Google Android: An Updated Security Review

Yuval Fledel, Asaf Shabtai, Dennis Potashnik, and Yuval Elovici

Deutshce Telekom Laboratories at Ben-Gurion University,
Beer-Shava, Israel
{fledely,shabtaia,dennisp,elovici}@bgu.ac.il

Abstract. Among the most significant smartphone operating systems that have arisen recently is Google's Android framework. Google's Android is a software framework for mobile communication devices. The Android framework includes an operating system, middleware and a set of key applications. Designed as open, programmable, networked devices, Android is vulnerable to various types of threats. This paper provides a security assessment of the Android framework and the security mechanisms incorporated into it. In addition, a review of recent academic and commercial solutions in the area of smartphone security in general and Android in particular is presented.

Keywords: Mobile devices, Google, Android, Security.

1 Introduction

Among the most significant smartphone operating systems that have arisen recently is Google's Android framework. Designed as open, programmable, networked devices, Android is vulnerable to various types of attacks that can make the phone partially or fully unusable, cause unwanted SMS/MMS billing, expose private information, or infect every name in a owner's phonebook [1].

Smartphone market share in the US has increased from 11 percent of all cellular phone subscribers in 2008 to 17 percent in 2009, and it is expected to increase significantly over the next few years, almost fivefold by 2013 [2]. The Android framework has gained much interest by both the developers' community and smartphone users in a relatively short period of time. In fact, according to [3] Android is the fourth most popular smartphone in the US as of February 2010. Smartphones based on the Google Android operating system are expected to increase 10 percent during 2010 [4]. Consequently, smartphones are likely to become a fertile ground for various types of threats. Another major factor attracting hackers is that smartphones are often carried for business purposes and are likely to have sensitive and valuable information. They also provide remote access to a company's most sensitive data, which can lead to data leakage if their phones are hacked into.

The increasing number of attacks on mobile platforms along with the increasing usage has led many security vendors and researchers to propose a variety of security solutions for mobile platforms. As a case in point, Symbian and Google have designed their operating systems to enable applications to run only in specialized

M. Griss and G. Yang (Eds.): MOBICASE 2010, LNICST 76, pp. 401–414, 2012.

sandboxes, minimizing the capability of malware to spread [5]. A robust application signing and certification mechanism was integrated into Symbian's operating system and was proven highly effective in reducing malware attacks. The risks to Android are nevertheless significant, mainly because it's an open source and open platform software stack operated in a heterogenic mobile environment. On one hand, it allows introducing new applications and services very quickly. On the other hand, it raises security issues that the academic community and security vendors attempt to address. This paper reviews and assesses the security mechanisms incorporated into the Android framework. Additionally, a list of security mechanisms which can be incorporated to harden the Android is presented.

2 The Android Framework

The Android[1] software stack is built on the *Linux kernel*, which is used for its device drivers, memory management, process management, and networking. The next level up contains the *Android native libraries*. These libraries are written in C/C++ and are used by various system components in the upper layers. Incorporating these libraries in Android applications is achieved through Java interfaces or native compiled code. The next level is the Android *runtime*, comprising of the *Dalvik Virtual Machine* and the *core libraries*. Dalvik runs .dex (Dalvik executable) files that are designed to be more compact and memory-efficient than Java .class files. The core libraries are written in Java and provide a substantial subset of the Java 5 SE packages as well as some Android-specific libraries. The *Application Framework* layer, written fully in Java, includes Google-supplied tools as well as proprietary extensions or services. The topmost *Application* layer provides applications such as phone, web-browser and email client.

Each application in Android is packaged in an .apk (Android package) archive for installation. The .apk is similar to a standard Java jar file in that it holds all code and non-code resources (e.g., images, manifest) for the application. The applications are written in Java based on the APIs provided by the Android SDK. An Android package is basically a collection of components: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. Components in one .apk are isolated from components in another .apk and can only communicate with each other and share data through means provided by the system. Each .apk is associated with a primary process in which all of the application's components are executed. Enck et al. [6] and Burns [7] provide an overview of the main components of an Android application and guidelines for using the Android-specific mechanisms correctly in order to protect the application.

