# Chapter 1
# Android Security, Pitfalls and Lessons Learned

Steffen Liebergeld, Matthias Lange

**Abstract**

Over the last two years Android became the most popular mobile operating system. But Android is also targeted by an over-proportional share of malware. In this paper we systematize the knowledge about the Android security mechanisms and formulate how the pitfalls can be avoided when building a mobile operating system.

## 1.1 Introduction

Smartphones are now very popular. Aside from calling and texting, people use them for connecting with their digital life–email, social networking, instant messaging, photo sharing and more. With that smartphones store valuable personal information such as login credentials, photos, emails and contact information. The confidentiality of that data is of paramount importance to the user because it might be abused for impersonation, blackmailing or else. Smartphones are very attractive for attackers as well: First, attackers are interested in the precious private information. Second, smartphones are constantly connected, which makes them useful as bots in botnets. Third, smartphones can send premium SMS or SMS that subscribe the victim to costly services, and thus directly generate money for the attacker. It is up to the smartphone operating system (OS) to ensure the security of the data on the device. In the last two years Android became the most popular mobile OS on the market. With over 1.5 million device activations per day Android is expected to cross the one billion active device barrier in 2013. Its world wide market share has reached 70 percent of all smartphones. On the downside Android also became a major target for mobile malware [19]. Interestingly the share of mobile malware that targets Android is around 90 percent, which

is larger than its market share. The question is why is the Android platform so attractive for malware authors? In this paper we investigate the Android architecture and the security mechanisms it implements. Android and its weaknesses have already been well researched and we systematize the results and give advice for platform designers to avoid those pitfalls in the future. Our contributions are:

**Android security mechanisms:** We describe the Android architecture from a security point of view and give details on application and system security. We further detail the mechanisms of Android that are targeted at fending off attacks.

**Android security problems:** We identify the inherent security problems of the Android platform.

## 1.2 Android Platform Security

Android runs on a wide range of devices and Android's security architecture relies on security features that are embedded in the hardware. The security of the platform depends on a secure boot process.

**Secure Boot** The boot process of an Android device is a five-step process. First the CPU starts executing from its reset vector to which the initial bootloader (IBL) code from the ROM is wired. Then the IBL loads the bootloader from the boot medium into the RAM and performs a signature check to ensure that only authenticated code gets executed. The bootloader loads the Linux kernel and also performs a signature check. The Linux kernel initializes all the hardware and finally spawns the first user process called *init*. Init reads a configuration file and boots the rest of the Android user land.

**Rooting** In general, mobile devices are subject to strict scrutiny of the mobile operators. That is it employs secure boot to ensure that only code is being booted, that has received the official blessing in the form of a certification from the operators. This is being done to ensure that the mobile OS's security measures are implemented and the device does not become a harm to the cellular network.

Rooting involves a modification to the system partition. Modifications to the system partition require root permissions, which are not available by default. There are two ways of obtaining root permissions: Either the customer boots a custom system that gives him a root shell, or he exploits a vulnerability to obtain root permissions at runtime.

Rooting, voluntarily or involuntarily has repercussions on device security. Unsigned kernels can contain malware that runs with full permissions and is undetectable by anti-virus software (*rootkits*). Further, rooted devices do not receive over the air updates. If an application has received root permissions, it can essentially do as it pleases with the device and its data, including

copying, modifying and deleting private information and even bricking the device by overwriting the bootloader.

## 1.3 Android System Security

The flash storage of an Android device is usually divided into multiple partitions. The system partition contains the Android base system such as libraries, the application runtime and the application framework. This partition is mounted read-only to prevent modification of it. This also allows a user to boot their device into a safe mode which is free of third party software.

Since Android 3.0 it is possible to encrypt the data partition with 128bit AES. To enable filesystem encryption the user has to set a device password which is used to unlock the master key.

**Data Security** By default an application's files are private. They are owned by that application's distinct UID. Of course an application can create world readable/writable files which gives access to everybody. Applications from the same author can run with the same UID and thereby get access to shared files. Files created on the SD card are world readable and writable. Since Android 4.0 the framework provides a Keychain API which offers applications the possibility to safely store certificates and user credentials. The keystore is saved at `/data/misc/keystore` and each key is stored in its own file. A key is encrypted using 128-bit AES in CBC mode. Each key file contains an info header, the initial vector (IV) used for the encryption, an MD5 hash of the encrypted key and the encrypted data itself. Keys are encrypted using a master key which itself is encrypted using AES.

