limited to a time when the phone is not in use and its display is turned off, thus hiding activity from the user. This is achieved by polling the power manager system service for the display's status through `isScreenOn()` and requires no privileges as well.

If the permission to access the Internet has been granted to an app on installation, it does not need to wait for the display to be turned off at all. Instead, it can retrieve a URL invisibly in the background. Communication from the command-and-control-server to the Android device can be implemented with zero privileges too. This can be achieved by registering a URI handler such as "test://" or "mymal://" in the app's manifest file. Upon registration, every URI prepended with such a protocol specifier will be handed over to the app it is registered to.

To achieve two-way communication, the command-and-control server's web page, which is called by an Android device, will contain a redirect instruction to a URI of the described format. Upon sending the browser to this URI automatically, it gets passed on to the malicious app. Thus, a full remote shell can be implemented without any privileges required. It can easily be hidden inside a legitimate app or installed as a standalone app without device owners noticing.

[2] Android inter-process communication message

3.5 Google Services Authentication Tokens

Researchers from the University of Ulm have discovered a data leakage-related attack vector of very high relevance for all users of Android versions prior to 2.3.4. It has been found that the default Google "Calendar Sync" and "Contacts Sync" apps transmit ClientLogin authentication tokens unencrypted [15]. These authentication tokens can be used for logging into any Google service under the authentication owner's identity and are valid for several days. The authentication tokens captured during the researchers' tests were in fact valid for 14 days.

Rogue wireless network access points can easily eavesdrop for such authentication tokens to impersonate the original owner. Subsequently, all data stored on Google servers can be obtained, including contact details such as phone numbers, home and email addresses, calendar data and so on.
Google has provided patches for all versions of the aforementioned standard apps. However, third-party apps can still authenticate via unencrypted ClientLogin, leaving their users prone to a loss of all information stored on Google servers.

3.6 Exploit Execution Framework

To test functionality of current and future exploits in a controlled environment, we developed an extensible client-side framework for exploit execution and privilege escalation. With our framework, we are also able to analyze exploitability of devices with specific test sets and payloads in a semi-automatic manner. The framework is similar to Jon Oberheide's RootStrap proof of concept-app [16] and to real-world malware such as GingerMaster [17, 18].

Its execution is a multi-staged process, whose general concept is illustrated in Figure 3.1. On startup, a configuration file is fetched from a remote server. The framework app extracts information from the configuration file on which exploit to execute to escalate its privileges and which payload to execute afterward. This way, easy extensibility of the framework is achieved: New exploits or differing settings only have to be configured in this file. No source code alterations have to be performed, as program behavior is dynamically determined through the configuration file. Thus, our framework can be distributed to many different devices and executed simultaneously.

After the configuration file has been loaded and parsed, an exploit binary suitable for the device will be downloaded and executed. Depending on a device's version, different exploits are used to gain root privileges.

Upon successful completion, the exploit is supplied with a payload to execute with root privileges. This payload is again determined by the configuration file and may be a shell script, a binary or an app installation file.