Android is a multi-process system, where each application (and parts of the system) runs in its own process. For the most part, security between applications and the system is enforced at the process level through standard Linux facilities, such as POSIX user IDs and group IDs assigned to applications. Files in Android (both application- and system-files) are subject to the Linux file permission mechanism. In

[1] www.android.com

addition, access-control is provided through the application level permission mechanism that enforces restrictions on the specific operations that a particular application can perform. Signing applications is another significant security feature in which all of the application's files are signed along with their meta-data in the .apk. A review of Android's inherent security mechanisms is provided in [8].

3 Android Security Analysis

In this section we describe findings from our assessment of various security aspects of the Android framework. The results presented in this section were validated on T-Mobile G1 and HTC Desire devices.

3.1 Analysis of the Android Framework's Cornerstone Layers

This subsection presents the outcome of our analysis of Android's lower layers. We adopted a security-oriented code-review approach to identify potential vulnerabilities. We focused on special locations that might be problematic such as interfaces, structures and configurations. The source code that was reviewed is from the Android open source project repository.

Linux Kernel

In 2009, 111 CVE (Common Vulnerabilities and Exposures) entries were logged for the Linux kernel (i.e., two entries a week). Drivers and vendor-specific additions are the main locations for these vulnerabilities. Android contains a considerable amount of vendor-specific code. Therefore, *Android code should be submitted to mainline*. Code that is integrated into the mainline passes several phases of checks and validations that are likely to identify and remove bugs; some may have security implications. However, Android has diverged from the mainline kernel and Android vendors have so far published the code only after shipping a product on the market.

Android modifications include *hardcode POSIX user ids (uids) and group ids (gids)* in the kernel code. These modifications contradict the basic design decisions in Linux. However, it increases security since the system services are not required to run with root privileges. Two examples of such modifications are the "paranoid network" that limits network access based on gids, and the Binder which accepts the first process that uses it as its master, but only if it has the "system" group.

The *Linux kernel is highly configurable*. On a typical Android device, common Linux options are disabled in order to reduce memory consumption. Less code also means a smaller attack surface. On the other hand, it also means omitting "security enabler" modules. A few examples from HTC Desire that have security implications are: disabling the auditing support and BSD task accounting (less input is available to intrusion detection systems); disabling SYN cookie support (if enabled, it can reduce the chance of SYN flood attack); and disabling security modules (e.g., SELinux). The following modules are enabled: PPTP, L2TP and IPSec-based VPN connections (supported on Android from release 1.6); CFS scheduler group scheduling; disk

encryption; and NetFilter (provides firewall capabilities). Enabling these modules in the kernel configuration requires a trivial amount of effort and also consumes minor memory space. However, providing the means for using these modules is not trivial and requires additional user-space components that will provide the API for these modules in order to avoid the need for root access.

Android employs Linux's *Completely Fair Scheduler (CFS)* that ensures that an equal share of CPU is distributed among all processes. In addition, specific processes can be granted a larger share, but are still prevented from monopolizing the CPU. To test this fairness mechanism we have created a simple application that starts 100 threads that loop doing nothing in particular. Running this application on a T-Mobile G1 device resulted in the entire device being frozen.

Linux also supports *storage quota*, but Android does not enable it. This means that an application can create files of any size, on both the internal flash and the SD card. Files can be created outside of the private application folder and therefore do not counted in the application's size figure.

Finally, we tested the applicability of existing *Linux root-kits and key-loggers* that usually require root level access in order to install. We successfully compiled and activated the rathole and Linux rootkit V (lrk5) root-kits, and the vlogger key-logger on Android [9]. The running root-kits on Android can remotely explore running processes, hide or kill specific processes, prevent hidden processes from being stopped, enable packet sniffing and provide several methods to communicate with the root-kit such as encrypted SSH backdoor and remote shell. Using the key-logger we managed to log keystrokes on the devices keyboard [9]. Detailed description of similar experiments is provided in [3].

