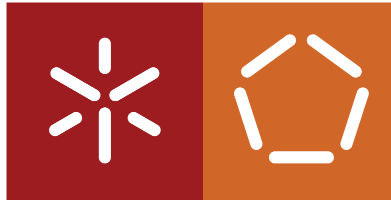

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Rui Miguel de Carvalho Videira Gonçalo

DroidGuardian: An Application Firewall for Android Devices

Master dissertation

Supervised by: Victor Francisco Fonte

Braga, December 19, 2014

ACKNOWLEDGEMENTS

At first, I would like to thank Professor Victor Francisco Fonte for his commitment to this project during its lifetime that had suffered several pauses. The motivation and expertise provided, as well as fundamental advises, brought rewarding achievements.

I would like to mention the open-source community to express my gratitude for all the relevant contents that are shared every day throughout several online platforms that help thousands of developers to contribute to both the creation and improvement of many software products.

A special note of appreciation goes to the Samsung Mobile Security R&D team that by its own initiative had shown particular interest in this project and had supported it with the offer of a Samsung device.

I would like to thank all my friends that despite the distance between towns and countries they have always been present and their support was an essential contribution to this work.

The most important acknowledgment goes to my parents, Óscar Gonçalo and Deolinda Gonçalo and sisters, Helena Gonçalo and Ana Lia Gonçalo that had always been by my side during the best and worst moments. Without their support I would never had finished this work. The acknowledgments extends to the rest of the family, in particular to my aunt Felisbela Carvalho.

At last, I cannot leave this document without having your name written down on it, Isabel. Thank you for being so supportive and shown so much concern regarding my work during the amazing time we were sharing our lives with each other.

ABSTRACT

Mobile devices running Android operating system are increasingly used to surf the web, and, generally speaking, to access a broad spectrum of network-based services. Its successful deployment as a mobile platform, however, also means it is an increasingly relevant target of malicious efforts that try to identify and exploit its vulnerabilities, and to gain access to valuable personal and organizational data. Google and Android manufacturers, such as Samsung, Sony, LG, etc, have been improving and bringing new security mechanisms into the platform. Although, it lacks real protection in several fields, particularly against malware that sends private data to remote servers.

Taking advantage of many years of work invested on the Linux kernel it is possible to develop valuable features aiming to provide helpful contributions in this subject. In fact, Android consumers should be able to be aware of all outgoing Internet connections their device establishes. Furthermore, they should decide whether a connection might or not be established, because it may bring serious risks regarding their private data or for other personal reasons.

This document describes the development of a proof-of-concept of such technology that aims to provide Android users a fine-grained access control over outgoing Internet traffic. This tool is called DroidGuardian and presents a firewall mechanism that uses the Linux Security Modules (LSM) framework to intercept all outgoing Internet connection requests and enables users to be notified and to either accept or reject such requests, in real time. Therefore, DroidGuardian presents a new contribution towards the development of a reliable firewall mechanism for Android devices.

RESUMO

Os dispositivos móveis que correm o sistema operativo Android são bastante usados para navegar na Internet e para usufruir de uma vasta gama de serviços *online*. O seu enorme sucesso enquanto sistema para dispositivos móveis coloca-o num patamar de alto risco, tornando-se num potencial alvo de ações maliciosas que exploram as suas vulnerabilidades a fim de ganhar acesso a dados pessoais. A Google e fabricantes de dispositivos que suportam Android, como a Samsung, a Sony, a LG, etc, têm vindo a desenvolver novos mecanismos de segurança. No entanto continua a faltar uma proteção séria a vários níveis, particularmente contra *malware* que envia dados privados para servidores remotos.

Aproveitando os anos investidos no desenvolvimento do *kernel* de Linux, é possível desenvolver funcionalidades valiosas que contribuem para o melhoramento do sistema Android. De facto, os seus utilizadores deveriam conseguir estar a par das conexões Internet que são geradas pelo seu dispositivo. Além disso, deveriam poder decidir se a conexão se deve ou não estabelecer, porque esta pode trazer sérios riscos por comprometer os dados pessoais, ou por outras razões.

Este documento descreve o desenvolvimento de uma prova de conceito da tecnologia mencionada, que tem como objetivo dar um controlo de acesso refinado sobre o tráfego Internet no dispositivo. Esta ferramenta é chamada DroidGuardian e apresenta um mecanismo de *firewall* que usa a framework de módulos de segurança de Linux para intercetar todas os pedidos de conexão Internet feitas de dentro para fora. Permite, assim, que os utilizadores sejam notificados sobre estes pedidos, dando a possibilidade de os aceitar ou rejeitar em *real time*. Desta forma, o DroidGuardian surge como uma nova contribuição no caminho do desenvolvimento de uma *firewall* confiável para o sistema Android.

CONTENTS

Contents . . . iii
1 INTRODUCTION . . . 3
1.1 Overview . . . 3
1.2 Goals to Achieve . . . 5
1.3 Outline . . . 5
2 ANDROID OVERVIEW . . . 7
2.1 Android Architecture . . . 7
2.1.1 Linux kernel . . . 7
2.1.2 Native Libraries . . . 10
2.1.3 Android Runtime . . . 11
2.1.4 Application Framework . . . 11
2.1.5 Applications . . . 12
2.2 Android Components . . . 12
2.2.1 Activities . . . 12
2.2.2 Services . . . 14
2.2.3 Broadcast Receivers . . . 15
2.2.4 Content Providers . . . 16
2.3 Summary . . . 16
3 ANDROID SECURITY . . . 17
3.1 System and Kernel Level Security . . . 17
3.1.1 Linux Security . . . 17
3.1.2 Application Sandbox . . . 18
3.1.3 Filesystem Isolation . . . 18
3.1.4 Security-Enhanced Android . . . 19
3.2 Android Application Security . . . 19
3.2.1 Manifest Permissions . . . 19
3.2.2 Application Signing . . . 21
3.2.3 Android Security Overview by Google . . . 21
3.3 Summary . . . 22
4 RELATED WORK . . . 23
4.1 Little Snitch . . . 25
4.1.1 Little Snitch rules . . . 25
4.1.2 Little Snitch architecture . . . 27

4.2 TuxGuardian . . . 28
4.2.1 TuxGuardian architecture . . . 28
4.2.2 TuxGuardian Protocol . . . 30
4.3 Summary . . . 30
5 LINUX SECURITY MODULES . . . 31
5.1 Introduction . . . 31
5.2 Design . . . 33
5.3 Implementation . . . 33
5.3.1 Header File . . . 33
5.3.2 Linux Capabilities . . . 36
5.3.3 Framework Initialization . . . 38
5.3.4 Security Functions in the Kernel . . . 41
5.4 Summary . . . 42
6 TECHNICAL CONCEPTS . . . 43
6.1 Loadable Kernel Modules . . . 43
6.1.1 Building . . . 43
6.1.2 Compiling . . . 44
6.1.3 Inserting and Removing . . . 44
6.2 Interprocess Communication using Sockets . . . 45
6.2.1 Stream Sockets . . . 45
6.2.2 Address Formats . . . 48
6.2.3 Address Lookup . . . 49
6.2.4 Kernel Sockets . . . 50
6.3 Android Tools . . . 52
6.3.1 Android Emulator . . . 52
6.3.2 Android Debug Bridge . . . 54
6.4 Android NDK and JNI . . . 55
6.4.1 Java Native Interface concepts . . . 57
6.5 Android development . . . 60
6.5.1 Application Not Responding . . . 61
6.5.2 Worker Threads . . . 62
6.6 Summary . . . 62
7 IMPLEMENTING DROIDGUARDIAN . . . 63
7.1 Conception . . . 63
7.2 DroidGuardian Architecture . . . 64
7.3 Kernel Module . . . 65
7.4 Native Layer . . . 67
7.5 Java Layer . . . 69

7.6 DroidGuardian Protocol . . . 72
7.6.1 Exchanging data between the Kernel module and the Native layer . . . 72
7.6.2 Exchanging data between the Native layer and the Java layer . . . 73
7.7 Discussion . . . 74
7.7.1 Process Name . . . 74
7.7.2 Dialog vs Notification . . . 74
7.7.3 Service and Dialog Communication . . . 75
7.7.4 Starting DroidGuardian on Boot . . . 76
7.8 Setting up the environment . . . 77
7.9 Summary . . . 77
8 USING DROIDGUARDIAN . . . 79
8.1 The Browser show case . . . 79
8.2 Benchmarking DroidGuardian . . . 80
8.3 Summary . . . 82
9 CONCLUSION AND FUTURE WORK . . . 83
9.1 Future Work . . . 85

LIST OF FIGURES

Figure 1	Android architecture	8
Figure 2	Activity lifecycle	13
Figure 3	Service lifecycle	14
Figure 4	Broadcasting an intent to start an activity	15
Figure 5	Application sandboxing	18
Figure 6	Little Snitch Connection Alert window	27
Figure 7	Little Snitch architecture	28
Figure 8	TuxGuardian architecture	29
Figure 9	TuxGuardian notification window	29
Figure 10	The LSM framework architecture (Accessing an inode)	34
Figure 11	The LSM framework files in the Linux kernel filesystem	35
Figure 12	Typical server-client based model of stream sockets	46
Figure 13	Android Virtual Device configuration	53
Figure 14	Application Not Responding dialog window	62
Figure 15	DroidGuardian architecture	64
Figure 16	Kernel Module process flow	66
Figure 17	Process flow in the Native layer	68
Figure 18	Process flow in the Java layer	70
Figure 19	Dialog results diagram	73
Figure 20	DroidGuardian window dialog	79

LIST OF TABLES

Table 1 Android releases and the corresponding Linux kernel versions 9
Table 2 JNI primitive data types mapping 57
Table 3 JNI reference types mapping 58
Table 4 Java types and signatures 61
Table 5 Benchmarking DroidGuardian 81

LIST OF LISTINGS

5.1 Code snippet of the security_operations structure (Linux kernel v3.11) . . . 34
5.2 Code snippet of default security functions (Linux kernel v3.11) 36
5.3 Code snippet of the cap_capable() function (Linux kernel v3.11) 36
5.4 Code snippet of capability functions (Linux kernel v3.11) 37
5.5 Code snippet of the security_fixup_ops() function (Linux kernel v3.11) . . . 37
5.6 Code snippet of the initialization functions (Linux kernel v3.11) 38
5.7 Code snippet of the security_init() function (Linux kernel v3.11) 38
5.8 Code snippet of the do_security_initcalls() function (Linux kernel v3.11) . 39
5.9 Code snippet of the init callbacks (Linux kernel v3.11) 39
5.10 Code snippet of the register_security() function (Linux kernel v3.11) 40
5.11 Code snippet of the security_module_enable() function (Linux kernel v3.11) 40
5.12 Code snippet of some security functions (Linux kernel v3.11) 40
5.13 Code snippet of the socket_create() hook in socket implementation (Linux kernel v3.11) 41

