Chapter 1
Android Security, Pitfalls, Lessons Learned and BYOD

Steffen Liebergeld, Matthias Lange

Abstract O
ver 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. As smartphones enter the corporate domain, a new scheme called bring your own device (BYOD) became popular. One solution is to logically partition the device such that personal and business information are isolated from one another. We systematize the solutions for partitioning in Android.

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 [38]. 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.

Recently companies started to allow the use of private smartphones in corporate networks. This scheme is commonly called *bring your own device* (BYOD). As these smartphones are being administered by the employees, who are not trained in security best practices, they often remain vulnerable, and thus put corporate assets at risk. BYOD mandates a new security feature, which we call *partitioning*. In partitioning, the device is logically split into isolated partitions. Android was not designed with partitioning in mind. We show how different solutions retrofitted partitioning into Android, and systematize the drawbacks and merits of each approach.

1.1.1 Contributions

In this work we systematize knowledge in the following areas:
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.
Android BYOD solutions: A number of BYOD solutions using Android have been proposed. We identify the different approaches and systematize them. We give information about drawbacks and merits of each approach.

Additionally, we detail *lessons learned* to help future mobile OS designers avoid security pitfalls.

1.1.2 Outline

We start by describing what Android is and how its architecture looks like in Section 1.2. Then we continue describing the platform and system security mechanisms in Section 1.3 and Section 1.4. Android application security is depicted in Section 1.5. In Section 1.6 we outline recent improvements in

Android security before we describe Android's most severe security problems in Section 1.7. Given this insight, we determine lessons learned in Section 1.8. Section 1.9 shows how partitioning was retrofitted into Android. We continue with a systematization of the Android virtualization solutions in Section 1.10. We conclude in Section 1.11.

1.2 Android Overview

Android is an OS and a software platform for mobile devices. Its development dates back into 2003 to a company called Android which developed software for mobile devices. In 2005 this company was bought by Google. In November 2007 Google announced together with 33 other members of the Open Handset Alliance that they will develop a mobile OS called Android [1]. One year later the first consumer device, the T-Mobile G1, became available [2]. Since 2010 Google sells their own mobile devices under the Nexus brand [3].

The Android development is in the hands of the Open Handset Alliance. The Android Open Source Project (AOSP) is the open source version of Android but in fact Google is the sole contributor. Usually Google develops Android internally and pushes its internal code base to public code repositories whenever they issue a public release. An exception was the release of Android 3.0 *Honeycomb* whose source code was never fully released[1].

The Android userland is licensed under the terms of the Apache Software License 2.0 which does not mandate source code availability [16]. This also allows OEMs to package Android with binary libraries which they are not forced to make open source. The underlying Linux kernel instead is licensed under the terms of the Gnu General Public License (GPLv2). That license requires that each modification or addition of code must be made available to the customers of a device with that software.

The source code of the AOSP project in its original form is deployed only to a selected set of devices. For each major release of Android Google partners with one OEM to create a device of the Nexus brand. The Nexus devices are made to showcase how Google envisions Android to be and usually receive updates to new versions of Android directly from Google.

OEMs modify the code taken from the AOSP and enhance it with their own custom code. They add new pre-installed applications, tweak the user interface and add additional functionality to stock applications to set each other apart. Taken together, these modifications to stock AOSP Android are called a *Skin*. Additionally, many carriers add custom applications to the devices they sell (*branding*).

[1] The sources of Honeycomb are included in the version history of subsequent releases, but the actual release was not marked with a tag, which makes it impossible to check out.

1.2.1 Android Architecture

The general Android architecture is depicted in Figure 1.1. At the bottom sits the Linux kernel which has been modified to accommodate for Android's special needs. In that sense Android is not a traditional Linux OS. It does not have a passwd file, no glibc and no X11, in fact nothing we assume to be part of a standard Linux distribution. The Linux kernel provides isolation, threading, scheduling and memory management. It also provides a driver model and a huge device driver base. As an addition Google added an IPC framework and power management enhancements to the kernel which are designed for the requirements of embedded devices. An interesting side note is, that the Android-specific code in the Linux kernel has twice the defect density of the core Linux kernel [34].