System Libraries

Android makes use of many native libraries. These libraries are intended to be used by native processes, other native libraries or by Dalvik through Java Native Interface (JNI). JNI is normally used for: (1) providing low-level functionalities; (2) implementing computationally intensive calculations; (3) hiding code and licensing issues; and (4) leveraging existing libraries.

The native libraries are written in C/C++, which is not type safe. Thus, native libraries have a higher chance of bugs than Java code. Since JNI loads native libraries into the memory space of a Dalvik process, bugs in the native library may crash the Dalvik process, corrupt its memory or cause arbitrary code execution. For that reason, system libraries are a target when searching for vulnerabilities. Example vulnerabilities in Android native libraries are CVE-2009-0606, and CVE-2009-0608. Usually, such vulnerabilities stem from using outdated vulnerable versions of ported libraries and not keeping pace with upstream bug fixes.

Dalvik Runtime

Dalvik is a Java Virtual Machine (VM) based on Apache Harmony which was extensively modified and adapted for environments with low memory. Securing Dalvik is crucial since vulnerability in the VM affects all applications. Dalvik

provides the possibility of executing native code through JNI without requesting permission for it. Employing native code, however, removes the layer of defense provided by the VM.

A potential weak spot is the .dex file loading code which is required to deal with .dex files from unreliable sources. The verification process of .dex files that is performed during the installation of the application is also applied whenever the .dex files are loaded to memory. By inspection of the Dalvik code we conclude that sanity checking is implemented in the initial loading code. However, we identified unchecked pointer operations and as a result, we were able create a malformed .dex file that during installation caused the Package Installer to crash resulting in a phantom application that cannot be uninstalled because the installer claimed it is already installed, nor can another package with the same package name be installed if it has a different signature.

Pure "desktop" Java malware spreads by injecting code into other class files without harming the valid structure of the victim class file and its verifiability. Java malware such as StrangeBrew and BeanHive search for writable class files in the user's working directory and then modify them to start execution from the viral segment. From our experiments with the StrangeBrew malware we conclude Java malware are not applicable to the Android framework for two reasons. First, they infect class file formats and must be adjusted to support injecting malicious code into .dex files. Second, in Android, applications do not have write privileges to any .apk files. Moreover, since it is not possible to list a folder of another application for its files, any effective search for files to infect is not feasible [9].

Forcing Windows OSs (XP and Vista) to automatically run a malicious Windows executable from a T-Mobile G1 device (located on the SD-card) by using an Autorun.inf file was also tested and found not feasible [9].

3.2 Application-Level Permissions

The application-level permission mechanism is responsible for securing APIs provided by the system and other applications [8]. Whereas some of the core permissions are reserved for Google applications, a large variety of Normal and Dangerous protection level permissions are still available for non-Google applications. As a result, abuse of such permissions is inevitable. As an example, an Internet access with the ability to read various contents stored on the device (e.g., files on the SD card, SMS messages or GPS tracking) can be used to acquire confidential information or to spy on the user without his/her knowledge. Other examples include: Denial of Service (DoS) attacks (e.g., denying the ability to place phone calls or draining the battery) and the abuse of paid services (e.g., phone calls, SMS/MMS messages, and chargeable network traffic). As a case in point, SMobile analysis of 68% of the applications available on the Android market indicates that 20% of the applications request permissions to access private or sensitive information that can be used for malicious attacks. Small portion of the applications have the ability to brick the device, read or use the authentication credentials from another service or application, or send unknown premium SMS messages without the user's

authorization [9]. A recent example is the AndroidOS.FakePlayer.a Trojan horse that masquerades as a media player but in the background sends SMS messages to premium rate numbers.