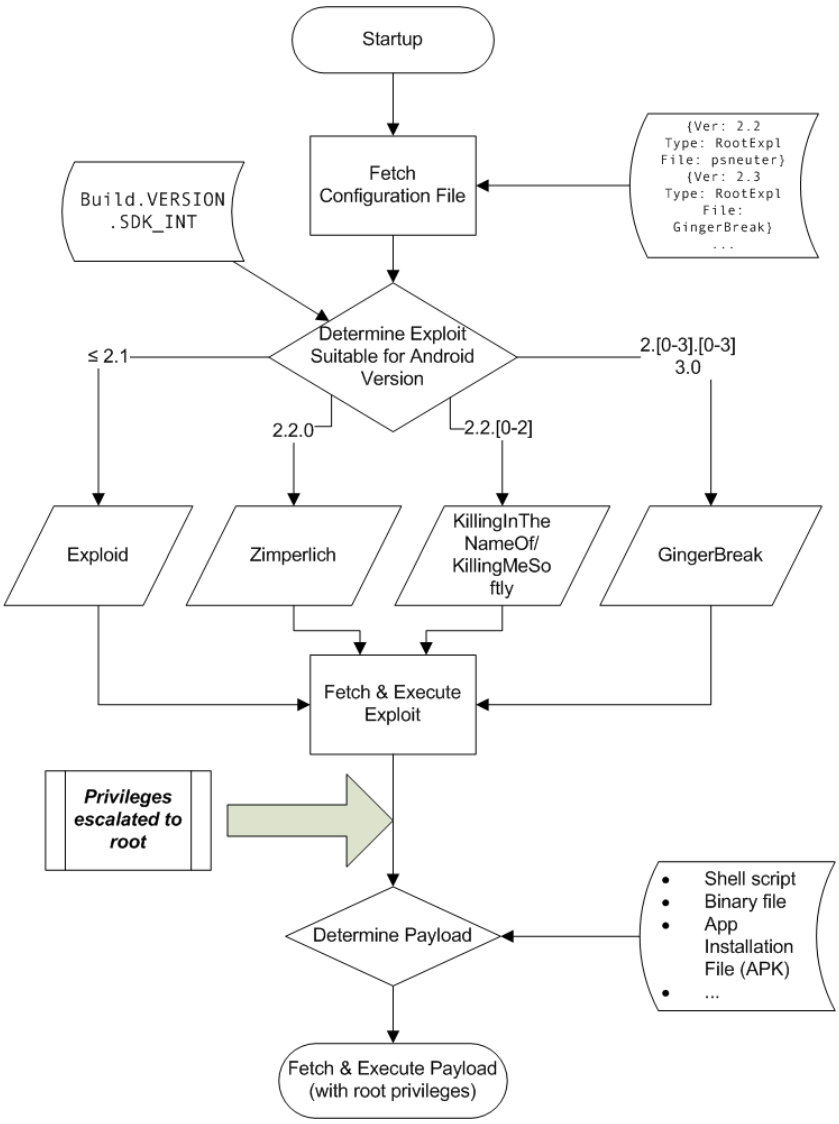

Figure 3.1: Exploit Execution Framework Lineout

Through its dynamic exploit download and privilege escalation behavior, which is also deployed in real-world malware, recognition techniques such as Bouncer and mobile signature-based antivirus software can be evaded. To avoid recognition before publishing to the Android Market, exploits will only be supplied once our framework, or similarly malware, has passed testing and is released to users. For greater effect, a malware author may choose to distribute exploits to devices only once a sufficiently big user base has been established. Recognition by mobile signature-based antivirus software can be avoided by altering or obfuscating exploit binaries.

Our framework app could easily be integrated into, or disguised as, another app. Naturally, a malware developer will provide decoy functionality with his app to avoid detection. Exploit downloading and execution would only be triggered once app installation numbers meet the malware author's expectations.

Thus, our framework emulates the behavior of real-world malware and demonstrates how detection can be avoided. Through its configurability and extensibility, it can be used to easily assess the vulnerability of multiple devices to different exploits. As has also been stated, current and future exploits can be tested in a controlled environment.

Currently, we are developing an additional framework component enabling Android devices to infect other Android devices via USB.

Figure 3.2: Proof of Concept Malware Debug Screen Capture

4 Propagation Scenarios

For mobile malware, current propagation scenarios significantly differ from those of desktop malware. Direct self-spreading mechanisms over primary communication networks known from desktop environments are very unlikely. However, different approaches exist, which utilize existing infrastructure such as the Android Market and websites. Also, novel approaches such as PC-to-device and device-to-device infections will be discussed.

4.1 Malicious Apps

Malicious apps are the most common infection channel and are comparable to trojanized programs on desktop platforms. They provide high convenience for malware developers, as the Android Market and third-party app markets potentially give access to all Android users. Malicious code can be packaged and redistributed with popular apps. Furthermore, users can choose to allow installation from websites, which can also be exploited by attackers.