## 1.4 Android Application Security

In Android application security is based on isolation and permission control. In the picture you can see, that there are processes that run with root privileges. Zygote is the prototype process that gets forked into a new process whenever a (Java) application is launched. Each application runs in its own process with its own user and group ID which makes it a *sandbox*. So, by default applications cannot talk to each other because they don't share any resources. This isolation is provided by the Linux kernel which in turn is based on the decades-old UNIX security model of processes and file-system permissions. It is worth noting that the Dalvik VM itself is not a security boundary as it does not implement any security checks. In addition to traditional Linux mechanisms for inter-process communication Android provides the *Binder* [8] framework. Binder is an Android-specific IPC mechanism and remote method invocation system. Binder consists of a kernel-level driver and

a userspace server. With Binder a process can call a routine in another process and pass the arguments between them. Binder has a very basic security model. It enables the identification of communication partners by delivering the PID and UID.

**Android Permissions** On Android services and APIs that have the potential to adversely impact the user experience or data on the device are protected with a mandatory access control framework called *Permissions*. An application declares the permissions it needs in its AndroidManifest.xml[1] such as to access the contacts or send and receive SMS. At application install time those permissions are presented to the user who decides to grant all of them or deny the installation altogether. Permissions that are marked as *normal* such as wake-up on boot are hidden because they are not considered dangerous. The user however can expand the whole list of permissions if he wants to.

**Memory Corruption Mitigation** Memory corruption bugs such as buffer overflows are still a huge class of exploitable vulnerabilities. Since Android 2.3 the underlying Linux kernel implements `mmap_min_addr` to mitigate null pointer dereference privilege escalation attacks. `mmap_min_addr` specifies the minimum virtual address a process is allowed to mmap. Before, an attacker was able to map the first memory page, starting at address 0x0 into its process. A null pointer dereference in the kernel then would make the kernel access page zero which is filled with bytes under the control of the attacker. Also implemented since Android 2.3 is the eXecute Never (XN) bit to mark memory pages as non-executable. This prevents code execution on the stack and the heap. This makes it harder for an attacker to inject his own code. However an attacker can still use return oriented programming (ROP) to execute code from e.g. shared libraries. In Android 4.0 the first implementation of address space layout randomization (ASLR) was built into Android. ASLR is supposed to randomize the location of key memory areas within an address space to make it probabilistically hard for an attacker to gain control over a process. The Linux kernel for ARM supports ASLR since version 2.6.35. The Linux kernel is able to randomize the stack address and the brk memory area. The brk() system call is used to allocate the heap for a process. ASLR can be enabled in two levels by writing either a 1 (randomize stack start address) or a 2 (randomize stack and heap address) to `/proc/sys/kernel/randomize_va_space`. In Android 4.0 only the stack address and the location of shared libraries are randomized. This leaves an attacker plenty of possibilities to easily find gadgets for his ROP attack. In Android 4.1 Google finally added support for position independent executables (PIE) and a randomized linker to fully support ASLR. With PIE the location of the binary itself is randomized. Also introduced in Android 4.1 is a technique called read-only relocation (RELro) and immediate binding. To locate functions in a dynamically linked library, ELF uses the global offset

[1] There are more than 110 permissions in Android. A full list is available at `http://developer.android.com/reference/android/Manifest.permission.html`

table (GOT) to resolve the function. On the first call a function that is located in a shared library points to the procedure linkage table (PLT). Each entry in the PLT points to an entry in the GOT. On the first call the entry in the GOT points back to the PLT, where the linker is called to actually find the location of the desired function. The second time the GOT contains the resolved location. This is called lazy-binding and requires the GOT to be writable. An attacker can use this to let entries in the GOT point to his own code to gain control of the program flow. RELro tells the linker to resolve dynamically linked functions at the beginning of the execution. The GOT is then made read-only. This way an attacker cannot overwrite it and cannot take control of the execution.

## 1.5 Android Security Enhancements

With Android 4.2 and the following minor releases Google introduced new security features in Android. We will present a small selection of these enhancements in the following paragraphs. The user now can choose to verify side-loaded applications prior to installation. This is also know as the on-device Bouncer. It scans for common malware and alerts the user if the application is considered harmful. So far the detection rates don't measure up with other commercial malware scanners [5]. With Android 4.2.2 Google introduced secure USB debugging. That means only authenticated host devices are allowed to connect via USB to the mobile device. To identify a host, adb generates an RSA key pair. The RSA key's fingerprint is displayed on the mobile device and the user can select to allow debugging for a single session or grant automatic access for all future sessions. This measure is only effective if the user has a screen lock protection enabled. Prior to Android 4.2 the optional `exported` attribute of a Content Provider defaulted to true which hurts the principle of least privilege. This lead to developers involuntarily making data accessible to other apps. With Android 4.2 the default behaviour is now "not exported".