Another source of difficulties arises from the shared user-ID feature. When an application, declaring a shared user-ID, is installed, all of the granted permissions are ascribed to the shared user-ID. At runtime, each of the applications sharing that user-ID will be granted by a combined set of permissions. A simple attack scenario based on exploiting the user-ID feature would probably take place as follows. The user installs two applications sharing a user-ID. The first requests access to the Internet while the second wishes to access the contact list. As soon as both applications are installed, each application is capable of both reading the contacts and sending them through the Internet. The user, however, is unaware of the collaboration between the completely unrelated applications. A major design flaw is that the user does not have the ability to only partially grant the requested permissions.

3.3 Installing Applications

The Package Manager (the service responsible for the installation process) validates the correctness of the .apk during installation. Validation includes: verification of the digital signature; confirmation of legitimacy of shared user-ID and permission requests; and the validation/verification of the included .dex file. The package installation API is guarded by the INSTALL_PACKAGES permission which is of Signature protection level and defined in the core Android package. Thus, malicious applications cannot install applications on their own. The Package Installer, a legitimate application-level wrapper for the Package Manager is included as one of the core applications that are provided with the Android-operated device.

There are three main methods for installing .apk files. The first, which is intended mainly for developers, is using the Android Debug Bridge (a command-line tool that is supplied along with the SDK). The installation command is issued from a PC while the Android device is connected via USB connection. The actual installation is done directly by the Package Manager without any user interaction and therefore has the ability to propagate worms from PCs to devices silently. The two remaining installation options are intended for the device owners. The first one is installing via Android Market. Since the Market application is signed by Google, it can interact directly with the Package Manager. The last installation method is based on installing applications from the SD card using 3rd party applications that enable the user to search for .apk files on the SD card and to initiate the installation process which is carried out by the Package Installer.

When installing an existing application, the installation will be allowed only when the signatures of the existing application and the new application match. The signature-matching safeguards against malicious applications that attempt to gain access to private data through substitution of the original package.

3.4 Web-Browser

Web-browsing exposes Android users to common attacks such as: Cross-Site Scripting (XSS); URL encoding attacks; social engineering; and malicious scripts. WebKit, Android's open-source Web engine, has a history of vulnerabilities. Previous attacks on the Web browser include a buffer overflow in an outdated native library, and an explicit XSS vulnerability. Both attacks enabled the attacker to run any malicious code on the device with all the abilities and privileges assigned to the Web browser application. Since the browser runs with its own POSIX user-id, an attack is limited to the browser, leaving other phone functions (e.g., dialing or messaging) unharmed. The browser is also limited by the application-level permissions it has been granted at installation (Internet access, ability to acquire wake locks, location-based APIs, network-related information retrieval, and writing to the SD-card). Nevertheless, in a successful attack, the attacker could gain information stored by the browser such as cookies, passwords, favorites and form-field values. Having access to all of the browser's private data, the attacker could corrupt it in order to prevent correct operation in the future. The browser provides several security-related configurable options. These include remembering form data and passwords; accepting cookies; displaying security warnings; loading images; enabling JavaScript; blocking pop-ups; and setting (or disabling) the homepage.

3.5 Connectivity and Communication

Multiple communication transports (Bluetooth, Wi-Fi, cellular, cable) provide many options for malware to infiltrate a device. Some malware can propagate through more than one transport. For example, Lasco is a malware which spreads via the Bluetooth on Symbian devices [11]. In addition, it also infects all Symbian Installation Source (SIS)-files using social engineering.

Bluetooth on Android supports pairing and audio headsets. Additional functions can be enabled using 3rd party applications (e.g., Object Push). For the pairing process, Android allows itself to become discoverable, but only for a short duration of two minutes. In addition, the owner needs to accept the connection. This significantly decreases the likelihood of being detected by attackers).

The two USB sub-protocols which are supported by Android are: mass-storage device and the Android Debugger Bridge (adb). By default, adb is disabled, and the device is mounted as mass-storage device where only the device's SD card are exposed. When USB debugging is enabled, the device can be managed with the adb tool which is provided in the Android SDK. This tool makes it possible to push and pull files to and from the device, install .apk files, redirect TCP and UDP packets, etc.