App Markets

Some paid content on the Android market is particularly popular with huge portions of Android users. Thus, a way often taken by malware authors to spread their malicious apps is the provision of free illegitimate copies of paid content, infected with the malware author's malicious code [19]. Users unwilling to pay for such content turn to these pirated copies and in turn get infected, accounting for the majority of all malware infections on Android devices by far.

Free content may contain malicious or spying code as well. When providing functionality popular enough, it is a convenient and safer way to widely spread malware, as it will not be removed for piracy reasons from the official app market. Additionally, developers with malicious intents may also develop apps free of any malicious code at the time of the app's release. Once a big and trusting user base has been established, an automatic update can be pushed out. It will contain malicious code portions and immediately infect every phone with the formerly non-malicious app installed. Even when a developer has no malicious intentions, an attacker may specifically try to overtake the Market accounts of popular apps' developers. The attacker will then push out malware to the original developers' user base.

An app containing malware will remain in the market until its malicious code portions are noticed. Typical end-users are very unlikely to identify malicious apps.

This especially applies when decoy features are functioning properly, successfully averting any suspicion. Also, as pointed out in Section 2.4, Google's malware recognition can be circumvented. This has been proven by the GingerMaster malware, which was admitted to the Android Market after deployment of the Bouncer service. It also applies for other malware which dynamically downloads its privilege escalation code only after admission to the app market.

Third-party app markets installed on modified operating system distributions ("custom ROMs") are particularly endangered of containing malicious apps. Pirated versions of paid content are considered a feature by many users of such markets and are thus not removed by the market operators on purpose. Also, third-party app markets, especially if operated by a non-professional community, simply do not have the personnel to closely monitor app uploads. Malware recognition techniques such as Google's Bouncer are not deployed. Furthermore, these community-run app markets do not require a registration fee as opposed to USD25 of the official market. This should not be underestimated, as it gives the possibility of easily setting up developer accounts with those markets automatically and free of charge. In the official app market, a malware author would be charged every time one of their accounts is removed for spreading malware.

Websites

Device owners can configure their devices to allow website sources for app installation. Those underlie no restriction or monitoring at all, increasing the risk of installing trojanized apps. Also, when this option is activated, users can be redirected to fake websites supposedly supplying a "critical update". Based on the user agent identification string of a device's browser, targeted attacks against vulnerable smartphones can be conducted. Rogue networks or attackers may even be able to rewrite web traffic to replace legitimate apps with malicious ones.

4.2 Infection via Personal Computers

Due to a lack of remote exploits for the Android operating system and its security model, which successfully prevents vulnerable, compromised apps from modifying any operating system components, device-to-device infections are virtually impossible. This applies for all Android versions prior to Version 3.1., which features USB host mode. USB host mode can be used to infect other Android-based smartphones with USB debugging enabled. Versions prior to Version 3.1 account for around 90% of all Android devices as of May 2012 [20], as shown by Figure 4.1.

However, to achieve higher infection rates on smartphones, malware authors may turn to desktop computers for smartphone malware propagation in the future. This seems very likely, given the attractiveness of smartphones as a target for malware. Technically, desktop computer malware have to implement the

Android Debug Bridge's protocol to install arbitrary software on any device with USB debugging activated.