**SELinux on Android** The SEAndroid project [15] is enabling the use of SELinux in Android. The separation guarantees limit the damage that can be done by flawed or malicious applications. SELinux allows OS services to run without root privileges. Albeit SELinux on Android is possible it is hard to configure and it slows down the device. Samsung Knox has been announced to actually roll-out SEAndroid on commercial devices.

## 1.6 Android Security Problems

According to F-Secure Response Labs 96% of mobile malware that was detected in 2012 targets the Android OS [11]. In this chapter we want to shed light on the security weaknesses of Android that enabled such a vibrant market of malware. In short, Android has four major security problems: First, security updates are delayed or never deployed to the user's device. Second, OEMs weaken the security architecture of standard Android with their custom modifications. And third, the Android permission model is defective. Finally, the Google Play market poses a very low barrier to malware. We will now detail each of these problems.

**Android Update Problem** There are four parts of the system that can contain vulnerabilities: the base system containing the kernel and open source libraries, the stock Android runtime including basic services and the Dalvik runtime, the Skin supplied by the OEM and the branding. The Android base system and runtime are published with full source by the AOSP. This code is the basis of all Android based smart phones. Any vulnerability found therein can potentially be used to subvert countless Android devices. In other terms, a vulnerability has a high *impact*. In practice, updates are very slow to reach the devices, with major updates taking more than 10 months [3]. Many vendors do not patch their devices at all, as the implementation of a patch seems too costly [4]. According to Google Inc.'s own numbers, the most recent version of Android is deployed to only 1.2% of devices [2]. To remedy this problem, Google announced an industry partnership with many OEM pledging to update their devices for 18 months. This partnership is called the *Android Update Alliance*. However, there has been no mentioning of the alliance since 2012, and updates are still missing [3]. Bringing the updates to the devices is more involved however. Once the update reaches the OEMs, they incorporate it into their internal code repositories. For major updates, this includes porting their Skin forward. A faulty firmware update has very bad consequences for the OEM's reputation. Therefore the updated firmware is subject to the OEM's quality control. In summary, incorporating an update into a device firmware is therefore very costly to the OEM both temporal and financial. Cellular operators certify devices for correct behaviour. This is done to ensure that the device does not misbehave and therefore does not put the network and its users at risk. Updated firmwares need to be re-certified before they can be deployed. Depending on the operator this can take a substantial amount of time. For example re-certification at T-Mobile takes three to six months [12], other carriers opt out of the process and do not ship any updates at all.

**Android Permission Model** The Android permission model has been under criticism since Android was introduced. It has been extensively studied by researchers. Here we present the problems that stand out. Kelley et al. conducted a study and found that users are generally unable to understand and reason about the permission dialogues presented to them at application

installation time [18]. In [16] Barrera et al. conducted an analysis of the Android permission model on a real-world data set of applications from the Android market. It showed that a small number of permissions are used very frequently and the rest is only used occasionally. It also shows the difficulty between having finer or coarser grained permissions. A finer grained model increases complexity and thus has usability impacts. The study also showed that not only users may have difficulties understanding a large set of permissions but also the developers as many over-requesting applications show. Felt et al. performed a study on how Android permissions are used by Apps. They found that in a set of 940 Apps about one-third are over-privileged, mostly due to the developers being confused about the Android permission system [17]. Another problem are combo permissions. Different applications from the same author can share permissions. That can be used to leak information. For example an application has access to the SMS database because it provides full text search for your SMS. Another app, say a game, from the same author has access to the Internet because it needs to load ads from an ad server. Now through Android's IPC mechanism those two apps can talk to each other and essentially leak the user's SMS database into the Internet.
**Insufficient Market Control** Anybody can publish her applications to the official Android App market *Google Play* after paying a small fee. There are alternative App markets, e.g. the Amazon Appstore [7] and AndroidPit [9], but Google Play is the most important one because it is preinstalled on almost any Android device. Any App that is published via Google Play must adhere to the Google Play Developer Distribution Agreement (DDA) [13] and Google Play Developer Program Policies (DPP) [14]. However, Google Play does not check upfront if an uploaded App does adhere to DDA and DPP. Only when an App is suspected to violate DDA or DPP, it is being reviewed. If it is found to breach the agreements, it is suspended and the developer notified. If the App is found to contain malware, Google might even uninstall the App remotely. In 2012 Google introduced *Bouncer* [6]. Bouncer is a service that scans Apps on Google Play for known malware. It runs the Apps in an emulator and looks for suspicious behaviour. Unfortunately it didn't take long for researchers to show ways on how to circumvent Bouncer [1]. Malicious Apps have been found on Google Play repeatedly [10].