One of the Android-specific Linux kernel changes is the "Paranoid-Network". Usually, on Linux systems, any user-space process can open network connections at will. On Android, a user-space application must receive the INTERNET permission in order to make any kind of network connection. The enforcement of the INTERNET permission is done at the kernel level and therefore even native applications are subject to this setting. The Paranoid Network setting works by hard-coding several

POSIX group IDs in the kernel. An application must be a member of the relevant group before it is allowed to create sockets.

3.6 Conclusions

From our analysis of the Android framework we identified two main threat classes which should be countered by employing proper security solutions/capabilities. First, whenever discovering a bug or vulnerability in one of the core components (such as a native library or a kernel component), an attacker might be able to run malicious code in a highly privileged mode and even gain full control over the device. This threat is amplified due to the fact that Android's code is publicly available; some system processes run with root privileges; and no fine-grained access control mechanism exists for system processes. Second, the application-level permission mechanism is not sufficient and installation of an application that maliciously uses permissions granted by the unaware user is a scenario which is likely to occur. The framework also provides the adb install feature that makes it possible to install applications and to grant permission to an application without any user interaction. In addition, a user cannot approve a sub-set of requested permissions (it is "all-or-none") and cannot verify that an application uses its granted permissions only for benign purposes. Moreover, the shared user-ID mechanism allows sharing permissions between applications without a user's awareness or the need for explicit approval.

These threat classes may results in compromising the availability, confidentiality and/or integrity of private content that is stored on the device (e.g., pictures, contacts, emails, documents), applications and services (e.g., phone, messaging, emailing, Internet) and resources (e.g., battery power, communication, memory and CPU).

4 Applicable Security Mechanisms for Android

In order to further harden an Android device and mitigate the identified high-risk threats, additional safeguards may be employed. Some of these mechanisms were tested and evaluated in our mobile security laboratory. Several security companies, such as SMobile, Mocana, McAfee and DroidSecurity are already providing security solutions for Android. Additionally, several security mechanisms have been proposed and evaluated by the academic community. In the following paragraphs we provide a description of several security mechanisms that can be adapted to harden the Android.

Anti-malware

To identify and remove malware, anti-malware software examines files, email attachments, memory, system configuration, MMS, Bluetooth objects, etc. It usually identifies known malware based on a signature repository. As mentioned earlier, several commercial solutions are available for Android which also provides an anti-malware component. There are also open-source anti-virus and rootkit detectors that can be ported to Android such as the ClamAV [12]. Anti-malware is a well-known solution and is extensively used in other platforms. Signature-based solutions provide

low false-positives, but will only detect known malware and require continuous updating of the signature repository. At this time, the anti-malware solution does not seem to be effective for mobile devices.

Firewalls

A firewall running on Android can prevent remote network attacks. It is a well known and highly effective solution; however, it will not protect against attacks via web-browser, SMS/MMS, email or Bluetooth and will not provide phone call filtering.

We have implemented a preliminary Firewall for Android which is based on NetFilter. NetFilter is a Linux kernel subsystem that provides firewalling capabilities (e.g., packet filtering and connection tracking capabilities). NetFilter is enabled on Android devices including T-Mobile G1 and HTC Desire, thus only a control application is needed. However, in order to update the firewall policy, the control application should run with root privileges.

In the basic firewall, that we activate on Android rules are very simple and provide the ability to block communication to/from specific IP addresses and ports. The more suitable firewall policy for Android is one that allows defining rules at the application level. In such a way the policy will define for each application who can access it and where it can send information to. We can also make sure that port scanning is not preformed from the device by a malicious application. However, firewalling at the application level is hard to achieve since, as mentioned before, any application that is granted with INTERNET permission can open a socket at its will.

Intrusion Detection System

Host-based intrusion detection system (HIDS) monitors the device, applications or user's behavior to detect/prevent abnormal or known malicious behavior. Anomaly-based IDS can detect unusual phone call/SMS activity, denial of service attacks, and protect the information on the device in case of theft or loss. While it may detect new and isolated attacks, it will probably suffer from high rate of false positives.