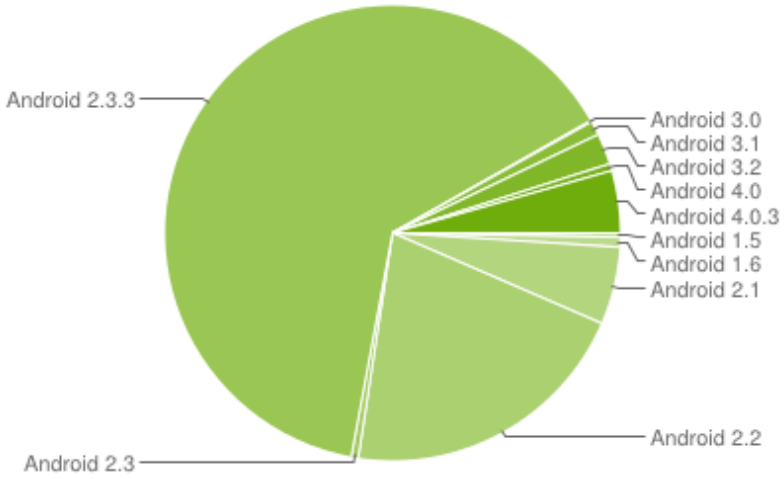

Figure 4.1: Android Version Distribution as of May 1, 2012 [20]

Rooting

To date, vulnerabilities are mainly exploited by users to root their phones, meaning to grant the user full administrative access to their smartphone. Such access is needed for various actions. This includes installation of apps in conflict with the Android security architecture or removing carrier branding, circumventing app or usage limitations (e.g., tethering), or even uninstalling provider-installed spyware. An example of such spyware is Carrier IQ, which is deployed by carriers to retrieve detailed data on customer device usage behavior. It uses rootkit technology to keep its activities unnoticed by users [21]. Furthermore, some users may wish to install modified operating systems on their devices, which is also only possible with privileged access.

This usage model is not driven by a third party's malicious intent. However, rooting one's smartphone may introduce higher risks of successful malware infection. App markets preconfigured for rooted or modified operating systems are not well monitored and contain many trojanized apps. Furthermore, some modified operating systems are less well maintained than preinstalled ones. They also often provide facilities for any installed software to easily gain root privileges. In this case, malicious apps would no longer need to escalate their privileges themselves through vulnerability exploitation. Thus, rooting a smartphone may pose a high security risk.

4.3 Device-to-device Infection

As stated in Section 4.2, autonomous device-to-device propagation is currently not possible. With Android versions 3.1 and 4.0, two major changes have been

introduced which may serve for device-to-device propagation:

- USB host mode (Android 3.1)
- Android Beam (Android 4.0), an NFC (Near Field Communication) based file and data transmission system with a range of approximately 10 cm

USB host mode is very likely to be usable for malware propagation. Similar to desktop computer propagation, an Android smartphone may use the Android Debug Bridge to push and install malicious apps to other devices with USB debugging enabled. This may happen both intentionally or unintentionally:

- Targeted attacks may be conducted against single persons. A device owner only has to leave their device out of sight for a short moment, and an attacker close by may infect it through plugging a USB cable into it. This requires only few seconds.
- Unintended device-to-device infections may occur as well. Given an already infected device, propagation code may run invisibly in background, waiting for other Android devices to be plugged in. Once two devices are connected, the host mode-capable device will imitate the Android Debug Bridge's protocol and infect the other device.

Android Beam is of limited utility, as it requires user interaction for installation. For example, a web link to a malicious app can be sent to another Android 4 device via Android Beam, but the user still has to click the link and confirm it. The limited physical distance reduces malware infection risks even further.
As USB host mode has only recently become available, device-to-device propagation has not yet been reported. However, Android 4 comes shipped with an adb server which allows remote access via shell on other connected Android devices. As a result, malware can use the presupplied adb program to install apps on other devices. Any difficulties in implementing the adb protocol are thus eliminated. Facilities for device-to-device infections are provided by the Android operating system.

4.4 Infection via Rogue Wireless Networks

Open wireless access points are very attractive for smartphone users, as the monthly amount of data which can be transmitted in current plans is very limited. Rogue wireless access points have several options to manipulate data traffic sent from or to a user's handheld device. For example, download and installation requests for apps distributed by single websites instead of the official vendor app market can be easily redirected to malicious APK files. Even legitimate apps may be replaced during transmission. Alternatively, users logging into the rogue wireless network may be presented with a fake website displaying a "critical update" to an app installed on nearly all devices such as Google Search.