6.1 Defining the Loadable Kernel Module's entry point 43
6.2 Defining the Loadable Kernel Module's exit point 44
6.3 Example of a Makefile to compile Loadable Kernel Modules 44
6.4 Linux command to insert Loadable Kernel Modules 44
6.5 Linux command to remove Loadable Kernel Modules 45
6.6 Declaration of the socket() function 45
6.7 Declaration of the bind() function 45
6.8 Declaration of the listen() function 46
6.9 Declaration of the connect() function 47
6.10 Declaration of the accept() function 47
6.11 Declaration of the sendmsg() and recvmsg() function 47
6.12 Declaration of the close() function 47
6.13 Declaration of the sockaddr_in structure 48
6.14 Declaration of the in_addr structure 48
6.15 Declaration of the sockaddr_in structure 48
6.16 Declaration of the in6_addr structure 48
6.17 Declaration of the sockaddr_un structure 49
6.18 Declaration of the inet_ntop() function 49

6.19 Declaration of the getnameinfo() function 50
6.20 Declaration of the sock_create() function 50
6.21 Declaration of the socket structure 50
6.22 Declaration of the proto_ops structure 51
6.23 Declaration of the sock_sendmsg() and texttttsock_recvmsg functions 51
6.24 Declaration of the msghdr structure 51
6.25 Declaration of the iovec structure 52
6.26 Command to start the Android emulator 53
6.27 Command to provide a kernel to the Android emulator 53
6.28 Command to provide kernel prints of the Android emulator 53
6.29 Command to show Android devices running on the computer 54
6.30 Example of the output from the adb devices command 54
6.31 Command to install Android apps using the adb utility 54
6.32 Command to pull files using the adb utility 55
6.33 Command to push files using the adb utility 55
6.34 Command to get logcat's prints using the adb utility 55
6.35 Command to start a remote shell using the adb utility 55
6.36 Declare a JNI library in Java 55
6.37 Declare native methods 56
6.38 The minimum set of instructions in the Android.mk file 56
6.39 Example of use of the javah tool 57
6.40 Creating a new Java string from a given C string 58
6.41 Creating a new C string from a given Java string 58
6.42 Declaring Java fields 59
6.43 Accessing a Java instance field 59
6.44 Acessing a Java static field 59
6.45 Getting and setting a Java instance field value 59
6.46 Declaring Java methods 60
6.47 Acessing Java methods 60
6.48 Example of use of the javap tool 60

7.1 Code snippet of the DroidGuardian's security_operations structure 65
7.2 Declaration of the droidg_socket_connect() function 65
7.3 Changing socket file permissions 69
7.4 Declaration of the dg_query structure 72
7.5 Declaration of macros to the dialog results 73
7.6 BroadcastReceiver responsible to start the service after the booting process 76
7.7 Declaring the receiver component on the Manifest file 76

1

INTRODUCTION

This document is a master dissertation that takes part of the second year of the Master Degree in Computer Engineering that is held at University of Minho in Braga, Portugal. The work presented in this master dissertation is included in the field of Android security.

1.1 OVERVIEW

Android is the most popular Operating System (OS) for mobile devices. Since its first release in 2008 that the worldwide market share has been constantly growing reaching almost 80% in 2013 [1]. This popularity has been growing due to the powerful features it brings to users. Android consumers are able to accomplish an infinite amount of tasks that facilitate their daily routines, through an useful set of resources, such as network connection, Global Positioning System (GPS), telephony, camera, etc, and a robust Application Programming Interface (API) that allow developers to build their own programs to run in Android devices usually designated as *applications* or *apps*, in short-form.

A valuable asset of Android devices is the network component that allow users to get connected to the Internet. It uses the same networking architecture of personal computers that run Linux systems. In fact, the Android OS core is based on the Linux kernel which presents a lot of similarities with Linux based operating systems to desktop computers. The access to the Internet is undoubtedly one of the most important features nowadays, and the Android's interface makes it really simple to exchange data over it.

However, high popularity also means a valuable target by malicious actions. Android has suffered from malicious attacks since its beginning what led to the development of security measures that have been introduced in almost every new OS version. One of the measures that was brought since the first release of Android was the *Manifest* permissions model. Whenever an user choses to install a certain application, it is prompted a list with all device's resources and operations that the application wants to grant access to or execute, which the user must accept in order to proceed the installation process. If he feels uncomfortable with some of the contents of the list, for instance, granting a simple local game access to the Internet, he may choose not to accept the app's permissions and the installation process is canceled. Once the Manifest permissions's list is accepted, the application will always have

access to the described resources , as well as execute the listed operations, and the user won't be asked again.

The Internet permission allow applications to open network sockets. If this permission is granted, the application is free to establish Internet connections to any remote server. Android provides no filter to control the incoming and outgoing traffic. Usually, applications use Internet to provide handy features. But, there are many cases where Internet is used to cause harm, very often without the users's knowledge. Malware takes advantage of the granted Internet permission to send out personal data it has access to. It is up to the user to inspect the Manifest permissions and decide whether the application's purpose fits the permissions it is requiring. Most of the times, applications actually need to use network resources for beneficial goals. For instance, even a simple game may need to connect to the Internet to download advertise data. Normally, users don't deny some application's installation because of the Internet permission, even though it may seem odd that it requires it. Users don't know if it will be used for legitimate or illegitimate actions. Unfortunately, in case of doubt they choose to take the risk.

Introducing a real case, it was going into discussion the possibility that the worldwide popular game *Angry Birds* had been sending users's personal data to the National Security Agency (NSA). According to the rumor, the NSA had installed backdoors on several applications, as *Angry Birds*, and had been collected huge amounts of private data [2]. This case presents a strong evidence that the Android platform lacks security measures.

In order to bring some control to users regarding network connections, it was purposed the development of a mechanism able to detect all outgoing traffic. Such technology would notice every connection that applications attempt to establish to the outside world, preventing the access to remote servers without the user consent. Along with the connection requests detection, the mechanism should also be able to deny such requests in real time. If the user does not feel comfortable that a certain application gets connected to a certain remote server, he should be able to deny the connection.

This document introduces the development of the aforementioned technology. It aims to provide Android users the ability to be noticed whenever a new outgoing Internet connection request is launched by an installed application along with its acceptance or rejection in real time. This mechanism goes by the name of *DroidGuardian* and consists in a proof-of-concept that the LSM framework may be used to protect Android consumers regarding network connections.

In practical terms, DroidGuardian enforces a fine-grained control over outgoing Internet connections acting as a firewall. It takes advantage of the LSM framework, part of the Linux kernel, to intercept socket connection calls. An alert message is launched whenever a new outgoing Internet connection request arises. Users are able to either accept or reject such request, being the decision saved as rule that will be enforced in future requests.

This document describes the entire development process, starting by fundamental concepts regarding the subject and followed by a technical description of DroidGuardian.

1.2 GOALS TO ACHIEVE

The purposed mechanism should fulfill the following requirements:

- Intercept all outgoing Internet connection requests launched by any process running in the system;
- Extract the following data from the connection requests:
  - Internet Protocol (IP) address of the remote server;
  - Port of the remote server;
  - Name and identifier of the process that launched the request;
- Provide a Graphical User Interface (GUI) to display the information above to the end user;
- Enable users to either accept or reject the request in real time;
- Implement a rule-based model to filter connection requests;

In order to assess the performance of DroidGuardian, it should be measured its overhead.

1.3 OUTLINE

This section gives a brief description of all chapters presented on this dissertation.

In Chapter 2, it is introduced a general overview of the Android platform. Since this document describes a project that involves the development of an Android application, it is very important to provide several basic notions regarding the Android platform architecture, as well as its fundamental components. Throughout the dissertation there are various references to the Android platform layers, which this chapter presents with an appropriate level of detail.

In Chapter 3, it is given a brief approach over the main Android security mechanisms. In order to understand how DroidGuardian fits the needs in the Android security model, it is necessary to understand how this model is designed.

In Chapter 4, it is presented the most relevant related work. There is a brief description regarding firewall mechanisms and how they are deployed in Linux systems, as well as the introduction to several research projects that were developed to Android. Also, this chapter presents two tools that were not designed for Android, but served as inspiration to build DroidGuardian.

In Chapter 5, it is introduced a detailed description regarding the mechanism used to interact with Internet sockets at kernel level. The chapter gives an introduction about the subject, followed by technical aspects that comprise the LSM framework.

In Chapter 6, it is presented general technical concepts that played an important role in the scope of this project. In order to achieve DroidGuardian's goals, it was necessary to understand the Linux

networking architecture, as well as the loadable kernel modules's mode of operating. The Android framework includes two main development kits which are introduced in this chapter.

In Chapter 7, it is given all details regarding the development of DroidGuardian. The chapter provides an internal overview of the mechanism with technical descriptions of each component. Along with the tool's description, it is presented a discussion section to explain the most relevant decisions and aspects in the development phase.

In Chapter 8, it is presented a show case of DroidGuardian. This chapter also presents an simple benchmarking in order to evaluate the overhead caused by DroidGuardian.

In Chapter 9, it is described the main conclusions regarding the entire development process from the designing phase to the deploying phase of DroidGuardian. It is discussed the goals achievement, and, in the end, it is presented a set of future enhancements to the tool.

2

ANDROID OVERVIEW

Since this project involves the development of an Android application, even though it runs a bit off the scope of common applications, it was mandatory to get a deep understanding of both the Android architecture and components. This chapter introduces these topics.

2.1 ANDROID ARCHITECTURE

Android platform architecture consists of four main layers, presented in Figure 1. At the bottom, we found the Linux Kernel, responsible for bridging hardware and software, providing drivers and essential components to the operating system's life. Above the kernel is placed a set of libraries and the Dalvik Virtual Machine (Dalvik VM), which is a lighter version of the Java Virtual Machine (JVM) specially designed and optimized for Android. The Application Framework was built on top of libraries and the virtual machine to provide higher-level services to applications in the form of Java classes. The topmost layer is composed of Android applications which with users interact. The following sections present a deeper insight into each layer.