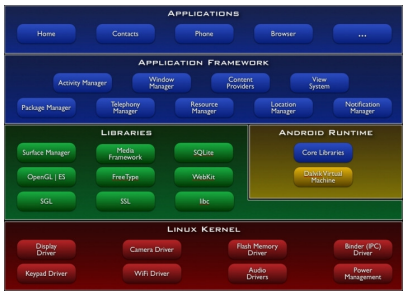

Fig. 1.1 Android system architecture, image courtesy of Google Inc [14].

Each major version of Android provides its own version of the Linux kernel. That is, older Android userland runs on newer kernels but not vice versa. Since mainline Linux version 3.3 some of Android's code modifications have been merged. This enables a mainline Linux kernel to boot Android [26].

The Android userland consists of three layers: the native layer, the application framework and the applications. The native layer implements a hardware abstraction layer (HAL) to abstract the device drivers. It interfaces directly with the Linux kernel. It is implemented in C and C++. This layer includes a number of open source libraries such as WebKit, libpng and libsqlite. Bionic is Google's equivalent of the libc. Also the native layer contains a set of native daemons which run as root.

The Dalvik VM, on top of which most of the applications are built, is also part of the native layer. Dalvik is a register-based process virtual machine and executes Dalvik Executable Format (DEF) code. On Android applications are mostly written in Java which is compiled to Java bytecode. The bytecode is compiled into DEF upon installation of the application on the device. The Dalvik VM is no security perimeter other than that applications written in

Java are not subject to memory corruption attacks. The task of isolation is left up for the Linux kernel.

The application framework is where the platform services such as location manager or package manager live. This layer gives the developers access to the lower levels of the platform.

At the very top there are the applications. They use services provided by the application framework. They may potentially use services from other applications as well. That means that applications can become part of the API which is an import consideration for Android security.

1.3 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.

1.3.1 Secure Boot

In general the boot process of an Android device is a five-step process. When the CPU is reset it will start executing from its reset vector. At the reset vector there is some ROM connected which contains the initial bootloader (IBL). The ROM is either fabricated into the system-on-chip (SoC) during production or the IBL is programmed into a programmable ROM. Its the IBL's duty to initialize the DRAM controller and the boot medium (MMC, eMMC, NAND flash, USB) controller. The boot medium usually is selected by the operating mode (OM) pin. Usually this pin is permanently set during production of the SoC. The IBL then loads the bootloader from the boot medium into the RAM. It then performs a signature check to make sure that only authenticated code is executed. If the check is successful the IBL hands execution to the bootloader. The signature check is performed using the public key of the original equipment manufacturer (OEM). The key is also stored in the ROM.

The bootloader offers more flexibility. It initializes more hardware like the display to show e.g. a boot logo. The bootloader performs a signature check on the Linux kernel.

The Linux kernel initializes all the hardware and finally spawns the first user space process called init. Init is stored on a ramdisk. The ramdisk is either merged in the kernel image (CPIO archive) or the Android boot image format is used. This ensures that also the root filesystem is signed and can be verified.

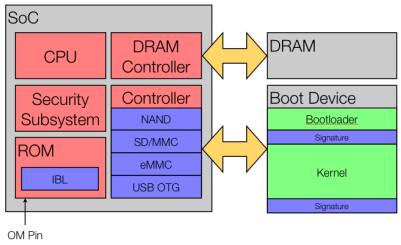

Fig. 1.2 Boot architecture on mobile devices. The OM pin determines the boot medium. The IBL initializes the DRAM and the boot medium controller and loads the bootloader into the RAM. The security subsystem may be used to speed up signature checks.

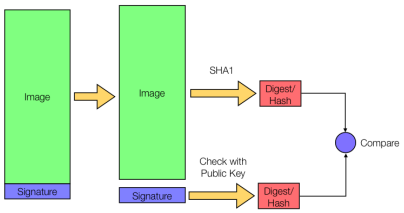

Fig. 1.3 Signature check: The image is hashed while the signature gets decrypted to return the original hash value. Both values then are compared.

Android init reads a configuration file (init.rc) and boots the user land. The configuration file contains information about the initial services to be started and their dependencies. This allows init to restart crashed services. Init does not verify user space components prior to loading.

1.3.2 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. The official firmware images however place restrictions on the device's capabilities. For example, some cellular operators disable tethering or only allow their own SIM cards (SIM lock).

Many users want to free their devices of such strict restrictions, for example to install their own Android distributions (ROM), e.g. CyanogenMod. There are many reasons for users to *root* their device such as reuse old hardware,

remove offending system apps, get better looks and get more speed. Also, a rooted device allows applications to run with root permissions.