Most academic initiatives to enhance protection of mobile devices have employed host-based intrusion detection systems comprising an agent collecting various features from the device and then applying various machine learning algorithms to classify the behavior of the system as benign or malicious or to detect anomalies [9]. In our Android security research we developed and evaluated the "Andromaly", which is an experimental anomaly-based IDS for Android [13][14]. Andromaly employs various methods, such as anomaly detection and temporal reasoning, to facilitate detection of maliciously behaving applications. An IDS such as the Andromaly can be used for reporting suspicious behavior of applications to Google via the Android Market.

Access Control

Android incorporates several access control mechanisms. While these mechanisms are enforced on the application level or only on files, Linux can provide other tools that are directly enforced by the kernel. As a case in point, we tested the Security-Enhanced Linux (SELinux) on an Android G1 device [15]. SELinux allows restricting

of any process in the system, including root-owned, and by that limiting access of processes and users to resources and/or services, thus limiting the potential damage from malicious or exploited applications. Its decisions are based on an access control policy, which should be deployed together with the base system. Our experimentation with SELinux on Android has shown that it consumes very few resources and incurs a very low overhead [15].

Android provides simple authentication functionality based on a screen lock pattern mechanism. Such mechanism can be extended so that the device can be locked remotely (when the device is lost or stolen), or by protecting sensitive information stored on the device, or on the SD card using password-based encryption.

In the same context, Ni et al. [16] present the DiffUser framework that provides role-based access control mechanism for smartphone users. DiffUser was implemented and evaluated on Android. Each user can be assigned with different rights. For example, only an administrator can install/uninstall applications; the guest user can only use the phone application.

Protecting Android Permissions

During the installation of application on Android, the user may view a list of required permissions, and may decline installation based on this list. However, there is no way for the user to allow only a subset of the required permissions. Nauman and Khan [17] added an advanced feature to the Package Installer enabling the user to decline certain requested permissions but still permit installing of the application. Such a change would be highly beneficial to security aware users. This solution would protect from granting unneeded permissions that could be maliciously used. However, applications granted with a partial set of permissions may crash if the developer did not anticipate and provide a solution for such a situation (i.e., handle cases in which partial permissions were given). This solution can be enhanced for corporate users to provide the option for hardening Android devices by limiting permissions granting based on a predefined policy.

Additional efforts for enhancing Android security at the application level permissions are presented by the Kirin system [18] and Secure Application INTeraction (SAINT) [19]. These two systems presented an installer and security framework that realize an overlay layer on top of Android's standard application permission mechanism. This layer allows applications to exert fine-grained control over the assignment of permissions through explicit policies.

Spam-Filter

A spam filter blocks unwanted MMS, SMS, emails, and calls from an unreliable origin. In the mobile phone arena, spam filters are implemented using the white/black listing approach, with caller ID and words/phrases dictionary being used as the source for allowing/blocking a call or a message. Products for spam filtering on Android are already available. eMail spam filtering can be provided by either the email server (e.g., gmail account) or by Android client side email application.

Application Certification

Android uses certificates in a limited way in order to ensure package integrity and that two or more packages are from the same origin. Applications that define their own permissions may choose to grant such permissions only to packages sign by the same author. There is no support for root Certificate Authorities (CAs) or for certificate chains in Android. In order to employ the application trust mechanism, Android needs to be modified to support trust levels of applications, associating CA certificates to the trust level, as well as verifying certificate chains. This mechanism is highly effective in detecting malicious applications before they are installed on a device. However, this solution is highly expensive in terms of implementation and maintenance.

Certification process was implemented by other mobile operating systems (e.g., Symbian, iPhone and Blackberry). Although certification has been proven very effective, it is not error prone and malicious applications can still unintentionally be approved and signed. In addition we can assume that users will continue to download and install "unapproved" applications that are available from free websites and prefer them over trusted applications that need to be paid for. Furthermore, Android is grounded in an open source approach, while the certification framework contradicts this approach; thus researchers should look for alternatives to capture application semantics without relying on manual code inspection.