2.1.1 *Linux kernel*

Android adopted a famous kernel with proven value concerning efficiency and security. Due to the wide set of constraints that mobile devices present comparing to desktop devices, the Linux kernel suffered some changes. It was modified in order to achieve excellent results in embedded environments. Therefore, the Android kernel is not a regular distribution of the Linux kernel, but a fork of the mainline kernel source code which allows the Android development team to both implement their necessary changes and follow the Linux kernel updates. This is a big advantage, because the Linux kernel is developed and maintained by a large community that releases, frequently, new patches and versions with enhancements, what lead Android kernel to adopt these enhancements. In fact, every new release of Android usually benefits from a new Linux kernel version. Table 1 shows Android releases and the corresponding Linux kernel version.

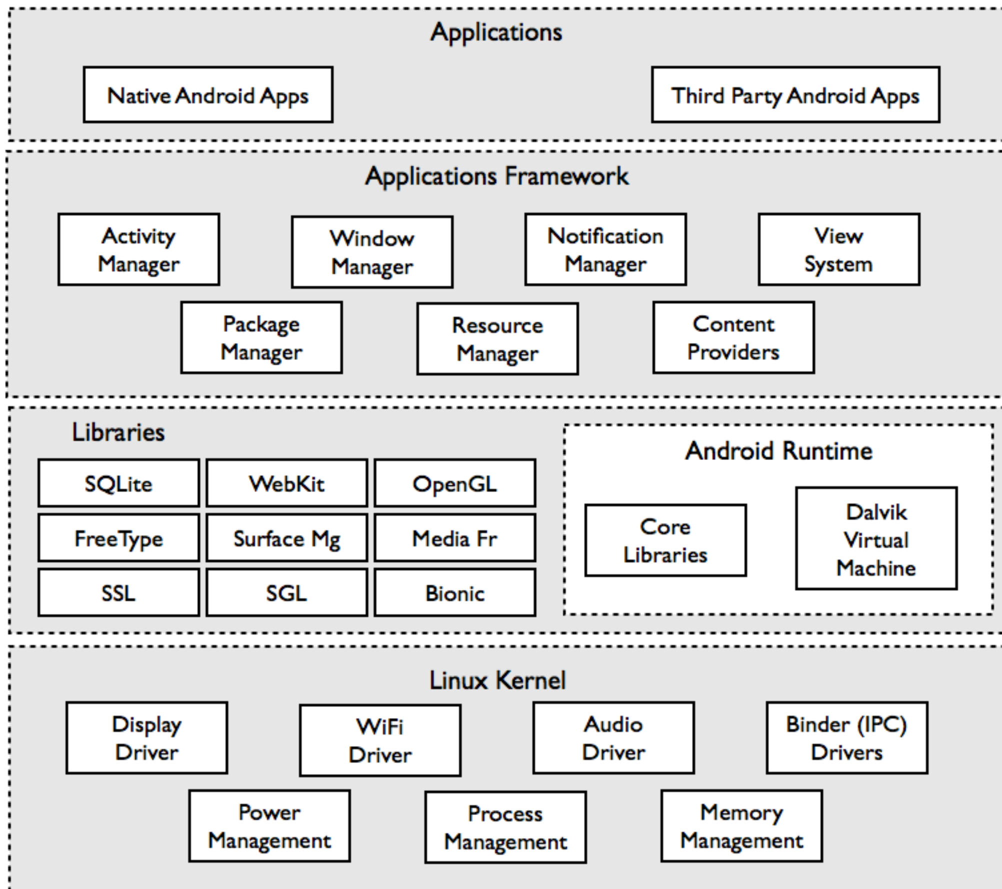

Figure 1: Android architecture

Google created the Android Open Source Project (AOPS)[1] to share the Android source code, that goes under the Apache Software License, Version 2.0, and related documentation. Since Android is a product of the Open Set Alliance, which includes a considerable amount of mobile manufacturers that present different specifications of hardware, several branches of the Android kernel source code are kept on the git repository[2].

The way a mobile device operates is quite different from a laptop or desktop. As mentioned earlier, the Linux kernel suffered several modifications in order to fit a mobile device needs. It became an *Androidized* kernel [3]. The following presents some of the most significant changes and new components brought to the kernel:

- **Wakelocks** was one of the updated components. In Linux, the power management behaves according to the position of the lid in a laptop computer. If the lid is down, the power management will usually put the computer into ”suspend” or ”sleep” mode, the state of the processes is

1 http://source.android.com
2 https://android.googlesource.com

Android version	**Linux kernel version**
Android Cupcake 1.5	Linux kernel 2.6.27
Android Donut 1.6	Linux kernel 2.6.29
Android Éclair 2.0/2.1	Linux kernel 2.6.29
Android Froyo 2.2	Linux kernel 2.6.32
Android Gingerbread 2.3.x	Linux kernel 2.6.35
Android Honeycomb 3.x	Linux kernel 2.6.36
Android Ice Cream Sandwich 4.0.x	Linux kernel 3.0.1
Android Jelly Bean 4.1.x	Linux kernel 3.0.31
Android Jelly Bean 4.2.x	Linux kernel 3.4.0

Table 1: Android releases and the corresponding Linux kernel versions

stored in RAM and the remain hardware turns off. This allows the laptop to save battery power. A mobile device should be in ”sleep” mode as often as it is possible, but must not ”sleep” when important processes are executing. Wakelocks are used to keep the system awake. Drivers developers need to grab and release wakelocks when important processing is being done or when an application is waiting for the user's input.

- **Low-Memory Killer** executes before the default kernel Out-of-Memory (OOM) killer. When the system lacks of free memory, processes can no longer allocate more memory and the kernel kills a task to get available space. This task is chosen based on priorities. Android's low-memory killer attributes OOM levels to processes depending on the components they are running and applies a threshold for each type of process. Android avoids the OOM state by reaching this threshold and killing tasks.

- **Binder** is an Interprocess Communication (IPC) mechanism adopted by Android that was based on OpenBinder. By IPC we understand a framework that has the purpose of exchanging signals and data across multiple processes. It is used for message passing, synchronization, shared memory and remote procedure calls. Binder develops an important role among Android application components, as Content Providers, Services, etc [4].

- **Anonymous Shared Memory (ashmem)** is another IPC mechanism that is implemented as the POSIX SHM functionality, part of the System V IPC in Linux. However, the Android development team argued that this mechanism leads to resource leakage within the kernel [3]. Therefore, ashmem is based on POSIX SHM, but takes some enhancements. For instance, it uses reference counting to destroy memory regions when all processes have exited and reduces mapped regions when the system needs memory.

- **Alarm** is another example of a driver that required some improvements comparing to the one of the default kernel. Android introduces the alarm timer, an hybrid solution that triggers a High-Resolution Timer (HRT) to fire when an event is supposed to run, while the system is running and, when the system suspends, the alarm timer looks at the list of events and sets the Real-Time Clock (RTC) to fire an alarm when the earliest event is to run [5].

- **Logger** is a new mechanism of logging developed specially to Android. In Linux, typically, he find two logging systems: the kernel's own log, accessed through the `dmesg` command, and the system's log, stored at `/var/log/`. In Android there is a logger driver on the kernel that maintains circular buffers in RAM where it logs every incoming event [6]. This contrasts with Linux logging systems, because they use task-switches and file-writers to log each event, turning the process quite complex and heavy.

From a security point-of-view, the Android kernel inherited the user-based permission model from Linux that will be explained in Chapter 3. A new security feature was implemented on kernel, available as a build option called `ANDROID_PARANOID_NETWORK` that restricts the access to some networking features, depending on the Group ID (GID) of the calling process [7].

2.1.2 *Native Libraries*

Android has a considerable amount of dynamically loaded libraries that supports both Android system to execute internal tasks and developers to use native code in their applications. Native libraries are written in C/C++, being available through the Java Native Interface (JNI). These libraries are placed at `/system/lib` in the Android filesystem. The following list presents the most relevant libraries:

- **Media Libraries** Enables playback and recording of audio and video formats. Based on OpenCore from PacketVideo;

- **SQLite** Provides relational databases that can be used by applications and systems;

- **SSL** Provides support for typical cryptographic functions;

- **Bionic** System C library;

- **WebKit** Browser-rendering engine used by Android browsers;

- **Surface Manager** Provides support for the display system;
- **SGL** Graphics engine used by Android for 2D.

2.1.3 *Android Runtime*

Android development team decided to use Java as the main language to build Android applications, because it is one of the most worldwide used programming languages. In Java, there is a Java compiler that translate Java code into architecture-independent byte-code, which is executed at runtime by a byte-code interpreter known as ”virtual machine”. As Java programmers, we are used to the JVM. However, considering the restrictions of mobile devices's power of execution, the JVM is quite heavy. Therefore, Google decided to build a new ”virtual machine” to deal with Java code and it is called *Dalvik*. Apparently, the name was stolen from a village in Iceland [8]. The Dalvik VM is designed to achieve an appropriate performance in embedded environments that uses slow CPUs, less RAM and are battery powered.

2.1.4 *Application Framework*

Similar to native libraries, the Application Framework offers a set of libraries to support developers. In this layer, libraries are written in Java and are available through Java APIs. The following list describes the most used libraries:

- **Activity Manager** Manages the activity lifecycle of applications and various application components. When an application requests to start an activity, Activity Manager provides this service;
- **Resource Manager** Provides access to resources such as strings, graphics, and layout files;
- **Location Manager** Provides support for location updates (e.g., GPS);
- **Notification Manager** Applications interested in getting notified about certain events are provided this service through Notification Manager. For instance, if an application is interested in knowing when a new e-mail has been received, it will use the Notification Manager service;
- **Package Manager** The Package Manager service, along with *installd* (package management daemon), is responsible for installing applications on the system and maintaining information about installed applications and their components;
- **Content Providers** Enables applications to access data from other applications or share its own data with them;
- **Views** Provides a rich set of views that an application can use to display information.

2.1.5 *Applications*

The top layer is composed of the main pieces of the entire system: applications. Android usually comes with several applications, as browser, mail, contacts, etc. Through Google Play, and other third party markets, users may download and install applications that are no different from those previously installed on the device. Android applications present the following filesystem structure:

- `src` includes the java packages and files;
- `gen` holds auto generated code for resources;
- `Android` *`X.X.X`* contains the android jar file for the targeted version of Android, for instance, `Android 2.3.3`);
- `assets` comprises those files that the developer bundles to the application;
- `bin` stores files for compiling and running the application, as the *apk* file and *classes.dex* files;
- `res` contains all application resources: layouts, values (like strings) and drawables;
- `AndroidManifest.xml` defines the application components;
- `proguard-project.txt` is the proguard configuration file.

Later in this document several references to Android folders will be made.

2.2 ANDROID COMPONENTS

The following sections present the Android components by which applications consists of. Each component was designed to develop a special role in the application's life and some rules need to be carried out in order to get the desired behavior, as well as efficiency.

2.2.1 *Activities*

Android provides the application's visual interface through the *Activity* component. Once created, it exhibits elements that users can interact with, like buttons, text boxes, spinners, etc. When developers are implementing Android activities, concepts regarding visual design must be taken into account so that users may have a pleasant experience. Regular applications have several activities, because the visual interface changes according to the user's desire, while he keeps tapping and clicking along the application's execution. Android provides mechanisms to save activities state when they are paused or stopped and keeps them in a stack so that they can be restarted later. This process is presented in Figure 2 that illustrates the activity lifecycle[3].

3 http://developer.android.com/guide/components/activities.html

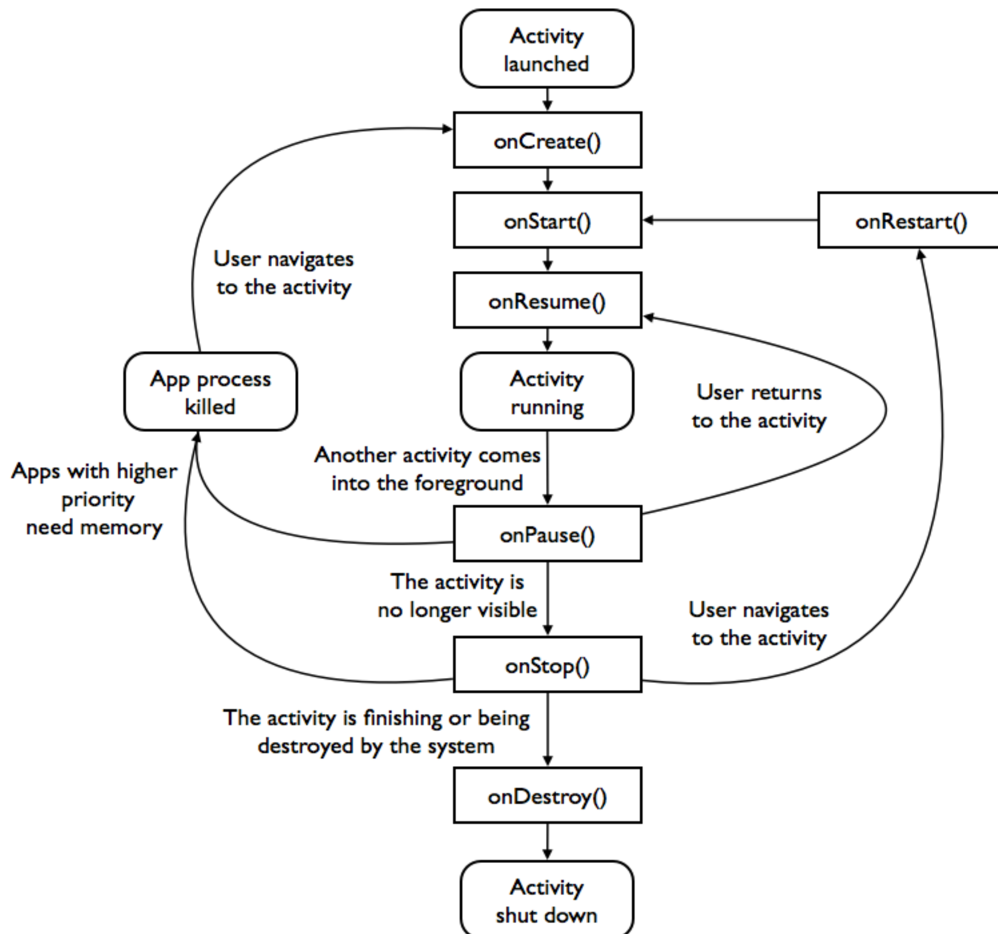

Figure 2: Activity lifecycle

Activities begin the execution calling `onCreate()` that, usually, defines the layout for the activity's user interface. The activity becomes visible when `onStart()` runs. Once the activity is visible, `onResume()` takes place and the activity will only stop of being visible when another activity comes to the foreground. When this happens, `onPause()` is called. At this point, one of three actions may take place. If the system needs memory to execute activities with higher priority, the activity is stopped. If it is requested to run again, it can assume the previous state, coming to the foreground and executing `onResume()`. The activity may also be stopped through `onStop()`. Once stopped it cannot go back to the previous state, but might be restarted through `onRestart()`. At last, the activity is destroyed by calling `onDestroy()`. The activity is shut down and its lifecycle ends.

2.2.2 *Services*

When developers intend to execute some operation that has no visible elements, they can use *Services*. This component is designed to perform long running operations in the background. For this reason, a service is able to run even if the component that called it, or even the application, stops its execution. Services usually take care of operations like Internet downloads, music playing, etc.

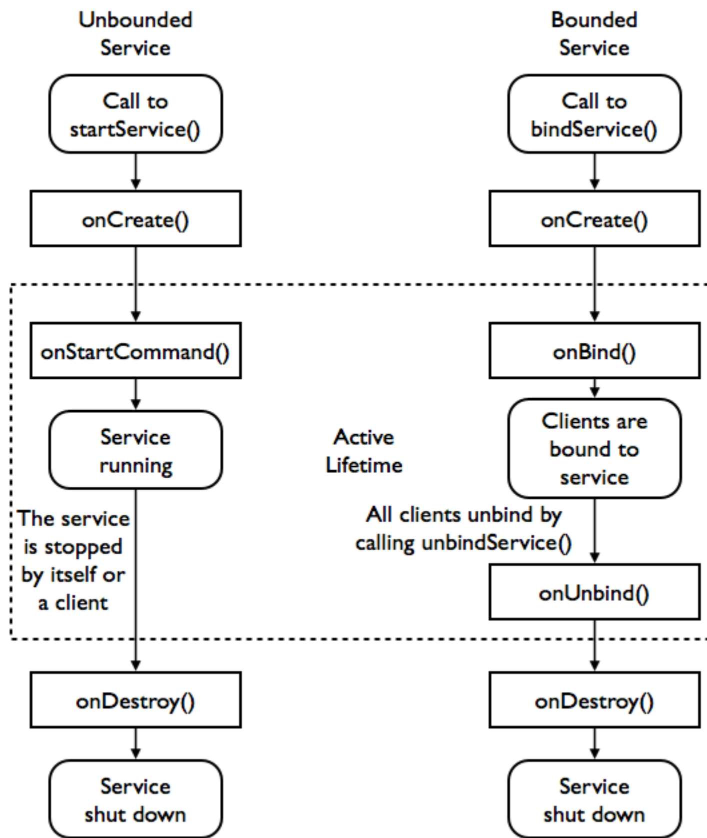

Figure 3: Service lifecycle

Services may be called in two distinct ways. An application component, such as an activity, may start a service calling `startService()`. It may run in the background indefinitely, even if the component that started it is destroyed. After completes its operations, the service should stop itself. In the other way, a service can be bound to an application component, when this binds to it by calling `bindService()`. In this case, the service executes using a service-client interface providing interaction with components, as sending requests, getting results, etc. A bound service runs while it is bound to some application component, being destroyed after that. Note that the same service may assume both forms, unbound and bound.

In Figure 3 it is shown both service lifecycle's approaches[4]. On the left side we can see that an unbounded service starts its work by calling `onStartCommand()`. After performing it may be stopped by a client or by itself, calling `onDestroy()`. In a bounded service, `onBind()` starts its execution and when all clients unbind the service, it calls `onUnbind()` and `onDestroy()`.

Services play a major role in the scope of this project, because it's through a service that Droid-Guardian is able to perform indefinitely in the background, being started when the device boots, as we will explain further in Chapter 7.

2.2.3 *Broadcast Receivers*

Broadcast receivers are built to handle events created by applications or by the system. Receivers are designed to perform a certain action when notified that some event occurred. For instance, a receiver can be set to start an activity when the device boots. The developer registers the action `BOOT_COMPLETED` wrapped in a package called *intent*. When the system performs this action, sends the package to the receiver. The receiver checks the action inside. If it is the desired action, the receiver sends another package to the system requesting an activity to start. Receivers must always be associated with intents. An *Intent* is a messaging object that connects all components in an Android applications by allowing them to be invoked and sharing some data[5]. Figure 4 exhibits the way application components use intents to communicate.

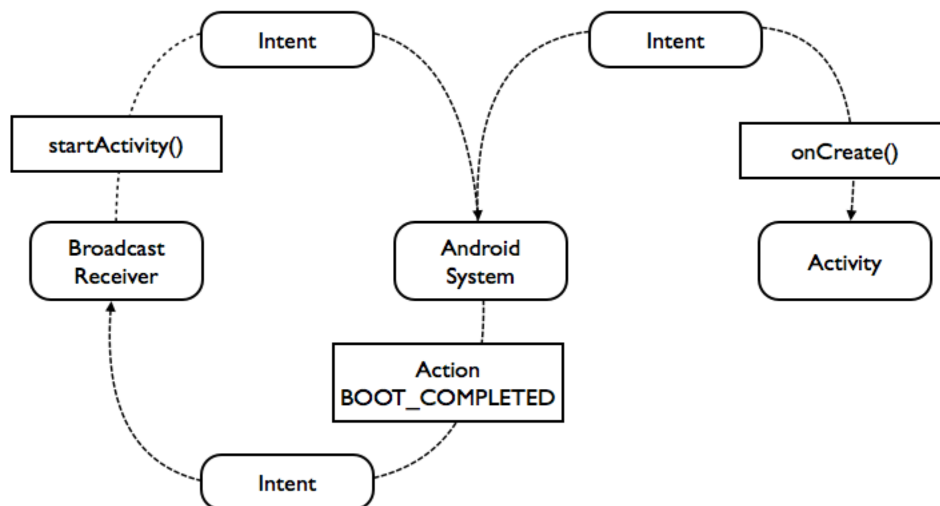

Figure 4: Broadcasting an intent to start an activity

4 http://developer.android.com/guide/components/services.html
5 http://developer.android.com/guide/components/intents-filters.html

2.2.4 *Content Providers*

It is common that Android applications need to access and share some resources in order to provide the user useful features. These resources can be user's personal data, such as videos, audio, images, contacts, etc. Android supplies a consistent standard interface to data that also handles IPC and secure data access. *Content providers* offer this mechanism as an application component, by which it allows the application to access a data repository. Providers are primarily designed to be used by other applications, even though they can be called only to manage its application's internal data. Providers present data to external applications using a relational database like interface, providing CRUD (create, retrieve, update and delete) functions and a Uniform Resource Identifier (URI) system[6].

2.3 SUMMARY

This chapter introduced fundamental concepts regarding the Android framework. It is presented its architecture and detailed each layer. Also, the main Android components that allow developers to build reliable applications are introduced. Some of these components play an important role in Droid-Guardian's life and for that reason they were carefully studied. The next chapter will bring an overview regarding the Android security model.

6 http://developer.android.com/guide/topics/providers/content-providers.html

3

ANDROID SECURITY

Android was designed to protect applications considering both security-oriented developers and those less familiar with safety concerns. By default, Android enforces good levels of protection, inheriting the Linux security model, but also applying its own mechanisms. It is provided with a multi-layered security that supplies the flexibility required for an open platform, while providing protection for all users of the platform[1]. This chapter introduces a general overview into Android security features.

3.1 SYSTEM AND KERNEL LEVEL SECURITY

The Android platform comprises three main blocks: device hardware, operating system and application runtime. Each block presents secure mechanisms that are briefly described in the following sections.

3.1.1 *Linux Security*

Android has inherited security mechanisms from the Linux kernel, namely, a user-based permissions model, process isolation and extensible mechanism for secure IPC. The user-based permissions model was originally developed for Unix environments, thus Linux takes advantage of it. Every user registered in the system has an unique identifier number known as User ID (UID). Along with users, there are groups that are identified by its unique GID. One group might have one or more users, and one user might belong to one or more groups. Note that all users belong to at least one group, which is the group that contains all users. Every resource in the system, or in simple terms, every file in the system has an owner, that is identified by its UID. This owner has the responsibility over the file and is able to alter its permissions. Files have also a group associated which is identified by its GID. Each file on a Linux system has three sets of permissions: User, Group and Other, comprising the triplet known as UGO. The User and the Group are those mentioned before. The Other is considered to comprise every user registered on the system. Each file might be accessed by three types: read, write and execute. So, each set of permissions can include read (r), which allows an entity to read the file; write (w) which

1 http://source.android.com/devices/tech/security

allows an entity to write the file; and execute (x) which allows an entity to execute the file. According to its permissions, a file may be read and/or wrote and/or executed by its owner, that has an unique UID, and/or by every member of the file's group, and/or by all other users that have an account on the system [9].

3.1.2 *Application Sandbox*

Using the user-based permissions model, the system's resources have a robust access control. Android took this feature and built an application sandbox where each application can only access its own files and components (unless the developer grants other permissions as we will see further in this chapter). When an application is installed on the system, an new unique UID is assigned to it and the application runs under this UID. In addiction, all data stored by that application is assigned the same UID. The Linux permissions are set on this application to allow read, write and execute access by its owner and no permissions otherwise. This mechanism is illustrated in Figure 5 [10].

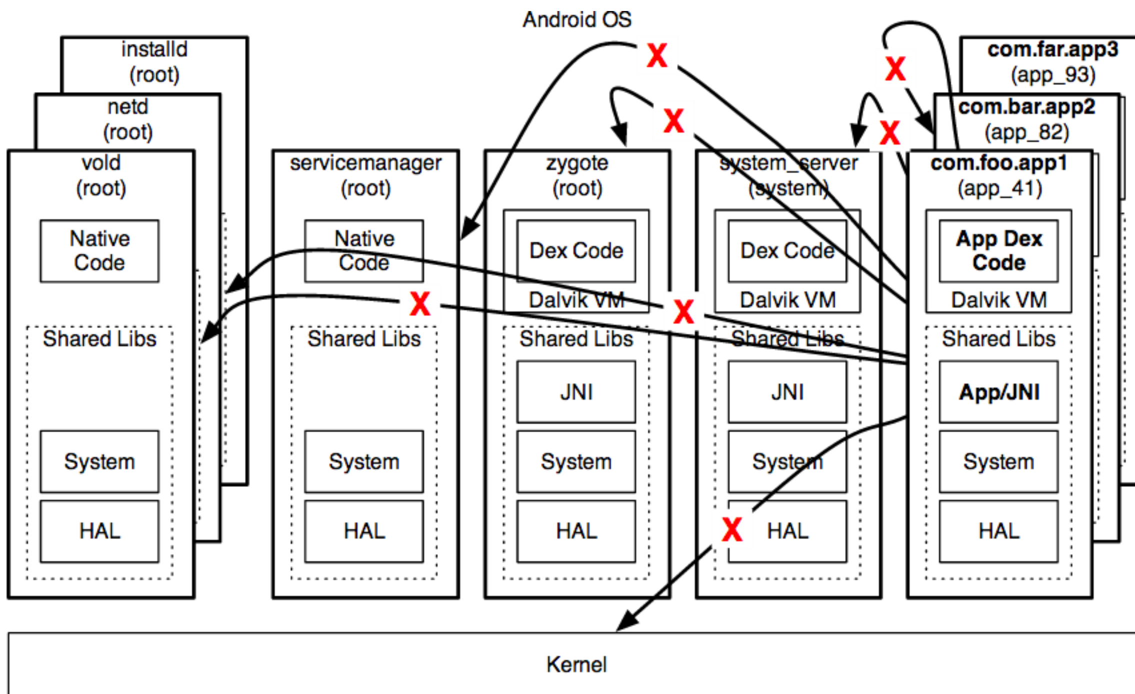

Figure 5: Application sandboxing

3.1.3 *Filesystem Isolation*

The user-based permissions model is also used to provide filesystem isolation, which fits in the application sandboxing model. Android creates a specific directory to each installed application under

the path /data/data/. Each directory is configured such that the associated application's UID is the owner and only its permissions are set. Within this directory is files/ directory that stores all files created by the application. These files are granted the same permissions and run under the owner's UID, providing isolation access from other applications. This access control is enforced to all applications. However, if an user access the Linux kernel using the root UID, he will break down the sandboxing mechanism and be able to access any data stored in any application.

The Linux permissions access control works on every Android filesystem except on the SD card (/sdcard directory). Therefore, any file written to external storage is accessible by any application.

3.1.4 *Security-Enhanced Android*

As mentioned above, the user-based permissions model inherited from Linux grants protection in the Android core. However this model follows a Discretionary Access Control (DAC) policy that raises the risk of harm, as we will see later in Chapter 5. To overcome the related threats, Android began to use a component that has been in the Linux kernel in the last years, Security-Enhanced Linux (SELinux). Among other features, this mechanism applies a Mandatory Access Control (MAC) policy [11] that reduces the effect of malware and protect users from potential flaws in code[2].

3.2 ANDROID APPLICATION SECURITY

Android applications extend the core Android operating system. The previous security features were not able to ensure the protection level desired to a worldwide used mobile platform as Android, therefore a set of features were developed to grant applications safety in a satisfactory degree. They are briefly described as follows.

3.2.1 *Manifest Permissions*

Besides the user-based permissions model adopted from the Linux kernel, Android brought a permissions model known as *Manifest permissions*. As mentioned earlier, each application is only allowed to access its own data, by default. However, Android offers a lot of resources and libraries so that developers can build powerful and useful applications. But, the gain of power brings security vulnerabilities. For instance, Android provides network resources that allow applications to establish Internet communications. But, malicious applications could take advantage of this feature and use it to spread user's personal data.

Google decided to implement the Manifest permissions model that forces developers to specify which resources and operations their applications have access to and may execute. Each resource requires a permission that must be declared on the Manifest file. At installation time, permissions

2 http://source. android.com/devices/techsecurity/se-linux.html

are set to the application and it will only have access to the declared resources. Using the previous example, if the developer wants to use network resources, he declares the `Internet` permission through the following statement:

```
<uses-permission android:name="android.permission.Internet"/>
```

on the Manifest file. Before the installation, the user gets the list of all Manifest permissions. This feature brings two main advantages. First, it alerts the user to all possible dangerous actions the application may take. For instance, if the user intends to install a simple game and the Manifest file exhibits SMS and phone call permissions, which means that the application can send SMS and make phone calls, something doesn't seem to be right. The user makes his judgement and decides to either install or not install the application. The second advantage ensures the protection of the application against malware. In the case of one application gets compromised, the attacker will only be able to access the resources that the application was allowed to. For instance, if an application that takes photos has only permission to use the camera and gets compromised, the attacker will only be able to access the camera and none of the remaining resources that need Manifest permission.

Android comprises a large set of Manifest permissions[3] and regular applications take advantage of a considerable amount of them. Since there is a considerable set of permissions that causes no harm to the device, users don't need explicitly to accept them in order to install applications. Therefore, Google established four categories where Manifest permissions fall into, also characterized as *protection levels*, described as follows:

- **Normal**. Permissions to access inoffensive resource. For that reason they are granted by default. As example, the permission to change the device's background.

- **Dangerous** Permissions to access resources that might cause harm to users. In this case, users must accept them before the installation. As example, the permission to access private data, or establish Internet connections.

- **Signature** Permissions that are required by applications signed with the same digital certificate. If the application is signed by the same certificate as the declaring app, the permission will be granted; otherwise the app being installed will not be granted the permission. The user is never questioned about these permissions in order to start an installation. The main purpose of this protection level is for two applications owned by the same developer to be able to share data seamlessly without bothering the user.

- **SignatureOrSystem** These permissions follow the same rule as *Signature* permissions, but adding a new rule that checks the Android system image. This type of permission is used by device manufacturers to allow applications created by different vendors to work together within that Android builds.

3 http://developer.android.com/ reference/android/Manifest.permission.htm

An important rule that follows the Manifest permission is the Principle of Least Privilege [12], which states that each application should keep permissions at its minimum, using the weak permission instead of a strong one that would allow it to carry out certain functions that will not be called. For instance, if the application only needs to read contacts, the permission required should be READ_CONTACTS and not full access to contacts that also allow to write contacts.

3.2.2 *Application Signing*

Google requires all Android applications to be signed through a digital certificate, being the private key held by the application's developer. This process ensures the authentication of a developer when he is trying to deploy his application into the market and establishes trust relationships between applications. Signing an application does not require a Certificate Authority (CA). In fact, most of Android applications are self-signed by developers. Google released tools that allow developers to sign their applications and provides useful documentation to facilitate the process[4].

The Application signing process concede an useful feature to developers that build more than one application. As mentioned earlier, each application is assigned an unique UID and is not allowed, by default, to share data and resources with other applications. However, if an user installs more than one application signed by the same developer, which means the same digital certificate, and these applications declare the shareUserId attribute in the Manifest file, Android assigns these applications the same UID. Therefore, they are seen by the Linux kernel as the same application and are able to share data and resources.

3.2.3 *Android Security Overview by Google*

Android Security chief, Adrian Ludwing, presented the Google's approach to fight malware and statistical data regarding infected devices [13] . Android enforces several layers of protection since the user accesses Google Play until the application is running on its device. These layers were introduced as follows:

- Google Play;
- Unknown Sources Warning;
- Install Confirmation;
- Verify Apps Consent;
- Verify Apps Warning;
- Runtime Security Checks;

4 http://developer.android.com/tools/publishing/app-signing.html

- Sandbox and Permissions.

Google Play requires developer information and application signing. Furthermore, each application is reviewed before it becomes available. This process involves a set of procedures that checks static code and dynamic behaviors. After the analysis it is assigned a probability of threat tag to the application, being *Block*, *Warn* or *Allow*.

Android does not allow the installation of applications from unknown sources by default. This feature ensures that all installed applications had passed the Play Store test. If the user disables this rule by allowing unknown sources, the following alert is displayed *"Your phone and personal data are more vulnerable to attack by apps from unkown sources. You agree that your are solely responsible for any damage to your phone or loss of data that may result from using these apps"*. Also, the feature *Verify Apps*[5] inspects applications prior to install, applying an additional layer of security. If the application presents suspicious code, the installation process might be blocked, in sever cases, or triggers a warning. This is quite useful for those applications that skip the Google Play process review, i.e. were installed from third-party sources.

3.3 SUMMARY

This chapter presented general topics regarding the Android security model. In order to understand why Android lacks security mechanisms, it is necessary to know what mechanisms it already applies. The next chapter presents related work regarding the project's main subject.

5 https://support.google.com/accounts/answer/2812853?hl=en

4

RELATED WORK

DroidGuardian behaves as a firewall filtering the device's Internet traffic. *Firewall* is a technology designed to prevent unauthorized access to or from a private network. Commonly, firewalls are used to avoid illegitimate Internet users from accessing private networks connected to the Internet. In order to provide this security measure, all incoming and outgoing messages pass through the firewall, which analyzes each message and allows, denies or proxies this traffic according to defined rules.

In Linux systems, there is a mechanism built in the kernel that implements a firewall called *netfilter*[1]. The netfilter project was born in 1998 and was merged into the Linux kernel in the following year. It provides a traffic filtering mechanism by placing a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the software stack [14]. When an Internet packet traverses a netfilter hook, a registered callback is executed acting upon the packet according to specified rules. These rules can be configured using the user-space tools *iptables*, for ipv4, or *ip6tables*, for ipv6. These provide a table-based system for specifying firewall rules that can filter or transform packets.

DroidWall [2] is an Android application that takes advantage of the iptables firewall. It provides a simple front-end where users are able to define which applications are allowed to access the Internet and those that are not. When the user allows or denies a certain application to access the Internet, a new rule is added to the iptables accepting or rejecting the respective Process ID (PID) to communicate over the Internet.

Beyond traffic filtering, several mechanisms have been implemented to enforce fine-grained access policies regarding network connections on Android devices. *Aurasium*[3] provides a security feature that isolates each application in a custom sandbox controlling the access to the device's data and resources. It defines a set of rules and policies that ensures the device's protection. Aurasium provides a technology that repackages each `apk` file applying an intermediate layer that intercepts certain operation calls that use valueable resources, as Internet, SMS messages, IMEI, contact information, etc. Each access is conditioned by a set of policies which may be defined automatically or by the user. One of the most interesting features of this tool is the ability to enforce its policies without modifying the

1 http://www.netfilter.org
2 https://code.google.com/p/droidwall
3 http://www.aurasium.com

Android system. Aurasium places native code into the native framework layer. Its developers take advantage of certain methods that handle network connections. Independently from the level a network operation is called, it will fall on the `connect()` method in the `OSNetworkSystem` Java class. This method calls the `libnativehelper.so` library that transfers control to the `connect()` method of the `libc.so` library. The socket is able to get connected by libc delivering a system call to the Linux kernel. This path allows to create a check point above the Linux kernel in order to monitoring all operations called by the uppermost layers. By controlling the Bionic libc library, Aurasium is able to execute a set of policies that protect the entire Android system [15].

Malware take advantage of the Manifest permissions to grant access to resources that the user is not aware of when installing such apps. Even though these permissions are presented before the installation process, the user is not able to figure out if he is his valuable assets in danger. In order to apply a control level between Manifest permissions and the permissions Android apps really need, it was developed an interesting mechanism called *AppFence*. This tool inspects the app's Manifest permissions and allows users to decide which should be applied and those that should not, overtaking the obligation of accepting all permissions in order to install the app. In fact, all permissions are accepted, but instead of granting access to certain resources, AppFence replaces them with *shadow* data, which is an inoffensive imitation of those resources. For instance, if an app requires access to the contacts list and the user do not agree, the system will provide an empty list to the app. If the application is bad intentioned regarding that list, it will cause no harm to the user. AppFence also protects against data leakage through outgoing Internet connections. It acts at socket level by intercepting data buffers. When the malicious app is writing to a buffer, AppFence acts in one of two ways: tricks the app by indicating that the buffer was sent or emulates a state in which the device has no Internet connections.

DroidGuardian was inspired by a powerful tool called *Little Snitch*[4], that aims to raise awareness regarding Internet connection attempts from the system's applications. Little Snitch provides a graphical interface so that users can filter outgoing Internet connections[5] through rules and accept or reject connections in real time. In order to develop DroidGuardian, a deep study and understanding of Little Snitch took place and the following section will cover the relevant details. It might be important to stress the fact that Little Snitch is designed exclusively for the Mac OS X operating system and it is not open source. All information presented below stems from both the use of the tool and available documentation.

Since Android is an embedded Linux environment product, in the initial research phase to design Droidguardian we stumbled upon a very interesting tool, similar to Little Snitch, although much simpler, called *TuxGuardian*[6]. This tool aims to exhibit in real time every new outgoing Internet connection attempt from applications, allowing users to accept or reject such connections. TuxGuardian

4 http://www.obdev.at/products/littlesnitch/index.html

5 Later versions of Little Snitch allow to manage incoming Internet connections as well, but this feature is out of this project scope.

6 http://tuxguardian.sourceforge.net

was designed for Linux based operating systems and is open source, which led to a thorough analysis introduced later in this chapter.

4.1 LITTLE SNITCH

Little Snitch is by definition a firewall built for Mac OS X. However, it is not a regular firewall that operates at network packet level, checking protocol headers, but a firewall that acts at higher level, closer to the application layer. Little Snitch is set to intercept network connections attempts originated from all the system's applications and processes. Once a network connection attempt occurs in the system's kernel, it is intercepted by Little Snitch which will either accept it or reject it. This decision is based on a set of rules created by the user and by Little Snitch. The following section introduces Little Snitch rules.

4.1.1 *Little Snitch rules*

A rule is composed of four elements:

- Condition
- Action
- Lifetime
- Annotations

When an application, or Unix process, tries to establish an Internet connection, it passes to the system some required data, as an IP address, a port and an Internet protocol. These data is collected by Little Snitch that compares it to the existing rule. The *condition* field of each rule has the following properties:

- Process
- Process owner
- Server
- Port
- Protocol
- Direction
- Enabled

An Internet connection may include the following triplet: an IP address or *Server*, a *Port* and an Internet *Protocol*. Every connection request is launched by a certain process that is owned by another process, indicated by the *Process* and *Process owner* elements, respectively. Little Snitch handles IP address by using numeric sequences (the network addresses representation format that includes dots and colons), or Domain Name System (DNS) hostnames, or DNS domains. Protocols state the behavior of the Internet connection and Little Snitch presents the following: Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP). Processes are characterized by applications, such as *Safari*, *Mail*, etc, or Unix processes, such as *storeagent*, *ntpd*, etc. These processes are owned by an entity, as *System*, *root*, etc. There are two other properties that belong to conditions: *Direction* and *Enabled*. The first one indicates if the connection is incoming or outgoing and the last one may be seen as a flag that states if the rule is on or off.

Connection attempts are compared to these properties and, if a match occurs, the matched rule takes its action. A connection that matches a disabled rule is not handled. The action is one of the following:

- Allow
- Deny
- Ask

It's easy to understand that the rule may either allow or deny the connection. In the first case, the connection is established as if it was not intercepted by Little Snitch. In the second, the process attempting the connection receives an error - like a network failure - and the connection does not take place. The ask action is triggered when Little Snitch does not have the connection data stored, in a sequence of either being the first time the connection occurs or the user didn't want to save it earlier. Therefore, Little Snitch launches a dialog message, called *Connection Alert*, reporting the connection attempt, revealing the connection properties and providing choice buttons so that the user may decide what to do. Figure 6 shows a Little Snitch Connection Alert window. The figure reveals the Connection Summary, a short text indicating the server (*ax.init.itunes.apple.com*), the port (*80*) and the protocol (*http*); the Action, composed by the choice buttons *Allow* and *Deny*; the Rule Lifetime, where the user assigns a time tag to the rule; Rule Options to determine if this application (*iTunes*) is allowed to established every connection or if there are some restrictions regarding the server, port and protocol. At last, the Research Assist Button exhibits some detailed information about the connection's properties that may help users to decide what to do.

Rule Lifetime plays an important role. It allow users to define the frequency they want that connection to occur. He can choose one of the following tags:

- *Forever* - The rule never expires;
- *Until Quit* - The rule expires when the last instance of the process that matches the rule terminates;

Figure 6: Little Snitch Connection Alert window

- *Until Logout* - The rule expires when the user who created the rule logs out;
- *Until Restart* - The rule expires when the computer is restarted;
- *Minutes* - The rule expires a certain amount of time after it was created;
- *Once* - The connection takes places and the rule is not saved.

The descriptions above explain how Little Snitch perform. For instance, a *forever* rule will only display a Connection Alert once. All matching connections after the rule is set up will be executed according to the action's rule. On the other side, if the user chooses the *once* tag, is either allowing or denying the connection only this time and desires to be notified if it happen again. In this case, the Connection Alert will be prompt as if it was the first time this connection appears in the system.

As mentioned earlier, Little Snitch is a powerful tool. It has a mechanism to distinguish important processes that need to establish Internet connections in order to keep the system executing without problems. These processes are automatically granted permission to connect to external servers. However, the user may check the related rules and change them. For this reason, Little Snitch provides the *Annotation* field, in which rules are characterized as *Protected* or *Unapproved* to inform the user about their special status. Besides this feature, Little Snitch provides different profiles to each network the system is connected, and other useful features that make this tool quite robust and valuable.

4.1.2 *Little Snitch architecture*

A simple version of Little Snitch architecture is presented in Figure 7. At the bottom we find a Kernel Extension responsible for the interception of connection attempts. In OS X, the ability to refuse an Internet connection cannot be performed at user level. Therefore, Little Snitch developers were forced to operate at kernel level, building a Kernel Extension [16]. The collected data from the bottom layer

is sent to the layer above, the Network Filter. Rules are matched in this layer. At the top is placed the user interface that permit users to check information, define rules, etc.

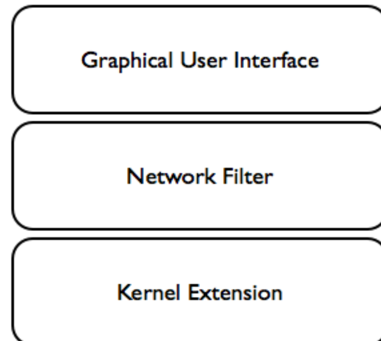

Figure 7: Little Snitch architecture

4.2 TUXGUARDIAN

TuxGuardian is an open source tool designed for Linux based operating systems, that intercepts outgoing Internet requests and triggers notification alerts to the user. Its basic behavior is quite similar to Little Snitch. Although this tool stopped being updated since 2006, it was very important in the scope of this project and played a major role in DroidGuardian's development process. In fact, that's where the name *DroidGuardian* came from. The following sections present TuxGuardian in detail.

4.2.1 *TuxGuardian architecture*

TuxGuardian is a host firewall that emerged to overcome the complexity of Linux security model to lay users, providing an interface to implement access control policies to the network outgoing traffic. It consists of a three layered architecture showed in Figure 8. Each layer has a specific function and establishes a communication to the next layer.

The *Security Module* is the bottom layer and takes advantage of the LSM framework to implement hook functions that grab Internet socket requests. Namely, TuxGuardian uses the callback functions `socket_create` and `socket_listen` to intercept both socket client and socket server Internet connection requests[7]. Local socket requests are not handled. Through this mechanism, TuxGuardian is able to block outgoing connections. In the same way as Little Snitch, this operation must be executed in kernel space. When the security module detects a connection attempt, sends a message to the layer above and waits a response in order to either deny or allow the connection.

7 For the sake of simplicity, we will not cover security modules framework in this chapter, but it will be detailed later in this document.

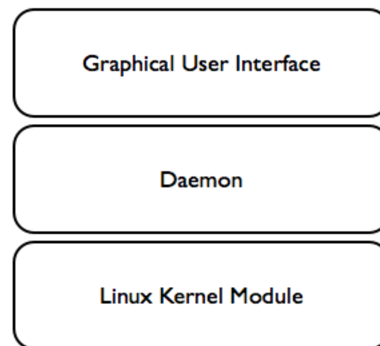

Figure 8: TuxGuardian architecture

The *Daemon* is by definition a program that executes in background waiting for some event to take place. In this case, it waits for the security module messages, that consists of the PID of the process that created the connection request. This communication process is established through local sockets. When the daemon gets the security module's message, checks the storage file to find a connection match. This procedure is also very similar to Little Snitch. If a match is found, TuxGuardian executes the corresponding action. Otherwise, it launches a notification window to get the user's response. Note that TuxGuardian is able to perform without the GUI component, denying all connections that are not placed in the storage file. TuxGuardian keeps the MD5 hash of each process path. Through PIDs, the daemon gets the process path name in the `/proc` directory and calculates its MD5 hash.

The GUI or *Frontend* displays the notification windows. The user receives the process name (the complete process path, for instance `/bin/ping`) that launched the connection attempt and decides to either accept it or reject it. Along with the process path, the corresponding MD5 hash is also displayed. Figure 9 presents the TuxGuardian notification window.

Figure 9: TuxGuardian notification window

4.2.2 *TuxGuardian Protocol*

The communication between layers is established through the TuxGuardian Protocol (TGP) [17]. This comprises a structure with the following fields:

- Sender
- Sequence number
- Query type
- Query data

Sender specifies the layer which sent the message:

- `TG_MODULE`,
- `TG_DAEMON`,
- `TG_FRONTEND`

corresponding to the security module, the daemon and the frontend, respectively. *Sequence number* acts as the message identifier. *Query type* characterizes the message, or query, as follows:

- `TG_ASK_PERMIT_APP` refers to the query sent by the security module to the daemon asking permission to either allow or deny the connection request;
- `TG_RESPOND_PERMIT_APP` refers to the response the security module gets from the daemon to the question above. *Query data* field stores the permission value;
- `TG_PERMIT_SERVER` refers to the first query, but indicating that the connection request involves a server;
- `TG_RESPOND_PERMIT_SERVER` refers to the response obtained from the previous question.

Depending on the nature of the query, *Query data* may store a PID or the permission values: either *yes* or *no*.

4.3 SUMMARY

This chapter presented several research works regarding Android mechanisms that try to apply fine-grained access control over Internet connections in Android devices. Outside the scope of the Android system, two tools designed for OS X and Linux that act as application firewalls were introduced. The next section covers the LSM framework.

5

LINUX SECURITY MODULES

The LSM framework was brought into the Linux kernel to provide a new approach regarding access control policies. Unfortunately, two factors are keeping developers away from getting into this framework: it presents a high degree of complexity and the most important, it lacks documentation. The LSM framework plays an important role in the scope of this project. Therefore, this chapter introduces this special framework, describing its origin and purposes, and presenting its internal mechanism that allows to build powerful security features.

5.1 INTRODUCTION

In 2001, Peter Loscocco and Stephen Smalley wrote an article introducing the SELinux [18]. They claim that the main reason that led to the development of such mechanism was the flawed assumption that security should reside in applications, leaving the role of the operating system behind [19]. Therefore, they supported the idea that secure applications require secure operating systems. A fundamental concept related to operating systems security is *access control policy*. In simple terms, this concept specifies what operations associated with an object are authorized to perform. Linux kernel inherited from the Unix security model the DAC that allows the owner of an object to set the security policy for that object (the control of access is based on the discretion of the owner). However, this model of access control brings some disadvantages. For instance, every program executed by a certain user receives all of the privileges associated with that user. It is able to change the permissions of all user's objects, creating potential security threats. In this sense, a MAC was purposed to protect the system against vulnerabilities left by other access control models. In MAC the operating system constrains the ability of a subject to perform an operation on an object, depending on its security attributes that can only be defined by an administrator and never by the user. Whenever a subject attempts to access an object, a permission rule enforced by the operating system kernel checks its security attributes in order to either allow or deny the access.

At the Linux 2.5 Kernel Summit, the NSA, based on the security issues mentioned above, presented their work on SELinux, a security mechanism of a flexible access control architecture in the Linux kernel [20]. NSA stated the need for such support in the mainstream Linux kernel. Other projects were presented to enforce access policies, namely Domain and Type Enforcement (DTE), Linux Intrusion

Detection System (LIDS) and POSIX.1e capabilities. Given these projects, Linus Torvalds decided to provide a general framework for security policy, called LSM. This framework allowed many different access control models to be implemented as loadable kernel modules. Linus said that LSM should be truly generic, where using a different security model was a question of loading a different kernel module. He also claimed that the framework should be conceptually simple, minimally invasive and efficient. At last, the mechanism should be able to support the POSIX.1e capabilities logic as an optional security module [21].

This security framework has motivated developers and gave them freedom to build their own LSM according to how they consider kernel objects should be accessed. SELinux[1] was originally developed by the NSA and has been in the mainstream kernel since version 2.6 (December 2003). In Linux every process and file has a label. SELinux uses these labels to mark executables when keeping track of permissions, that is why it is characterized as a labeling system. There are three forms of access control presented as follows:

- *DTE* is the primary model of enforcement and means that it is defined both the label on a process and the label on a filesystem object based on its type.
- *Role-Based Access Control (RBAC)* restricts the system access to authorized users by assigning roles that include specific permissions.
- *Multi-Level Security (MLS)* states that process control is based on the level of the data they will be using. For instance, a process of level B cannot read data of level A.

Smack (Simple Mandatory Access Control Kernel)[2] has been in the mainstream kernel since version 2.6.26 (July 2008). This module was implemented to provide simplicity to users. The complexity of DTE is avoided by defining access controls in terms of the access modes already in use.

AppArmor (Application Armor)[3] was originally developed by Immunix, which was a commercial operating system acquired by Novell in 2005. Novell laid off AppArmor programmers in 2007, but they continued the work. Since 2009, Canonical contributes to the project. This module has been in the mainstream Linux kernel since version 2.6.36 (October 2010). While SELinux is based on applying labels to files, AppArmor uses pathnames to make security decisions. For instance, two different security policies may be applied to the same file if that file is accessed by way of two different names. Many Linux administrators claim that AppArmor is the easiest security module to configure. Yet, others state that a pathname-based mechanism is insecure and that security policies should apply directly to objects (or to labels attached directly to objects) rather than to names given to objects.

TOMOYO Linux[4] is another MAC implementation for Linux. It has been in the mainstream kernel since version 2.6.30 (June 2009). This security mechanism follows the pathname-based philosophy,

1 http://selinuxproject.org
2 http://schaufler-ca.com
3 http://wiki.apparmor.net
4 http://tomoyo.sourceforge.jp

like AppArmor. TOMOYO Linux focuses on the behavior of a system, allowing each process to declare behaviors and resources needed to achieve its purpose. A precise comparison chart is available at TOMOYO webpage[5].

Recently, Yama[6] has been added to the mainstream kernel since version 3.4 (May 2012). Yama provides a few operations over and above the standard DAC, such as the protection against the creation of hardlinks to files that the user does not have access to, and the protection of process running as the same UID from being able to attach to each other and trace them using `ptrace`.