Furthermore, the Google Market protocol is capable of forcing devices to install or remove apps through the `INSTALL_ASSET` and `REMOVE_ASSET` commands. If it is possible to impersonate Google servers, manipulate transmitted traffic or successfully hijack a session and redirect it to own servers, an attacker might be able to force smartphones into installation of malicious apps. However, further research into this matter has still to be done.

5 Threat Scenarios

Though conventional desktop computer and mobile device malware share many threats for affected users, some are exclusive to mobile platforms due to their use cases and usage environments. This chapter aims to give an overview of threats for smartphones devices. The implications of mobile botnets and the weakness of cellular network environments will be discussed. We will also illustrate the impact of compromised smartphones on private and corporate targets, e.g., financial fraud and espionage.

5.1 Privacy Issues and Classical Malware Threats

Due to usage scenario integration on mobile devices – most prominently conducting online banking transfers as well as receiving mTANs with the same device, among others – smartphones have become personal communication centers, electronic wallets and even workstations. Use cases slowly evolve towards typical desktop computer fields of application, and even go beyond. Due to the centralization of potentially critical data, platform openness and limited administrative control over a device, as well as presupplied channels for malware distribution, smartphones become highly valuable targets to attack. In the following, attractiveness and threat scenarios will be assessed for private and corporate targets in particular.

5.1.1 Private Targets

Information Leakage through Logging Service

The central Android logging service has proven to be a very rich resource for various personal information. Many apps tend to write status messages to the logging service containing parameters which disclose personal details of their device owners. For example, several GPS-based apps were found to write the device's geo-coordinates to the logging service in regular intervals, thus providing full profiles on the device owner's movements to other apps installed [13]. Some apps log web requests or other network communication. Thus, by only reading log files, much sensitive information can be gathered, depending on the apps installed and their behavior regarding logging. For more details, see Section 3.3.

Online Banking

Online banking transactions are often confirmed and authenticated via the mTAN method. Since the abandonment of printed iTAN lists by many banks, popularity of this method is increasing rapidly, as it is considered easier by customers than TAN generators. In both cases – when a smartphone is only used to receive mTANs or when used to issue transactions as well – the user's bank account is put at high risks, if his device is compromised.

On infected phones, login credentials for online banking portals entered by the device owner can be recorded and forwarded easily. This applies when a compromised smartphone is used for receiving mTANs and for issuing transactions, or even when used only to log in to the banking account to check its balance. Only logging in once is sufficient for an attacker to withdraw all money from an mTAN-protected bank account, as ordering a transaction and intercepting the mTAN text message is trivial. Under the Android operating system, an app can register to receive SMS messages before the phone's own messaging application through the `RECEIVE_SMS` permission. Even without preemptive interception, an attacker only would have to react quickly enough to confirm the transaction after parsing the respective SMS message via the `READ_SMS` permission.

Even when the smartphone is only used to receive mTANs, it may be used to withdraw money from the device owner's bank account under one of the following conditions:

- Both desktop computer and smartphone of one person are under the same attacker's control. This scenario seems very likely in the future, given the circumstances pointed out in Section 4.2. A personal computer infected with any common malware can easily infect a plugged-in smartphone with USB debugging enabled.
- With the information found in a smartphone's messaging application, such as email and SMS messages, targeted phishing or social engineering attacks can be carried out easily. Due to the attacker's knowledge of the owner's communication, such attacks can be conducted in a very sophisticated and convincing way. Hence, infection of the user's desktop computer would not be necessary; compromisation of their smartphone suffices.

Contact Information, Location Data, Credentials and Private Details

Espionage, communication eavesdropping, blackmailing, botnet formation, collecting valid email addresses for spam mailing and recording of sensitive information such as login credentials or credit card data are just some of the classical applications of trojan software. All of these apply for mobile platforms as well. In some cases, they may even be more dangerous on mobile devices: Users are less cautious and store a lot of information and communication centrally.