## 1.7 Lessons Learned

From our study of Android security problems we compile a set of lessons learned to educate future OP developers in avoiding these pitfalls.
**Timely security updates** are an absolute must for any secure system. This is especially important for open source systems where the code is public and bugs are easy to spot. For Smartphones an update system has to take all involved parties into account. We think that the key lies in clear abstractions

and a modular system. That would enable the cellular operators to certify a device by looking on the radio stack alone.
**Control platform diversity:** The OS designer should enforce that third party modifications to the OS do not introduce security breaches by design. He should enforce contracts on security critical points in the system that third party code has to follow. For example Google should enforce that any device running Android must only contain code that enforces the Android permission system.
**Ensure lock screen locks screen under all circumstances:** Ensure that no third party can mess with the lockscreen.
**Design permission system with user and developer in mind:** A permission system should be designed such that the permissions it implements are understood by both the developer to avoid over-privileged Apps and the user, so that she can make an educated decision when granting permissions. Granting all permissions at installation time is problematic. Users ofter grant permissions just to be able to install an App. Also, it does not allow for fine-grained permissions. Maybe a better solution would be to ask the user to grant permissions on demand.
**Ensure that the App market does not distribute malware:** The App market is the most important distribution place for Apps. People trust in the App markets, and have no chance to determine the quality of an App by themselves. Aside from having a mandatory admission process, an App market should also scan for repackaged Apps.

## 1.8 Conclusion

In this work we investigated the security of the Android mobile OS. We described the Android security measures, and its problems. We derived a set of lessons learned that will help future mobile OS designers to avoid pitfalls.

## 1.9 Acknowledgements

This work was partially supported by the EU FP7/2007-2013 (FP7-ICT-2011.1.4 Trustworthy ICT), under grant agreement no. 317888 (project NEMESYS).

## References

1. Adventures in BouncerLand: Failures of Automated Malware Detection within Mo-

bile Application Markets. http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf (July 2012)
2. Android Dashboard. https://developer.android.com/about/dashboards/index.html (December 2012)
3. Arstechnica: The checkered, slow history of Android handset updates. http://arstechnica.com/gadgets/2012/12/the-checkered-slow-history-of-android-handset-updates/ (December 2012)
4. Arstechnica: What happened to the Android Update Alliance? http://arstechnica.com/gadgets/2012/06/what-happened-to-the-android-update-alliance/ (June 2012)
5. An evaluation of the application verification service in android 4.2. http://www.cs.ncsu.edu/faculty/jiang/appverify/ (December 2012)
6. Google Mobile Blog: Android and Security. http://googlemobile.blogspot.de/2012/02/android-and-security.html (February 2012)
7. Amazon Appstore. http://www.amazon.com/mobile-apps/b/ref=sa_menu_adr_app?ie=UTF8&node=2350149011 (April 2013)
8. Android Developer Documentation: Binder. http://developer.android.com/reference/android/os/Binder.html (January 2013)
9. AndroidPit. http://www.androidpit.com/ (April 2013)
10. Arstechnica: More "BadNews" for Android: New malicious apps found in Google Play. http://arstechnica.com/security/2013/04/more-badnews-for-android-new-malicious-apps-found-in-google-play/ (April 2013)
11. F-Secure Mobile Threat Report Q4 2012. http://www.f-secure.com/static/doc/labs_global/Research/Mobile20Threat20Report20Q4202012.pdf (March 2013)
12. Gizmodo: Why Android Updates Are So Slow. http://gizmodo.com/5987508/why-android-updates-are-so-slow (March 2013)
13. Google Play Developer Distribution Agreement. http://www.android.com/us/developer-distribution-agreement.html (April 2013)
14. Google Play Developer Program Policies. http://www.android.com/us/developer-content-policy.html (April 2013)
15. Seandroid wiki. http://selinuxproject.org/page/SEAndroid (April 2013)
16. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 73–84. CCS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1866307.1866317
17. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. CCS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046707.2046779
18. Kelley, P., Consolvo, S., Lorrie, C., Jung, J., Sadeh, N., Wetherall, D.: An conundrum of permissions: Installing applications on an android smartphone. Workshop on Usable Security (2012)
19. Symantec: Internet security threat report. Tech. rep. (April 2013), http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf