Introduction to Android Security and Malware

Intern - Arthur Miller
Mentors - Kurt Derr, Sam Ramirez
Idaho National Laboratory

***Abstract*—Android is the world's most popular mobile operating system (OS), and bring your own device programs are on the rise for government agencies. Increased interest in using the Android OS in this information secure context, has made security on these devices a top concern. This paper will include information gathered on the Android security model; including the Device Administrator API and Android's recent implementation of SELinux. Two known malicious applications were used to test Android's security capabilities. An experiment was also initiated that extracted contents of RAM to acquire an encrypted application's data. This project has shown the lack of effectiveness of Android malware scanners', and that prevention of installation of malware lies on the user of the device. Once malware is installed user information can be obtained, and hardware components can be controlled by the malicious app. Secure application data can be breached if the app is able to gain system or root level permissions; these permissions also allow for an encrypted application's data to be retrieved. Implementation of SELinux does give Android the tools to reduce ramifications of a system or root exploit. Several other security features have also recently been added to the Android OS that help mitigate the damage malware can do on a device.**

I. How it Effects the Department of Energy (DOE)

Implementation of BYOD programs for Government agencies has become a top priority. In 2012 the United States Federal Government released the document *A Toolkit to Support Federal Agencies Implementing Bring Your Own Device (BYOD) Programs* [5]. Interest in Android devices being used in these programs was a key reason for the NSA's work on SELinux for Android [21]. For the DOE to implement Android devices in one of these programs, security must be a top concern. To ensure national security, secured data must be protected against malicious threats. This includes malicious threats attacking mobile devices that have access to intellectual property and other Government assets. To assure the DOE's BYOD implementation stands up to this standard, mobile device security on operating systems such as Android must be thoroughly examined and tested. This project is a good starting point, but research into Android vulnerabilities needs to continue.

II. Introduction

Android OS is the most popular mobile platform in the world [8]; which also makes it the largest target for mobile malware and other security threats [19]. Android's open source nature makes it easy for any developer to create an application (or simply app) and deploy it through the Google Play Store, a third party app store (such as Amazon, Samsung, GetJar, etc.), through email, or another side loading technique [18, 2]. With these dynamics, security against malware is a serious concern. There are two questions that must be asked.

- What can Android do to prevent malicious apps from being installed?
- What features are in place to mitigate damages done by malware that is installed?

The goal of this project is to shed light on these questions. By seeing what security features are in place to prevent malicious applications from being installed, as well as testing these features. And analyzing what the Android OS can do to minimize the damage of a malicious application that is installed. Two malicious apps were installed, testing Android's built-in malware scanner. After installed the malicious apps were used to examine what user information could be extracted with only app level permissions. Speculation of what permissions a system exploit would give to an app was also taken into consideration. As well tests were completed on devices with root level privileges that could be exploited by the malicious apps.

These are the the materials that were used throughout this project. Ubuntu 12.04LTS was the host operating system. Wireless testing was completed on a private Wi-Fi network. WSO2 Enterprise Mobility Manager (v 1.1.0) was the open source mobile device management (MDM) and mobile application management (MAM) system implemented. The Android test devices used were a non-rooted Galaxy Nexus (v4.3), a non-rooted Nexus S (v4.0.4), a rooted Galaxy Nexus (v4.2.2), and a rooted Galaxy Note I (v4.1.2). The Eclipse integrated development envirnment (IDE) (vJuno) was used to compile and export the applications tested. Eclipse was also used to run the server side graphical user interface (GUI) for the AndroRAT malware. VirtualBox was used with an image of Kali Linux to run the Android Metasploit malware attack. The tool used to extract RAM contents was the Linux Memory Extractor (LiME). Cross-compiling for the kernel and kernel modules was done using the Android NDK. A list of jargon associated with this topic can be found in Appendix A.

The rest of the paper will be laid out as follows: section III will cover Android's security model, sections IV, V, and VI will cover experiments that show concrete examples of malware capabilities, and the remaining sections will give a big picture of the work completed as well as possible future work.