Rooting involves a modification to the system partition. Because the system partition is mounted read only, it has to be re-mounted with read/write permissions. Re-mounting however requires root permissions. There are two ways of obtaining root permissions initially: Either the customer boots a custom system that gives him a root shell, or he exploits a vulnerability to obtain root permissions.

Booting a custom system involves unlocking the bootloader, that is to allow it to boot unsigned binaries. Many devices, such as Google's Nexus devices allow unlocking the bootloader out of the box. On other devices a bug in the bootloader has to be exploited to enable booting of unsigned binaries.

Once the bootloader is unlocked, a modified system is booted, that allows modification of the firmware image. The modified firmware image contains the *su* binary which allows any application to request root at any time. su then checks in a local database whether the requesting App's UID has been granted root privileges before. If not, su starts an activity which will prompt the user to confirm the elevated privileges. The user's decision is then stored into the database.

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.

1.4 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.

1.4.1 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 (signed with the same key, see Section 1.5.3 for more details) 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. The encryption key is derived from the user password using PBKDF2 with 8192 iterations.

1.4.2 Filesystem Encryption

Since Android 3.0 it is possible to encrypt the data partition. On the kernel-side this task is performed by *dm-crypt*. Because Google wanted to keep Android free of GPL code they decided to implement the necessary ioctls in *vold* instead of using the established dm-crypt userland component *cryptsetup*.

To enable filesystem encryption the user has to set a device password. This password is used to encrypt a 128 Bit master key. The master key is derived from /dev/urandom. To encrypt the master key the user password is hashed with a salt also taken from /dev/urandom. Finally the master key is encrypted using AES calls into the openssl library.

Vold is responsible for setting up the crypto mappings between a virtual crypto block device and the real block device. It then encrypts each sector as it is written and decrypts each sector as it is read. Details of the encryption of a block device are kept in the crypto footer which is kept in the last 16 Kbytes of each partition. The crypto footer for example contains the encrypted master key.

When the user changes his password not the whole partition needs to be re-encrypted but just the master key. The encryption and decryption can be speed up by using a crypto engine which is integrated into the SoC. This requires a proper driver in the Linux kernel which is integrated with dm-crypt.

1.4.3 Device Admin

In many companies, employees are allowed to access their corporate assets with their private smartphones. This scheme is known as *Bring Your Own Device* (BYOD). The problem with BYOD is that it introduces devices into

the corporate domain which are not under strict control by corporate IT in the first place. As smartphones are notoriously insecure, they may set corporate assets at risk. In exchange for an employee to use his own device enterprises usually want more control over the phone.

In Android 2.2 Google added an API called *Device Admin*. That API is for an application to request privileges to make that application the administrator of the device. Device admin allows an application to enforce certain policies such as require a lock password, password policies, monitor unlock attempts, require encryption and disable the camera. Through device admin a device can also be remotely wiped.

1.5 Android Application Security

In Android application security is based on isolation and permission control. This controls what applications are able to do.

When you list all the processes on an Android device you will see a picture similar to Figure 1.4. In the picture you can see, that there are processes that

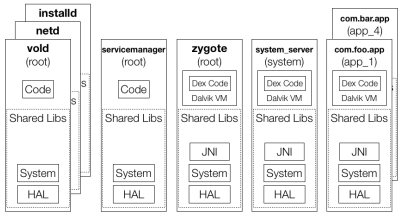

Fig. 1.4 Schematic figure of the processes running in an Android system. Some services run with root privileges which makes them a valuable target for root exploits.

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* [15] 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.

1.5.1 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[2] 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.

1.5.2 Permission Enforcement

Depending on the type of permission it is enforced locally in user space or by the Linux kernel. Local permission enforcement is performed by a so called manager which runs in the application's address space. Please refer to Figure 1.5 for a detailed illustration. An application can try to access a device

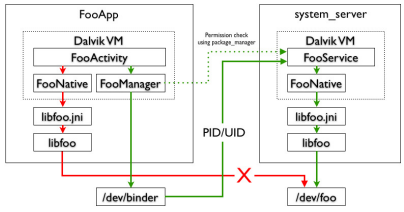

Fig. 1.5 Local permission enforcement on Android is performed by its respective manager.

directly but this will fail in the most cases where the device access rights are set properly, because the application runs with its own UID and GID. Instead the application has to talk to the respective device manager which sets up a communication with the device's service through binder. The device service runs in the system_server process which runs with the *system* user. Through binder the device service receives the PID and UID of the application. With

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

the help of the package manager the device service checks if the respective UID has been granted the requested permission. If the check is successful the device service will allow access to the device.