For convenience reasons, app credentials are retained unencrypted or only obfuscated. When encrypted, the corresponding key is usually saved in plain text and

easily available. This way, users are not forced to enter passwords on a regular basis. As a result, obtaining credentials for any app is trivial, given root privileges. [22]

Multiple real-world espionage cases have gained some attention recently and similar scenarios will become more common, due to thriving smartphone numbers and increasingly sophisticated attacks. Best known is the News of the World eavesdropping case in which the Murdoch-owned newspaper spied on celebrities and politicians as well as on abduction or even murder victims. Although these targeted privacy breaches were not technically sophisticated, they demonstrate very well what can be done with mobile spyware.

Through the possibility of targeted infection of end-user devices as pointed out in Section 4.3, single persons can be attacked successfully in public places such as cafés during short moments of inattention. These attacks can be of high effectiveness when targeting persons of political, military or corporate power or interest.

5.1.2 Corporate Targets

Industrial espionage is one of the biggest threats for medium to large sized enterprises and even to whole economies. Employee and management workstations usually underlie close monitoring and are supplied with security patches centrally and quickly. Corporate security and data policy as well as technology such as firewalls, intrusion detection systems and content filters account for usually very high security standards. They aim to prohibit data breaches and compromisation of employee workstations.

Corporate-supplied smartphones, however, are often less well monitored. Central administration without removing all end-user privileges on their devices is an issue of very high difficulty. Private smartphones usually are not monitored at all. However, they are often used to enter a company's networks, store sensitive work-related documents, carry out work-related communication or are plugged into workstations' USB ports for battery charging.

Such usage may not only result in data breaches where an attacker is able to copy contracts, product designs or other mission-critical documents through a compromised smartphone. It can also lead to the infiltration of the corporate network. An infected mobile device, when logged into the company's network, may be used by attackers as a base of operations for mapping the company's infrastructure, for intercepting internal data traffic or for other forms of attacks and eavesdropping. Company-provided smartphones often contain VPN login credentials, giving an attacker corporate network access at will. Furthermore, operating system security vulnerabilities in USB device management allow for the infection of computers with malware when a compromised smartphone is plugged in, e.g., for charging its battery. The Stuxnet malware was capable of

spreading from USB memory sticks to industrial control and production machines in a similar manner [23].

From this single infected machine, a pivot attack can be conducted to infiltrate the whole corporate network by taking over multiple workstations and servers. Persons in a respective position within their company such as management or research and development can also be targeted specifically to eavesdrop on their communication or steal sensitive documents, without the further aim of infiltrating the corporate network. They can be followed into public places such as cafés or restaurants, infecting their smartphones while inattentive through another smartphone with USB host mode capability (cf. Section 4.3). Alternatively, professional attackers may take the device from a jacket's pocket to infect it within seconds.

5.2 Mobile Botnets

It is a typical feature of malware to connect infected machines for forming botnets which are useful to their operators for various reasons. Typical botnet functionality includes spam message delivery, stealing credentials (cf.Section 5.1) and performing denial of service attacks. Due to limited network capacity of mobile devices and enforcement of bandwidth sharing, the latter is not practicable on smartphones.

Spam message delivery and obtaining credentials is significantly simplified on smartphones with privileged access by an adversary. Any credentials for communication applications can be stolen, once root privileges are acquired [22]. This way, smartphones can not only serve as spam relays themselves, but provide spammers with high quality contact details from various services' address books. Alternatively, spammers can use stolen credentials to deliver spam messages from other machines with better network capacity.

5.3 GSM-based Pivot Attacks

The GSM implementation by many base transceiver stations is prone to various easily conductible attacks. Recent results presented at the 28th Chaos Communication Congress have demonstrated that most European GSM networks are not capable of prohibiting impersonation attacks. In these, an attacker fakes the identity of another GSM subscriber, thus receiving any communication addressed to the attacked person [24].