III. Android Security Model

A. Background

Google acquired Android Inc. in August of 2005 and the first commercially available smart-phone running Android was released in October of 2008 [1, 9]. Popularity grew rapidly and in less than five years of commercial sales it was the world's

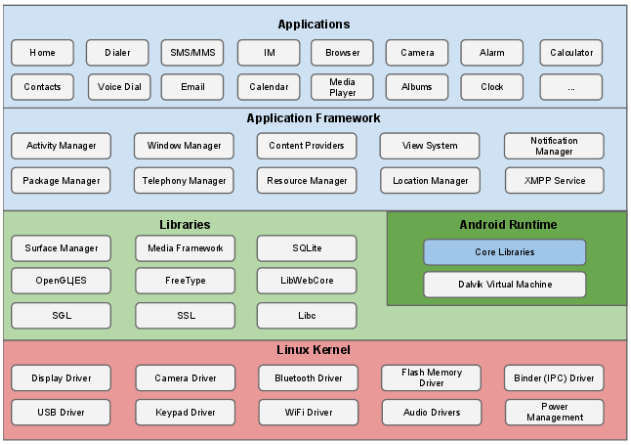

Fig. 1. *This figure is the Android security stack [15]*

most popular mobile OS. Google programmer's develop the OS in house, and then release their work to the Android Open Source Project [16]. After released anyone can view the OS source code, and device manufactures (e.g., Samsung, HTC, LG, etc.) may modify and use the OS for their devices [16]. This allows for a four tier security program [15].

1) Design Review
2) Penetration Testing and Code Review
3) Open Source and Community Review
4) Incident Response

B. Introduction

Android uses a two layer security model. The first layer which is in user space, is its user-granted permissions system. The second layer is application isolation (or sandboxing). Enforcement of both layers is done at the kernel level. To help with enforcing the application sandbox, Android requires each application to be signed by the developer. Applications also have to adhere to strict inter process communication that is handled by the kernel. The Device Administrator API is another security feature in user space. It can be used by a system administrator to control certain aspects of deployed devices (i.e., for a BYOD program). Android has also recently added SELinux as an added security resource in the kernel space [15].

C. Permission System

Android uses an application-requested user-granted permissions system. An application will request permission for certain restricted system resources. Such as internet access, use of the camera or other hardware components, access to private app data, and many other resources. At install time a list of all permissions that an application is requesting will be displayed on the screen. For the app to be installed the user must first grant these permissions - then the installation process will start. This system gives that the user the final say in whether an application is trusted enough to be allowed certain permissions [15].

D. Application Sandboxing

After an application is installed, application sandboxing is the next layer of defense. Sandboxing is the confining of each application to its own secure area and restricting its access to other applications data or system resources. This is achieved by first starting each application on its own process, and assigning each app its own unique user identifier (or UID). Unlike a traditional Linux system, where permissions are assigned on a per user/group basis, Android assigns permissions on a per application basis. When an application is started it is ran on its own instance of the *Dalvik Virtual Machine* (VM) — preventing poor memory management by one application from affecting other apps or even the OS. Inter process communication (IPC) is handled primarily through *Binder* — which is an Android specific kernel module (communication occurs through Intents, Services, and Content Providers in user space) [15]. There are two methods of communication for sharing private data between applications.

- An application can explicitly allow for sharing of information, then other apps can ask the system for access to the information.
- The second method requires that both applications be *signed* by the same developer.

To install an application on an android device the app must be *signed* with a unique certificate. This allows Android to identify an app to a specific developer. If two applications signed by the same developer are on a device, the system will allow communication between the two [15].

All of these security measures are enforced at the kernel level. Vulnerabilities in the kernel make the entire security system vulnerable [15]. Android has recently taken on the challenge of mitigating these risks by implementing SELinux, this will be examined in greater detail in subsection III-F.

E. Device Administrator API

Android Froyo (v2.2) added the Device Administrator API. This allows for applications to be created and implemented that can help system admins secure company assets. It gives the ability for admins to ensure that devices have a password set, and allows for enforcing specific password quality requirements. It also allows for admins to require full device encryption. Being able to dynamically disable the camera on a device and performing a data wipe are also features included. These are the current capabilities to the API, limitations for real world implementation will be discussed in the IV-A3 subsection [3].