There are some special permissions which are not enforced by a user space manager but by the Linux kernel. One example is the Internet permission. In order for an application to have access to the Internet it needs to be in the *inet* group. The PARANOID_NETWORK patch to the Linux kernel checks if a process is a member of the inet group and only allows such processes access to the network. Other examples where permissions are enforced by the kernel are the camera, logging and access to the SD card.

1.5.3 Application Provenance

In order to distribute applications through the Google Play Store a developer needs to sign up for a developer account and pay $25. So the question remains whether a user can trust the developer of an application.

Prior to uploading an application to the Play store it must be digitally signed. The certificate can be self signed by the developer. This shows that this process is essentially useless for the user to put trust in the developer. In fact signing is used to ensure the authenticity of the author on updates. Also the certificate is used to establish trust relationships between applications signed with the same key. Those apps are allowed to share permissions and the UID. The user has to trust that the developer keeps his certificate private. If the certificate is lost, the attacker can use it to sign malware and upload it as an update to the original App.

If the private key is lost or expired there is no way to update an existing application. Also Google does not provide a standard way to revoke keys to avoid abuse.

1.5.4 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 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.

1.6 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 [9].

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".

1.6.1 SELinux on Android

The SEAndroid project [28] 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.

1.7 Android Security Problems

According to F-Secure Response Labs 96% of mobile malware that was detected in 2012 targets the Android OS [21]. 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.

1.7.1 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*.

With source code available bugs can easily be found by malicious adversaries. In the case of open source libraries like WebKit, the adversary can learn about vulnerabilities directly from public bug trackers and mailing-lists and repository changelogs. Therefore these bugs have a very high *visibility*, and it is vital that the underlying vulnerabilities are fixed quickly to limit the system's *exposure*.

According to Google Inc, the response to a vulnerability works as follows [11]:

1. *The Android team will notify companies who have signed NDA regarding the problem and begin discussing the solution.*
2. *The owners of code will begin the fix.*
3. *The Android team will fix Android-related security issues.*
4. *When a patch is available, the fix is provided to the NDA companies.*
5. *The Android team will publish the patch in the Android Open Source Project*
6. *OEM/carrier will push an update to customers.*

In practice, updates are very slow to reach the devices, with major updates taking more than 10 months [7]. Many vendors do not patch their devices at all, as the implementation of a patch seems too costly [8]. According to Google Inc.'s own numbers, the most recent version of Android is deployed to only 1.2% of devices [6]. 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 [7].

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 require that any device needs to be certified for correct behaviour before being allowed to use the cellular network. This is done to ensure that the device does not misbehave and therefore does not put the network and its users at risk.

Before an updated firmware can be deployed to the actual smartphones, it needs to be re-certified by the cellular operators. Depending on the operator,

this can take a substantial amount of time. For example re-certification at T-Mobile takes three to six months [23], other carriers opt out of the process and do not ship any updates at all. Notice, that the branding also needs to be ported.

1.7.2 Custom Android Modifications

Skins and brandings are usually not available in source form, because the intrinsics of their implementation are being kept as a trade secret. This has the disadvantage that the code of the Skin is not subject to extensive public review, leaving its security properties solely at the hands of the OEM.

Recently the federal trade commission (FTC) of the USA issued a complaint against the OEM HTC for deliberately weakening the Android security model by not implementing its security measures in its Skin [29]. In the complaint, the FTC mentions a number of problems. For brevity we will concentrate on the following two: "permission re-delegation" and "insecure application installation". Permission re-delegation happens, when applications provide other applications with access to resources without checking whether these applications have the permission to access these resources. An example is the HTC voice recorder, which allows any application to access the microphone. In addition, HTC implemented a way to install applications aside from the official Google play market. However, HTC failed to present the requested permissions to the user, but automatically accepted any permission request, which essentially undermines the Android permission model.

A range of phones based on the Samsung Exynos 4 SoC had the following vulnerability. The Linux kernel's licence mandates that all code running inside the kernel needs to be made available in source code. Device manufacturers however like to keep the workings of their devices a trade secret. In Android this conundrum is solved with split drivers: A small portion of the code is run inside the kernel, and a larger portion runs in userland. The userland part is usually not distributed in source form. The userland part communicated with the kernel part via a custom kernel interface. One driver of the Samsung Exynos 4 SoC simply provided a device file that allowed the userland part direct memory access to the device. Unfortunately the kernel driver did not implement any range checks, and the device file was readable and writable for all applications. That means that essentially any application can readily modify kernel memory and thus root the device [12].

Another problem comes from handling the screen lock. The screen lock can be configured to keep the device locked until the user passed the correct PIN or gesture. Thus, its the purpose of the screen lock to lock the screen both to ensure that only the user himself can tamper with his data and that his private information remains confidential. However, Android allows Apps to present the user with custom user interfaces even when the screen is locked. This is for example for VoIP Apps that allow the user to take a call without having to unlock the screen. However, it is up to the App to ensure that the screen lock is engaged after the action (e.g. the VoIP call) is finished. With this mechanism, the security of the device is up to the App developer, and some fail at implementing it correctly [18]. In a similar case certain Android devices allow an adversary to bypass the screen lock by exploiting an animation that introduces a delay before the screen lock is shown [20].

These cases reveal how third parties like OEMs and carriers that add or modify the code of the AOSP can have a negative impact the security of Android-based devices.

1.7.3 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 [36].

In [32] 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 [35].

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.

1.7.4 Insufficient Market Control

As described in Section 1.5.3, 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 [13] and AndroidPit [17], 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) [24] and Google

Play Developer Program Policies (DPP) [25]. 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* [10]. 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 [5].

Malicious Apps have been found on Google Play repeatedly [19].

1.7.4.1 Application Repackaging

A popular way to get malware onto an Android device is called *application repackaging* [4]. A legitimate application, say Angry Birds, is downloaded to a rooted phone. This application is then copied from the device, decompiled and then repackaged with malware and re-signed. Then it is uploaded under a slightly different name, say Angry Birds Paris, to pretend it is still a legitimate application.

1.8 Lessons Learned

All the knowledge about Android security enables us to provide a number of lessons learned, that we present here to aid developers of future mobile OSes to avoid security pitfalls.

Timely updates are an absolute must for any system that has public interfaces. Source code access, public bugtrackers and mailinglists greatly ease the detection of vulnerabilities. Therefore the importance of timely deployment of security patches is even more pronounced when the system is based on open source software. When designing a mechanism for timely update deployment, one has to take care of all parties involved, including the OEM and the carriers. We think that the key to timely updates lies in clear abstractions. If there was a layer that cannot be customized by OEM and carrier, that layer could be updated independently of the rest of the system. This could be very helpful for the base system as it removes the delay incurred by porting Skin and branding. Moreover, the time incurred for re-certification on the carrier's side should be avoided. We think the way to go is to isolate all software directly interfacing with the baseband from the (general purpose) mobile OS, and to establish a well defined interface between the two. Doing so would enable updates of the mobile OS without re-certification as long as the baseband software is unchanged.

Control platform diversity: The OS designer should enforce that third party modifications to the OS do not introduce security breaches by design. That is define contracts on security critical points in the system that third party implementations have to adhere to. Ensure that these contracts are held. An example is the permission system in Android. 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 it 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 for 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.

1.9 Bring Your Own Device

As described in Section 1.4.3 the device admin API allows an application to enforce certain policies to ease BYOD. However the devices are still in the hands of the employee who are often not trained for security best practices. So if if a device is equipped with device admin, it may set corporate assets at risk. To remedy the problem, a number of solutions have been proposed. All of them have in common that they isolate personal and corporate information from one another.

The actual isolation can be implemented in two ways. Either the isolation is implemented in the middleware, or the entire system is duplicated to run in isolated virtual machines.

Both solutions have their merits and drawbacks. Implementations in the middleware like BizzTrust [33] can leverage existing device drivers, and are therefore easily portable. However, the middleware is already very complex and any implementation enforcing isolation increases the complexity. In fact, in a middleware-based solution the whole software stack must be counted to the trusted computing base (TCB). With this complexity it becomes hard to reason about the system, and any mistake in the implementation is potentially fatal to the isolation capabilities.

With virtualization instead, the whole userland software stack is duplicated. Depending on the implementation of the virtualization layer, the isolation properties can be very well reasoned about. However, virtualization allows only for very coarse grained isolation. It enables setups with a small number of isolated partitions. For example, it enables systems with a private and a corporate partition, which is a excellent solution for BYOD. The corporate partition can be administered by corporate IT, whereas the private partition is under full control of the user. Virtualization does not scale very well because every partition duplicates the whole smartphone OS, which takes a lot of memory. Therefore the number of partitions is limited.

A number of virtualization solutions have been proposed. In the next Section we will briefly introduce the theory and nomenclature of virtualization. We will follow up with a categorization of existing solutions.

1.10 Android Virtualization

For the scope of this paper *virtualization* denotes the act of running an OS in a controlled environment. There are three ways virtualization can be implemented: Containers, type 1 and type 2 hypervisors.

Containers establish isolated environments on top of a shared OS kernel. The kernel ensures that each environment has its own local naming and can only communicate with other environments in a controlled manner. Examples are Linux containers and Linux chroot. These are employed in Cells [30]. All containers share the same kernel. Consequently a kernel compromise is fatal to the security of all containers. Container solutions do not need to virtualize the Android kernel.

Another option is to virtualize the whole Android system including the kernel. Two architectures have been proposed:

Type 1 In type 1 virtualization virtual machines are the main level of abstraction. That is, the hypervisor is a specialized OS that is optimized for virtualization. Consequently, the hypervisor's complexity is orders of magnitude less than that of a general purpose OS like Android. The trusted computing base of a VM comprises the hypervisor and its runtime. Examples for this virtualization architecture are L4Linux [27, 37] and OK:Android [22].

Type 2 In type 2 virtualization the host Android system is enhanced with a kernel module that establishes a VM. The VM runs alongside the applications of the host Android system. The trusted computing base of a VM includes the host kernel, middleware and any application with root permission. An example for this virtualization architecture is VMWare SVP [31].

For a better understanding, all forms of virtualization are illustrated in Figure 1.6.

Type 1 and type 2 virtualization run the entire Android kernel inside a virtual machine. Current smartphones are based on system on a chip solutions that sport one or more ARM CPU cores. Some instructions of the ARM instruction set behave differently when executed in a non-privileged mode than they do in most privileged mode. However, the CPU does not trap when executing these instructions in non-privileged mode (sensitive instructions), which means that virtualizing these instructions with trap and emulate is not possible. Instead, virtualization can be implemented with either of three options:

Emulation In emulation, the complete guest kernel is run in a program that interprets the instructions of the guest at runtime, and runs them using host instructions to create functionally identical effects. Emulation comes at large costs in terms of performance and battery duration.

Binary rewriting In binary rewriting, the guest kernel is patched at runtime such that all sensitive instructions are replaced with instructions that trap. The kernel is then run in a trap- and emulate fashion. Binary rewriting is less costly in terms of performance and battery drain than emulation, but still incurs a large overhead.

Rehosting In rehosting the guest kernel is ported to the host interface. This technique comes with intensive modification of the guest kernel, and requires source code access. Given useful abstractions in the hypervisor, rehosting can offer good performance.

With the ARM Cortex-A15, ARM added hardware virtualization capabilities to the platform that promise much better virtualization performance.

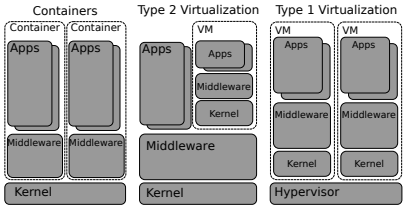

Fig. 1.6 Android virtualization architectures. From left to right: Containers, Type 2 and Type 1 virtualization.

1.11 Conclusion

In this work we investigated the security of the Android mobile OS. We described secure boot and rooting. We proceeded with a detailed description of the Android architecture. We introduce Android application security and the measures against memory corruption attacks as well as an analysis of Android system security. We also analysed all the defence measures of the platform and identified its shortcomings. Given all this insight, we formulate lessons learned that are meant to help the designers of future mobile OSes avoid these pitfalls.

We identified the upcoming trend of BYOD and systematized the different solutions on how industry and research community tries to retrofit support for BYOD in Android with partitioning.