Projects such as OsmocomBB[1] provide alternative firmware for mobile phone baseband processors. Devices installed with such firmware can easily carry out eavesdropping attacks on other GSM subscribers. Android devices, too, can be

[1] http://bb.osmocom.org/

flashed with baseband processor firmware. However, there are no known attempts to install modified baseband software with additional features similar to OsmocomBB. Such firmware would render any Android device capable of eavesdropping, man-in-the-middle and impersonation attacks on nearby GSM phones.

Hence, just one mobile device under an attacker's control in a radio cell is sufficient to attack any other subscriber and to serve as a remote long-range wiretap. This matter, though theoretically feasible, is of very high difficulty and no further research into it has been conducted yet.

6 Conclusion and Advisory

6.1 Conclusion

Most current successful attacks can be attributed to negligent user behavior. As pointed out by recent research and publications, however, attacks on Android-powered devices are becoming more sophisticated. They are now capable of spreading mechanisms which do not require explicit user confirmation. Malware may be delivered unnoticed through desktop computers, other Android devices or trojanized apps.

Due to the open usage model of the Android market, malicious apps cannot be avoided completely. Especially pirated apps or multimedia content in popular demand targeting user groups with typically low awareness levels are predestined to spread to many devices before being identified by Google as malware. Google's "killswitch" `REMOVE_ASSET` command will not be able to delete modern malware from Android devices in the future. Malicious apps utilizing root exploits to escalate their privileges can inject code and place binaries outside app storage locations. Removing the APK file and its working directory will no longer suffice.

Furthermore, targeted attacks are becoming more feasible. USB host mode capable Android devices may be used to quickly infect USB-debugging enabled smartphones of persons of military, political or economical interest. This is made trivial by the adb program which is preinstalled on any Android 4.0 device. Also, classical desktop malware can serve as a distribution channel, making up for the lack of self-spreading capabilities of Android malware.

With our extensible exploit execution framework, we developed a test environment to evaluate and emulate local attacks and malware. We can furthermore assess the vulnerability of multiple devices easily and simultaneously. Currently, we are developing an extension to our exploit execution framework which will demonstrate device-to-device infection capabilities.

Several new and well-known threat scenarios apply for Android smartphones. These include easily conductible money fraud, industrial espionage, corporate or military network infiltration and even denial of service attacks on today's already heavily loaded mobile networks. Android devices may even serve as remote bases for attacks on other GSM subscribers, though this is regarded highly improbable.

6.2 Advisory

Given the weak operation environment like cellular networks and considering the security-critical corporate and private usage scenarios, Android-based smartphones should prohibit the execution of any native code added after shipping completely. This can be accomplished by either providing an option in the operating system's settings or a dialog asking permission for setting the "execute" bit for files.

It may seem like a drastic measure, but the possibility to execute native code was introduced at a time when processing power was limited. It is stated in the Android Native Development Kit documentation that one of the main purposes of native binaries is to execute "self-contained, CPU-intensive operations" [25]. However, new devices are equipped with much more advanced hardware which no longer requires code parts to be native to run fluidly. Prohibiting non-vendor native code is the only way to contain exploits against system security flaws which are not patched quickly enough, or not patched at all.

Alternatively, the privilege of setting the executable bit for files may be limited to the root user and to the Android Market app. This way, developers may provide native binaries with their APK files which will be marked executable by the Market app. Any file added to the file system later on, i.e. not during the installation process, cannot be declared as executable. SEAndroid has proven very effective for this objective.

Code signing of native code may also be an option. The number of apps making use of own binaries is very limited. Thus, it is practicable to require native code to be signed. The signature might be a verified developer's, or, after thorough checking for malicious code portions, may be given by Google. Testing could include common malware heuristics checks, sandbox execution and system/API call pattern analysis. Prohibiting execution of non-signed binaries could also prevent usage of exploits in usual apps.

The flaws of the Android permission model pointed out in Section 3.3 which do not lead to full compromisation of devices, but to privacy breaches, can only be corrected by Google. Of course, device manufacturers should be responsible to supply their customers with security patches. In many cases, customers would be forced to acquire a new phone to receive security updates, because their old handheld device is simply abandoned.

The final party involved is the customer. Awareness for security risks on mobile platforms seems significantly lower than on desktop platforms. Identical or even more caution should be applied for mobile devices, as the trustworthiness of software cannot be determined as easily.

Bibliography

[1] Gartner, "Mobile Device Sales Q3 2011," November 2011. http://www.gartner.com/it/page.jsp?id=1848514.

[2] Android Security Team, "Android Security Overview," December 2011. http://source.android.com/tech/security/index.html.

[3] Android Team, "What is Android?," Feburary 2012. http://developer.android.com/guide/basics/what-is-android.html.

[4] National Security Agency, "SELinux," January 2009. http://www.nsa.gov/research/selinux/.

[5] C. Mullaney, "Android.Bmaster: A Million-Dollar Mobile Botnet," February 2012. http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet.

[6] H. Lockheimer, "Android and Security," February 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[7] J. Oberheide, "remote kill and install on google android," June 2010. http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/.

[8] T. Bray, "Exercising Our Remote Application Removal Feature," June 2010. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.

[9] T. Vidas, D. Votipka and N. Christin, "All your droid are belong to us: A survey of current android attacks," in *5th USENIX Workshop on Offensive Technologies*, Carnegie Mellon University, August 2011. http://www.usenix.org/event/woot/tech/final_files/Vidas.pdf.

[10] S. Smalley, "SE Android release." SELinux Mailing List, Mailing List Archives (marc.info), January 2012. http://marc.info/?l=selinux&m=132588456202123&w=2.

[11] National Security Agency, "SEAndroid Project Page," January 2012. http://selinuxproject.org/page/SEAndroid.

[12] S. Smalley, "The Case for SE Android," January 2012. http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf.

[13] A. Lineberry, D. Richardson and T. Wyatt, "These Aren't The Permissions You Are Looking For," in *DEF CON 18*, August 2010. https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf.

[14] T. Cannon, "No-permission android app gives remote shell," December 2011. http://viaforensics.com/security/nopermission-android-app-remote-shell.html.

[15] B. Königs, J. Nickels and F. Schaub, "Catching AuthTokens in the Wild - The Insecurity of Google's ClientLogin Protocol," June 2011. http://www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html.

[16] J. Oberheide, "Android Hax," in *SummerCon*, June 2010. http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf.

[17] X. Jiang, "GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread)," August 2011. http://www.cs.ncsu.edu/faculty/jiang/GingerMaster/.

[18] D. Fisher, "GingerMaster Malware Seen Using Root Exploit for Android Gingerbread," August 2011. http://threatpost.com/en_us/blogs/gingermaster-malware-seen-using-root-exploit-android-gingerbread-081811.

[19] Lookout Mobile Security, "2011 Mobile Threat Report," 2012. https://www.mylookout.com/_downloads/lookout-mobile-threat-report-2011.pdf.

[20] Android Team, "Platform Versions," February 2012. http://developer.android.com/resources/dashboard/platform-versions.html.

[21] T. Eckhart, "Carrier IQ," November 2011. http://androidsecuritytest.com/features/logs-and-services/loggers/carrieriq/.

[22] Android Bug Tracker, "Issue 10809: Password is stored on disk in plain text," August 2010. https://code.google.com/p/android/issues/detail?id=10809.

[23] M. Brunner, H. Hofinger, C. Krauss, C. Roblee, P. Schoo and S. Todt, "Infiltrating Critical Infrastructure with Next-Generation Attacks - W32.Stuxnet as a Showcase Threat," December 2010.

[24] K. Nohl and L. Melette, "Defending mobile phones," in *28th Chaos Communication Congress*, 2011. http://events.ccc.de/congress/2011/Fahrplan/attachments/1994_111217.SRLabs-28C3-Defending_mobile_phones.pdf.

[25] Android Team, "What is the NDK?," January 2012. http://developer.android.com/sdk/ndk/overview.html.