F. SELinux

1) Background: SELinux is a loadable kernel module that can be added to the Linux kernel through its Linux Security Module (LSM) framework [10]. It was originally created by the National Security Agency (NSA) and later the software merged into the mainline Linux kernel starting with version 2.6 [10]. It allows implementation of a fine grained *mandatory access controls* (MAC) permission model, rather than the default discretionary access controls (DAC) that Linux

uses. Mandatory access controls allow an administrator of a system to define how applications and users can access different resources such as files, devices, networks and inter-process communication [20]. Android's port of the Linux kernel had always used the DAC model [23]. The NSA released a reference implementation of SELinux for Android to the open source development community in January of 2012 [21]. Immediately following, it was committed to the Android Open Source Project [21]. The 4.3 update of Jelly Bean implemented SELinux in permissive mode, which logs policy based information but does not enforce it [21]. The latest release KitKat (v4.4) uses SELinux in enforcing mode as an addition to its DAC system [21].

2) Limitations: There are several things that SELinux for Android can not prevent. Kernel vulnerabilities in general are beyond the scope of SELinux, because it relies on a secure kernel — although it may block exploitation of certain vulnerabilities. It also can not prevent exploitation of any weaknesses in the policy that is implemented [22].

3) Capabilities: Even with these limitations, there are several key capabilities that SELinux's kernel MAC does offer to help secure the Android OS. It can protect against the miss-use of privileged daemons by confining their privileges, thereby limiting the damage that can be done by them. This is one way it can defend against privilege escalation attacks. It also re-enforces Android's application sandbox. This is done by adding hard-coded policies for application permissions and communication rules - on top of Android's DAC model. There by limiting vulnerabilities that can be exploited by a malicious application. Middleware MAC can take Android's permission system out of the users hands and put it into the policy righter's. A policy could be written for SELinux that only allows installation of applications with specific signatures or applications with certain permissions. This can help enforce organizational security goals through an enterprise mobility manager system. To summarize, SELinux for Android helps prevent and mitigate root exploits, re-enforce the application sandbox, and gives organization's more control over its employees devices [22].

4) Android Implementation: The Android OS uses a unique Linux kernel, with modifications made for the mobile platform. These modifications required revisions to the standard version of SELinux to get it working with Android. It also required modifying Android kernel space components to incorporate SELinux's kernel MAC functionality. In Android's security stack, almost everything above the kernel is different then standard GNU Linux distributions. This meant SELinux had to be built from the ground up for the user space. One reason for this was Android implemented its own means of inter process communication (Binder) that would have required a complete change in Android source code to implement SELinux in the user space. SELinux for Android incorporates a separate Middleware MAC (MMAC) layer to solve this problem. The MMAC layer should only interact with the kernel MAC layer to determine policy. SELinux's effectiveness ultimately relies on the policy that is in place. A generic SELinux policy for standard Linux distributions is available. The generic policy however could not be used for Android. Due once again to how different Android's user space is than that of standard Linux distributions. This meant Android's policy had to be created from scratch as well [23].

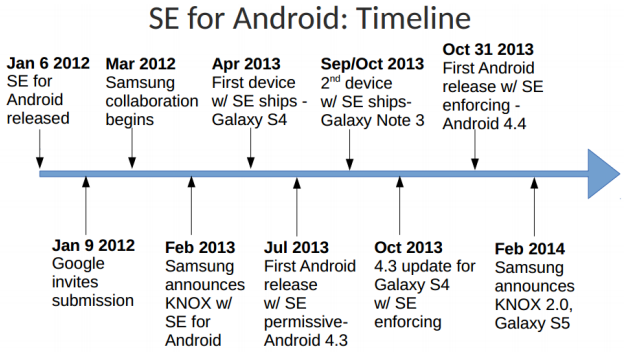

Fig. 2. This figure is a timeline of SELinux for Android [21]

IV. Testing the Device Administrator API Capabilities

A. Implementing WSO2 Enterprise Mobility Manager

1) Setup: WSO2 Enterprise Mobility Manager (EMM) uses the Device Administrator API along with other standard app permissions for the MDM and MAM system. Implementing WSO2 EMM involved first installing and configuring the server which integrated a GUI for the device administrator. The Android client application was then configured and exported as an .APK file. When the server and client were both configured and ready to deploy, the application was installed onto the Android device. Once the device was connected to the server and had granted the application Device Administrator privileges, testing could be done. To test the MDM system, a policy was configured and then assigned to a user. Enforcement of this policy was then tested on the device. The MAM system also comes with a web based mobile application store — applications such as Viber were added to the store and downloaded by the client device.

2) Capabilities: The WSO2 MDM system successfully deployed the policy onto the device. It allowed a device administrator to remotely *monitor* the device through a browser GUI. This included device information, fine grained gps location of the device, and a list of 3rd party applications that had been installed. A full list of features is included at the end of this subsection. The application store for WSO2 EMM was a web based store that connected to the EMM server. An application could be loaded into the MAM store, and then manually reviewed. Once the app was approved, the user was allowed to install it on the device — through the web based store. A known messaging application named Viber was added to the store by its package id. When a user chose to install Viber, it redirected them to the Google Play Store and continued the installation there. Applications could also be added that were not available through the Play Store, but testing was not completed for this method.

3) Limitations: Policies were not *enforced* effectively by WSO2 EMM. Once a policy was sent to a device, certain features like encryption were recommended to the user —

TABLE I
WSO2 EMM FEATURES

	1	2	3	4
Operations	Password Policy	Set Password	Lock Device	Storage Encyption
Operations	Toast Message	Wi-Fi Settings	Mute	
Device Info	Device Name	IMSI	Model	IMEI
Device	Battery Percent	Storage		
Other	GPS Location	Installed Apps		

not enforced. Password policies only mandated what the structure of the password was (e.g., length, special characters, expiration, etc.), it did not force the user to initiate a password or to maintain one. If the policy was not followed the device administrator was never notified. These are limitations due to the programming of the EMM solution. There were also severe limitations due to Android's permission model. Added privileges to a MDM and MAM solution such as this one is through the Device Administrator API. As covered in subsection III-E, the API adds only a few features. These limited capabilites are not enough to implement a full featured MDM and MAM solution. Several features such as controlling the Wi-Fi network a device can connect to and tracking the GPS location were added through normal permissions. Features listed in the chart above are the only useful features for a MDM and MAM solution that Android allows enforcement on — without extended privilege (i.e., the system). Yet top MDM and MAM solutions, such as AirWatch, BlackBerry, IBM, etc., boast the ability to whitelist or blacklist applications and even set access point name (APN) settings [28]. These capabilities are only allowed to system applications [4]. The reason these companies can use these features, is because their apps are signed by the original equipment manufacturer's signature (e.g., Samsung, Motorola, LG, etc.) [7]. This gives them same UID as the system and thus the same privilege as the system. With this added privilege more detailed enforcement policies can be put in place.

V. TESTING ANDROID'S SECURITY MODEL WITH MALWARE

A. Malware Scanners

Android's Jelly Bean (v4.2-4.3) update included a built in malware scanner [17]. Android's built-in malware scanner ("Verify Apps") as well as several top scanner in the Play Store were tested. AndroRAT and the Android Metasploit applications were installed on the non-rooted Galaxy Nexus (v4.3) for this experiment. The first test was to have both applications pre-installed on the device, and then install the scanner application to scan for malware. The malware applications were then re-installed. At install time the scanners

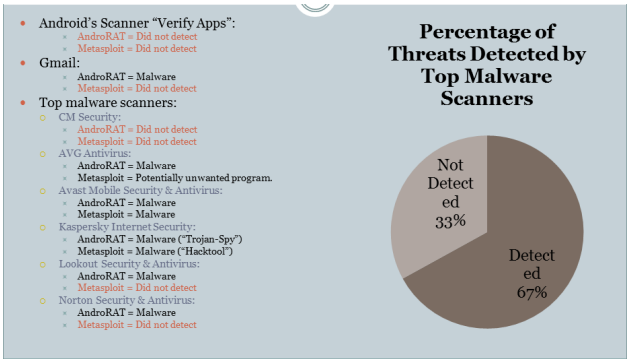

Fig. 3. This figure shows the results for malware installation testing

would have to detect the malware. Android's built in scanner will scan each application that is installed from an "Unknown Source" (i.e., not from the Play Store) — which for this experiment was every installation of both apps [17]. It should also detect malware after installation time, by periodically scanning installed applications [17]. Android's scanner did not detect any malware from the various installs. Interestingly, Gmail determined AndroRAT to be a virus and would not allow an email with it as an attachment to be sent. A list of the experiment's results (Fig. 3).

B. Malware Capabilities

After installation both of the malicious apps needed to be opened to communicate with the remote server. Each malware application had a set of capabilities that exploits android's permission system to spy on the user. These capabilities were tested to confirm that they worked properly. Android's permission system is enforced such that after the user grants an application permissions, they will have those permissions until they are un-installed. That knowledge bundled with the list of permissions available to an application, is enough to know the theoretical damage that can be done by installed malware. These two applications give concrete examples of the damage that can be done if permissions are granted and misused by malicous apps.

1) AndroRAT Attack: After AndroRAT is executed it runs either as a service running in the background or a blank activity. With either configuration, the user will not be notified of its malicious actions or be able to easily recognize that it is running. It also has permission to run at startup when the device is rebooted. It has internet permissions so it can connect to its remote server to wait for a command and then feed information back. It can gather the following information: the device IMEI, phone number, network carrier, Wi-Fi information, text messages, call logs, and contacts. It can monitor phone calls and text messages in real time. It can send text messages as well as make phone calls. It can take pictures and record video from both cameras and use the microphone to record audio. It can access GPS or network location of the device (this feature was not working). It can view and download files stored on the SD card of the device. It can also toast a message to the screen and vibrate the phone.

It is worth mentioning one potential use for the AndroRAT attack. A common malware technique has been to send Premium SMS (see Appendix B for definition) messages from an infected device. The deployer of the malware can take the money that was sent to the Short Codes — as a quick money making scheme. They were able to hide these messages from the user. This security risk was addressed in the Jelly Bean update, and now if a Premium SMS is sent to a Short Code the user will be notified by the OS [17].

2) Android Metasploit Attack: The Android Metasploit application runs through a terminal window. A remote shell connection with the device can then be made. The person using the remote shell can view the filesystem of the infected device. Files can be added and removed from folders with read/write permissions. And if the infected device is rooted, the entire filesystem can be viewed and edited. Even without root, read only files can be displayed on the screen or downloaded. Applications could be uploaded to the device, installed, and opened all through this remote shell. A list of running processes could be viewed and the UID that the shell is running from could be obtained.

With the exclusion of a few features being broken, most likely due to programming, the malware's successfully achieved their goals. Once installed they sent sensitive information about the device and its user back to a remote server. They could view and manipulate the filesystem and also were able to take control of certain hardware components. The permissions taken advantage of by these malware's are the most common permissions exploited by malware.

C. System Exploits

System exploits can be as powerful an attack by malware as a root exploit. Apps signed with system privileges have access to apps, app data, passwords, and system settings. With this access a malicious app can gain almost any information wanted about the device, user, or applications. The *Master Key* exploit and its derivatives are examples of this type of attack [7].

VI. Acquiring Encrypted Application Data

A. Extracting RAM With A Loadable Kernel Module

A malicious app can only retrieve information from another application's sandbox if they have elavated their privilege (or if they have the same signature). Assuming that an application has obtained root or system privileges, an application that has encrypted its data and securely stored its encryption key can still be vulnerable. This vulnerability is because while an application is being used, memory is stored in RAM unencrypted [14]. This experiment initiated a proof of concept that malware could utilize volatile memory forensics tools to extract a RAM dump file. Which then could be analyzed to retrieve sensitive information. To extract the contents of RAM from an Android device, several steps are necessary. First the kernel that the device is running must be known. This is because the kernel module that is need to extract the contents in RAM must be cross compiled with the exact kernel running on the device. After successfully cross compiling the Linux Memory Extractor (LiME), the newly created kernel module needed to be placed into the devices sdcard. From a root shell of the device the kernel module must be loaded onto the kernel. At this point a command can be ran to extract the RAM contents. The RAM dump file needs to be removed from the device so it can be analyzed. A tool such as Volatility could be used on a PC to analyze the RAM dump [13]. The experiment ran during this project was able to successfully acquire the RAM dump file from an android virtual machine, but analyses of the contents was not completed.

VII. Conclusion

Android's security enhancements, especially since the Jelly Bean update, have tightened down security for both the consumer and corporate users. If a device is not rooted (and malware cannot perform a privilege escalation attack to gain root) application data is secure. The biggest concern is spyware which if installed can gain information about the device and user. With little confidence can the responsibility of preventing malware installation be put on malware scanners. Especially not Android's built in scanner, but even third party scanners were not effective against the malicious applications tested. If all these measures fail and an application can get installed and gain root (or system) privileges most Android devices do not have security measures in place to keep private data secure. Even if an application encrypts its data it is still possible to acquire unencrypted data that has been stored in RAM. KitKat, the latest update does bring the tools, through SELinux, to help prevent privilege escalation attacks and mitigate the damages if they occur.

VIII. Future Work

This research topic has shed light on several other projects that could be explored further. The current project was a general examination of Android security, these ideas for future work will be a specific topic relating to Android security that has been brought to lite through this project.

A. Complete malware proof of concept

Continuing the work started in this project, a malicious application could be made to show Android's vulnerability to a root exploitation. This would include meshing aspects of each malicious attack that was tested. To created a malicious app that can remotely check for root (or escalate to root privilege), once determined or gained the app will upload the kernel module to the device and extract a ram dump file. This RAM dump file will be downloaded to a remote server and analyzed to find secured content. This is a very powerful concrete example of what damage can be done with root privileges on Android. This project could also be furthered to show other concrete examples of what malware can do once it has obtained root (i.e., viewing and editing non-encrypted private app data, changing APN settings, etc.).

B. Further research into SELinux

SELinux has only recently been added to the Android OS, thus there is very little information on its implementation. Some research has been done on the subject, but this information was mostly gathered before Android's implemented it in enforcing mode (v4.4). Understanding the strengths or weaknesses of Android's implementation of MAC and MMAC is necessary to understand Android's security for the next generation of devices. These policies are Android's defence against the privilege escalation attacks that have been disastrous to user's security.

C. Additional work with Android MDM/MAM system

With the release of Android's newest OS version (version L) also comes the promise of a more corporate friendly Android. They have released developer preview API's that add to the Device Administration API established in Froyo. These added tools will hopefully allow a product like WSO2 EMM to be a fully functioning MDM/MAM system. Both Samsung Knox API's and SELinux also have the potential to add even more control over a device through a MDM/MAM system. Extending the functionality of WSO2 EMM or even just familiarizing with the new API options could serve as useful research for the corporate setting.

D. Android malware scanning tool

Being able to quickly and accurately identify a malicious application before installation is an invaluable tool to have for any OS. There are several ways for this to be done. One is to compare hash tables for an already known application with an application that is proposed to be added to an application market (such as the one provided by WSO2 EMM). There are also methods which cross check an application with a large database of known malware. Another technique is heuristic scanning. This can be used at run time to check the application for abnormal behaviours or permissions, and then could report that information to a user. Or it could scan apps on a device and notify the user if behaviours out of the ordinary are occurring. Determining how to detect poly-morphic malware should also be examined in this project.

E. Malware installation testing

Understanding and implementing techniques of installing malware on a Android device. Testing different methods of installation will give a better understanding of how to prevent these types of attacks. This project could include binding malicious apps to other non-malicious apps, then repackaging and making available on a third party app market. Fully understanding how the Play Store is able to maintain a "reasonably" safe app market should be examined in detail as well. Other techniques such as drive-by downloads and in-app purchase drive by downloads could be implemented and examined. Examining poly-morphic methods of hiding an applications malicious intent could also be further explored.

REFERENCES

[1] Bloomberg. *Google Buys Android for Its Mobile Arsenal*. 2011. URL: http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal.

[2] Android Developers. *Alternate Distribution Options*. 2014. URL: http://developer.android.com/distribute/tools/open-distribution.html.

[3] Android Developers. *Device Administration API*. 2014. URL: http://developer.android.com/guide/topics/admin/device-admin.html.

[4] Android Developers. *Reference: Manifest.permission*. 2014. URL: http://developer.android.com/reference/android/Manifest.permission.html.

[5] Federal Chief Information Officers Council Digital Services Advisory Group. *A Toolkit to Support Federal Agencies Implementing Bring Your Own Device (BYOD) Programs*. 2014. URL: http://www.whitehouse.gov/digitalgov/bring-your-own-device.