Automated Code Analysis and Verification

Android .apk files encapsulate valuable information that can help in understanding an application's behavior. This information includes requested permissions, framework methods called by the application, framework classes used by the application, User Interface widgets and more. We took that avenue by exploring the use of machine learning classifiers on static features extracted from Android's application files [19]. In this approach, the application file is represented by static features extracted from the file and the classifiers are then applied to learn patterns in the code in order to classify new files. Schmidt et al. [12] evaluated a framework for static function call analysis and performed a statistical analysis on function calls used by native applications. Chaudhuri [21] presented a formal language for describing Android applications and data flow among application's components. This formal language can be used for statically analyzing Android applications and data flow between applications and comparing those with security specifications defined in the application's manifest. This provides the ground for security decisions such as is the application safe and does it do what it claimed to do. Therefore it can provide the means for a developer to certify is application, and for the user to verify the proof of the certification before installation.

Such an approach is closely coupled with certification and can provide an automated alternative as a part of the certification process; developers can certify their applications, and users can verify the proof of the certification before installation. Such a method can also be used for rapid examination of Android packages and informing Google team, via the Android Market of suspicious applications.

Data Leakage Prevention (DLP)

DLP is a relatively new field in computer security used mainly by enterprises. DLP mechanisms prevent sensitive/private content from leaking out. Identification of such content is done by applying various content and context inspection mechanisms (e.g., predefined keywords and patterns/regular expressions, fingerprinting of sensitive content and statistical algorithms). These mechanisms haven't been integrated yet into smart mobile platforms despite the fact that these devices can store content that should be protected (e.g., location, documents, contacts, calendar etc.)

We have investigated the implementation of DLP solution on Android. Its main requirement is the possibility to monitor and block outbound communication over the network (Wi-Fi/3G), SMS, or MMS. We analyzed several ways to hook into the data flow. Due to security considerations, such changes require deep OS integration, and can't be supported by just adding a regular (add-on) Android application.

Network data flow can be interrupted in the following locations: Java API, C API (i.e., native libraries) and kernel. *Altering the Java API* is simple but can be easily bypassed by calling the native libraries. *Hooks in native libraries*, such as libc, requires relatively low maintenance yet is can still be bypassed by using statically-compiled binaries. *Kernel hooking* is very difficult to bypass, but requires high maintenance.

Interruption of SMS messages can be done in the following locations: SMS application, application framework, serial line, rild daemon and kernel. *Altering the SMS application* requires low effort but can be easily bypassed by installing a 3rd party SMS application. *Modifying the application framework* is easy to implement and also difficult to bypass. *The serial line* used by rild can be rerouted to a monitoring daemon as demonstrated by Mulliner and Miller [22]. *Altering the rild daemon* and *the kernel* is also possible, however, it is has no additional benefits over the previous methods.

MMS is simply an SMS message that is marked as non-textual, and contains a URL. Therefore monitoring MMS is similar to the combination of file uploads (i.e., network monitoring), and SMS monitoring.

An Additional DLP feature, anti-theft, is provided for smartphones by several security vendors. This feature provides remote control capability over the device in case it gets lost or stolen. This module enables to locate the device, block it and wipe its data remotely.

5 Summary and Conclusions

In this paper we analyzed security issues pertaining to Google's Android in order to identify potential security flaws that should be mitigated using security solutions for Android devices. The risk arising from these vulnerabilities is amplified by the fact that as a smartphone, Android devices are expected to handle personal data and provide PC-compliant functionalities, thereby exposing the user to all the attacks that threaten users of personal computers.

We reviewed the security-related mechanisms that are inherently integrated in the Android framework and surveyed additional security mechanisms that can be applied on Android-based handsets. Several of these mechanisms were tested and evaluated in our laboratory. A security suite for mobile devices, especially open-source and open platform such as the Android, should include a collection of tools, optionally operating in collaboration.