1.12 Acknowledgements

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

References

1. Industry Leaders Announce Open Platform for Mobile Devices . http://www.openhandsetalliance.com/press_110507.html (November 2007)
2. T-Mobile Unveils the T-Mobile G1 — the First Phone Powered by Android. http://www.t-mobile.com/company/PressReleases_Article.aspx?assetName=Prs_Prs_20080923&title=T-Mobile20Unveils20the20T-Mobile20G120E2809320the20First20Phone20Powered20by20Android (September 2008)
3. Google Offers New Model for Consumers to Buy a Mobile Phone. https://sites.google.com/a/pressatgoogle.com/nexusone/press-release (January 2010)
4. New android threat gives phone a root canal. http://www.symantec.com/connect/blogs/new-android-threat-gives-phone-root-canal (March 2011)
5. Adventures in BouncerLand: Failures of Automated Malware Detection within Mobile Application Markets. http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf (July 2012)
6. Android Dashboard. https://developer.android.com/about/dashboards/index.html (December 2012)
7. 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)
8. Arstechnica: What happened to the Android Update Alliance? http://arstechnica.com/gadgets/2012/06/what-happened-to-the-android-update-alliance/ (June 2012)
9. An evaluation of the application verification service in android 4.2. http://www.cs.ncsu.edu/faculty/jiang/appverify/ (December 2012)

10. Google Mobile Blog: Android and Security. http://googlemobile.blogspot.de/2012/02/android-and-security.html (February 2012)
11. Memory Management Security Enhancements. http://source.android.com/tech/security/\#memory-management-security-enhancements (December 2012)
12. Root exploit on Exynos. http://forum.xda-developers.com/showthread.php?t=2048511 (December 2012)
13. Amazon Appstore. http://www.amazon.com/mobile-apps/b/ref=sa_menu_adr_app?ie=UTF8&node=2350149011 (April 2013)
14. Android architecture. http://developer.android.com/images/system-architecture.jpg (April 2013)
15. Android Developer Documentation: Binder. http://developer.android.com/reference/android/os/Binder.html (January 2013)
16. Android Open Source Project: Licenses. http://source.android.com/source/licenses.html (April 2013)
17. AndroidPit. http://www.androidpit.com/ (April 2013)
18. Arstechnica: Critical app flaw bypasses screen lock on up to 100 million Android phones. http://arstechnica.com/security/2013/04/crital-app-flaw-bypasses-screen-lock-on-up-to-100-million-android-phones/ (April 2013)
19. 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)
20. Engadget: Samsung's Android phones affected by another lockscreen bypass, fix is in the works. http://www.engadget.com/2013/03/20/samsungs-android-phones-affected-by-another-lockscreen-bypass/ (March 2013)
21. F-Secure Mobile Threat Report Q4 2012. http://www.f-secure.com/static/doc/labs_global/Research/Mobile20Threat20Report20Q4202012.pdf (March 2013)
22. General Dynamics: OK:Android. http://www.ok-labs.com/products/ok-android (April 2013)
23. Gizmodo: Why Android Updates Are So Slow. http://gizmodo.com/5987508/why-android-updates-are-so-slow (March 2013)
24. Google Play Developer Distribution Agreement. http://www.android.com/us/developer-distribution-agreement.html (April 2013)
25. Google Play Developer Program Policies. http://www.android.com/us/developer-content-policy.html (April 2013)
26. KS2012: Status of Android upstreaming. https://lwn.net/Articles/514901/ (January 2013)
27. L4Android: Android on top of L4. http://www.l4android.org (April 2013)
28. Seandroid wiki. http://selinuxproject.org/page/SEAndroid (April 2013)
29. UNITED STATES OF AMERICA federal trade commission, Complaint against HTC. http://ftc.gov/os/caselist/1223049/130222htccmpt.pdf (February 2013)
30. Andrus, J., Dall, C., Hof, A.V., Laadan, O., Nieh, J.: Cells: a virtual mobile smartphone architecture. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 173–187. ACM (2011)
31. Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., Zoppis, B.: The vmware mobile virtualization platform: is that a hypervisor in your pocket? ACM SIGOPS Operating Systems Review 44(4), 124–135 (2010)
32. 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
33. Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.R., Shastry, B.: Practical and Lightweight Domain Isolation on Android. In: Proceedings of the 1st ACM work-

shop on Security and privacy in smartphones and mobile devices. pp. 51–62. SPSM '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046614.2046624
34. Coverity Inc.: Coverity Scan 2010 Open Source Integrity Report. http://www.coverity.com/html/press/coverity-scan-2010-report-reveals-high-risk-software-flaws-in-android.html (2010)
35. 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
36. 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)
37. Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., Peter, M.: L4Android: A Generic Operating System Framework for Secure Smartphones. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM '11 (2011)
38. 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