[6] FileInfo. *.APK File Extension*. 2014. URL: http://www.fileinfo.com/extension/apk.

[7] Jeff Forristal. "Android: One Root to Own Them All". In: *Black Hat USA 2013* (2013).

[8] Gartner. *Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time*. 2013. URL: http://www.gartner.com/newsroom/id/2573415.

[9] gizmodo. *T-Mobile G1: Full Details of the HTC Dream Android Phone*. 2008. URL: http://gizmodo.com/5053264/t-mobile-g1-full-details-of-the-htc-dream-android-phone.

[10] IBM. *Anatomy of Security-Enhanced Linux (SELinux)*. 2012. URL: http://www.ibm.com/developerworks/library/l-selinux/.

[11] LinuxInfo. *Kernel Space Definition*. 2005. URL: http://www.linfo.org/kernel_space.html.

[12] LinuxInfo. *User Space Definition*. 2005. URL: http://www.linfo.org/user_space.html.

[13] Holger Macht. "Live Memory Forensics on Android with Volatility". diploma thesis. Friedrich-Alexander University Erlangen-Nuremberg, 2013.

[14] Noa Bar-Yosef Michael Shaulov. "Anatomy of a Targeted Attack against Mobile Device Management (MDM)". In: *Black Hat USA 2013* (2013).

[15] Android Open Source Project. *Andorid Security Overview*. https://source.android.com/devices/tech/security/. 2014.

[16] Android Open Source Project. *Codelines, Branches, and Releases*. https://source.android.com/source/codelines.html. 2014.

[17] Android Open Source Project. *Security Enhancements in Android 4.2*. 2014. URL: https://source.android.com/devices/tech/security/enhancements42.html.

[18] Android Open Source Project. *Welcome to the Android Open Source Project!* https://source.android.com/. 2014.

[19] SecureList. *Kaspersky Security Bulletin 2013. Overall Statistics for 2013*. 2013. URL: http://www.securelist.com/en/analysis/204792318/Kaspersky_Security_Bulletin_2013_Overall_statistics_for_2013.
[20] SELinuxProject. *What is SELinux really?* 2009. URL: http://selinuxproject.org/page/FAQ.
[21] Stephen Smalley. "Security Enhancements (SE) for Android". In: *Android Builders Summit 2014* (2014).
[22] Stephen Smalley. "The Case for Security Enhanced (SE) Android". In: *Android Builders Summit 2012* (2012).
[23] Robert Craig Stephen Smalley. *Security Enhanced (SE) Android: Bringing Flexible MAC to Android*. Tech. rep. National Security Agency, 2012.
[24] Techterms. *API*. 2014. URL: http://www.techterms.com/definition/api.
[25] Techterms. *Daemon*. 2014. URL: http://www.techterms.com/definition/daemon.
[26] Techterms. *Middleware*. 2011. URL: http://www.techterms.com/definition/middleware.
[27] Verizon. *What is Premium Messaging?* 2014. URL: http://www.verizonwireless.com/support/faqs/Premium_TXT_and_MMS/faq_premium_txt_and_mms.html.
[28] Computer World. *MDM tools: Features and functions compared*. 2014. URL: http://www.computerworld.com/s/article/9238981/MDM_tools_Features_and_functions_compared.

APPENDIX A
JARGON

Kernel space: Is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services [11].
User space: Is that portion of system memory in which user processes run. This contrasts with kernel space [12].
Middleware: The most common type of middleware is software that enables two separate programs to communicate and share data [26].
Daemon: A computer daemon is a constantly running program that triggers actions when it receives certain input [25].
API: Application Program Interface — An API is a set of commands, functions, and protocols which programmers can use when building software for a specific operating system. The API allows programmers to use predefined functions to interact with the operating system, instead of writing them from scratch [24].
.APK file extension: Application package created for Android [6].

APPENDIX B
DEFINITION OF PREMIUM MESSAGE

Premium Messaging is an option to purchase or subscribe to messaging programs, provided by third party content providers, for premium charges (e.g., charges that are in addition to standard messaging charges). The premium charges for subscriptions recur monthly, while the premium charges for purchases occur only once. Many programs offer both one-time purchases and recurring subscriptions. These programs are initiated through special numbers, which are four, five or six-digit numbers, known as Short Codes. [27]