Our review indicates that the defensive shell around Android was designed with extensive care since the security mechanisms embedded in Android address a broad range of security threats. However, despite these Android-integrated measures we conclude that it is highly important to incorporate a mechanism that can prevent or contain potential damage deriving from an attack on the Linux kernel layer such as the SELinux access control mechanism. Also, better protection should be added for hardening the Android permission mechanism and protecting owner's private data by modifying the permission mechanism, using a firewall, Intrusion Detection System, automated static analysis, encryption and a DLP mechanism.

Finally, remote management mechanisms can be used to consolidate several other security mechanisms while providing the ability to remotely control, configure and manage the device (e.g., setting network parameters or firewall policy, pushing security updates, tracking the device location, uninstalling/installing applications, bricking the device and deleting or encrypting data). Context-aware capabilities can also be added to dynamically allow and restrict access to resources (documents, emails) and services (camera, Internet, phone, messaging) based on a predefined policy and on the instantaneous context of the device.

References

1. Piercy, C.: Embedded devices next on the virus target list. Electronic Systems and Software 2(6), 42–43 (2005)
2. Frost, Sullivan: World mobile anti-malware products markets. Frost and Sullivan Report # M154-74 (2007)
3. Papathanasiou, C., Percoco, N.J.: This is not the droid you're looking for. In: DEF CON 18 (2010)
4. Pelino, M.: Predictions 2010: Enterprise Mobility Accelerates Again. Forrester (2009)
5. Lawton, G.: Is It Finally Time to Worry about Mobile Malware? Computer 41(5), 12–14 (2008)
6. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security and Privacy 7(1), 50–57 (2009)
7. Burns, J.: Developing Secure Mobile Applications for Android. Technical Report, iSEC (2008)
8. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google Android: A Comprehensive Security Assessment. IEEE Security and Privacy 8(2), 5–44 (2010)
9. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S.: Google Android: A State-of-the-Art Review of Security Mechanisms. CoRR abs/0912.5101 (2009)

10. Vennon, T., Stroop, D.: Threat Analysis of Android Market (2010), http://threatcenter.smobilesystems.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf
11. Emm, D.: Mobile Malware – New Avenues. Network Security 2006(11), 4–6 (2006)
12. Schmidt, A.D., et al.: Enhancing Security of Linux-based Android Devices. In: 15th International Linux Kongress, Germany (2008)
13. Shabtai, A., Kanonov, U., Elovici, Y.: Intrusion Detection on Mobile Devices Using the Knowledge Based Temporal-Abstraction Method. Journal of Systems and Software 83(8), 1524–1537 (2010)
14. Shabtai, A., Elovici, Y.: Applying Behavioral Detection on Android-Based Devices. In: Cai, Y., Magedanz, T., Li, M., Xia, J., Giannelli, C. (eds.) Mobilware 2010. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 48, pp. 235–249. Springer, Heidelberg (2010)
15. Shabtai, A., Fledel, Y., Elovici, Y.: Securing Android-Powered Mobile Devices Using SELinux. IEEE Security and Privacy 8(3), 36–44 (2010)
16. Ni, X., Yang, Z., Bai, X., Champion, A.C., Xuan, D.: DiffUser: Differentiated User Access Control on Smartphones. In: Proceedings of the 5th IEEE International Workshop on Wireless and Sensor Networks Security (2009)
17. Nauman, M., Khan, S.: Design and Implementation of a Fine-grained Resource Usage Model for the Android Platform. To appear in International Arab Journal of Information Technology (2010)
18. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of Computer and Communications Security Conference, pp. 235–245 (2009)
19. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: Proceedings of the 25th Annual Computer Security Applications Conference, Honolulu, Hawaii (2009)
20. Shabtai, A., Fledel, Y., Elovici, Y.: Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In: International Conference on Computational Intelligence and Security, Nanning, China (2010)
21. Chaudhuri, A.: Language-Based Security on Android. In: Proceesings of the ACM Workshop on Programming Languages and Analysis for Security, pp. 1–7 (2009)
22. Mulliner, C., Miller, C.: Fuzzing the Phone in your Phone, Black Hat USA (2009)