Since the first release of the LSM framework that new updates are committed in almost every new version of the Linux kernel. Even though the framework was built to provide different options to users through loadable modules, a radical upgrade was committed between version 2.6.25 and 2.6.27. The framework boot engine changed to turn LSM no longer a removable module, being loaded at compile time ever since. Following sections provide a technical description of the framework.

5.2 DESIGN

The basic abstraction of the LSM interface is to mediate the access to internal kernel objects. Security modules should answer a simple question *"May a subject `S` perform a kernel operation `Op` on an internal kernel object `Obj`?"*. The mechanism that allows modules to enforce this task lies in *hook* functions that are placed in the kernel. Figure 10 illustrates how hook functions intercede in the access to kernel objects, using as example the access to an *inode*.

Immediately before the kernel accesses the object, represented as *inode*, the hook makes a call to a function that the LSM provides. The module, based on policy rules, either allow or deny the access, forcing an error code return in the last case.

5.3 IMPLEMENTATION

The LSM framework comprises several files in the kernel filesystem which are listed in Figure 11.

In the root directory are placed the `include` and `security` folders. The first contains the main header file of the framework, while the second stores the source code regarding the various LSM models that are included in the kernel release. These files are introduced and discussed in the following sections.

5.3.1 *Header File*

The `include/linux/security.h` file contains the declaration of all hook functions. This declaration takes one out of two forms depending on the value of the `CONFIG_SECURITY` conditional

5 http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison
6 https://www.kernel.org/doc/Documentation/security/Yama.txt

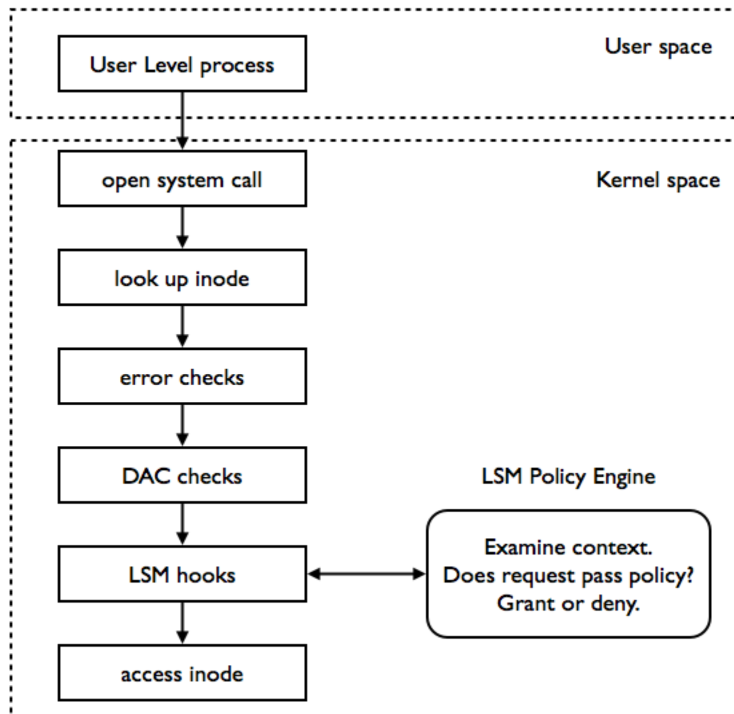

Figure 10: The LSM framework architecture (Accessing an inode)

group. This value is defined on the configuration file that specifies the initial settings of the kernel. If `CONFIG_SECURITY` is set to run with `Y` value, an extensive structure that includes pointers to all hook functions is declared. In case of the configuration item is set as `N`, it will not run and default functions will be declared driving the kernel to load the default security module.

Listing 5.1 presents a code snippet where the `security_operations` structure is declared along with some hook functions.

```
struct security_operations {
  char name[SECURITY_NAME_MAX + 1];
  int (*ptrace_access_check) (struct task_struct *child, unsigned int mode);
  int (*ptrace_traceme) (struct task_struct *parent);
  (...)
  int (*bprm_set_creds) (struct linux_binprm *bprm);
  int (*bprm_check_security) (struct linux_binprm *bprm);
  int (*bprm_secureexec) (struct linux_binprm *bprm);
  (...)
}
```

Listing 5.1: Code snippet of the `security_operations` structure (Linux kernel v3.11)

Within this structure hook functions are organized in groups. Each group is defined by a conditional item which value is specified on the kernel's configuration file, just like the configuration item

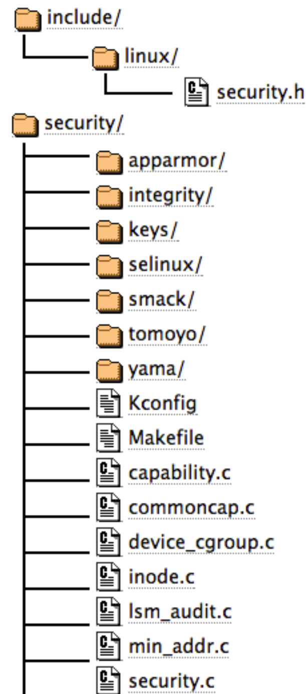

Figure 11: The LSM framework files in the Linux kernel filesystem

`CONFIG_SECURITY`. Depending on the value of each item, the corresponding hook functions are either declared inside the structure or assume a default action. These conditional groups are the following:

- `CONFIG_SECURITY_PATH`: includes security hooks for pathname based access control.
- `CONFIG_SECURITY_NETWORK`: enables socket and network security hooks.
- `CONFIG_SECURITY_NETWORK_XFRM`: security hooks for the Transformer (XFRM) framework that implements per-packet access controls based on labels derived from IPSec policy.
- `CONFIG_KEYS`: provides support for retaining authentication tokens and access keys in the kernel.
- `CONFIG_AUDIT`: enables auditing infrastructure that can be used with another kernel subsystem.

If the configurable option `CONFIG_SECURITY` is not selected, the default security module is loaded. This module only executes a few capabilities being permissive in all other hooks, which means that allows access to all kernel internal objects. An example of default capabilities is presented in Listing 5.2.

```
static inline int security_capable(
  const struct cred *cred,
  struct user_namespace *ns, int cap)
{
  return cap_capable(cred, ns, cap, SECURITY_CAP_AUDIT);
}

static inline int security_capable_noaudit(
  const struct cred *cred,
  struct user_namespace *ns,
  int cap)
{
  return cap_capable(cred, ns, cap, SECURITY_CAP_NOAUDIT);
}
```

Listing 5.2: Code snippet of default security functions (Linux kernel v3.11)

The same process is kept to the other configurable options. Depending on their values, security hooks are either declared or coded with default instructions.

5.3.2 *Linux Capabilities*

Linux capabilities were designed to provide a solution to the Unix user-based privilege model composed by privilege users (root) and non-privilege users (regular user). The first type has permission to execute every operation and the former can only execute a few set of operations. Therefore, processes run either with all permissions or with very restrictive permissions. However, most of the time processes do not need all privileges to execute a task and this exposure raises serious risks when a process gets compromised [22]. To solve this issue, a set of functions called *common capabilities* were built to give the security framework a default behavior in case of no LSM model is chosen. The related source code is written in the `security/commoncap.c` file. For instance, the source code of the capability function declared in Listing 5.2, is defined in this file and a code snippet is presented in Listing 5.3

```
int cap_capable(const struct cred *cred,
        struct user_namespace *targ_ns,
        int cap,
        int audit)
{
  struct user_namespace *ns = targ_ns;

  /* See if cred has the capability in the target user namespace
  * by examining the target user namespace and all of the target
   * user namespace's parents.
  */
```

```
  for (;;) {
                /* Do we have the necessary capabilities? */
                if (ns == cred->user_ns)
                        return cap_raised(cred->cap_effective, cap) ? 0 : -EPERM;

                /* Have we tried all of the parent namespaces? */
                 if (ns == &init_user_ns)
                         return -EPERM;
 (...)
}
```

Listing 5.3: Code snippet of the cap_capable() function (Linux kernel v3.11)

When the kernel is not loaded with a LSM, there must be a default module that does not execute any operation and let processes access kernel internal objects as if there were no hook functions. The security/capability.c file sets all hook functions with default instructions. If the return type of the functions is void they have an empty body, i. e. they don't have instructions. Otherwise they return the int value of 0, which turns the hook as permissive. Listing 5.4 shows some of these default hook functions.

```
static int cap_syslog(int type)
{
  return 0;
}

static int cap_quotactl(int cmds, int type, int id, struct super_block *sb)
{
  return 0;
}

static int cap_quota_on(struct dentry *dentry)
{
  return 0;
}
```

Listing 5.4: Code snippet of capability functions (Linux kernel v3.11)

These functions are called in the security_operations structure if the corresponding hook functions are not declared. Listing 5.5 presents the code snippet of the function security_fixup_ops.

```
#define set_to_cap_if_null(ops, function)        \
  do {                  \
    if (!ops->function) {          \
      ops->function = cap_##function;      \
      pr_debug("Had to override the " #function \
         " security operation with the default.\n");\
```

```
      }            \
  } while (0)

void __init security_fixup_ops(struct security_operations *ops) {
  set_to_cap_if_null(ops, ptrace_access_check);
  set_to_cap_if_null(ops, ptrace_traceme);
  set_to_cap_if_null(ops, capget);
  set_to_cap_if_null(ops, capset);
  set_to_cap_if_null(ops, capable);
  set_to_cap_if_null(ops, quotactl);
  set_to_cap_if_null(ops, quota_on);
  set_to_cap_if_null(ops, syslog);
  set_to_cap_if_null(ops, settime);
  set_to_cap_if_null(ops, vm_enough_memory);
(...)
}
```

Listing 5.5: Code snippet of the `security_fixup_ops()` function (Linux kernel v3.11)

5.3.3 *Framework Initialization*

The header file mentioned in Section 5.3.1 declares some functions in charge of getting the LSM loaded as shown in Listing 5.6.

```
/* prototypes */
extern int security_init(void);
extern int security_module_enable(struct security_operations *ops);
extern int register_security(struct security_operations *ops);
extern void __init security_fixup_ops(struct security_operations *ops);
```

Listing 5.6: Code snippet of the initialization functions (Linux kernel v3.11)

These functions are implemented in the `security/security.c` file. The first function being executed is the `security_init()` function. A code snippet is present in Listing 5.7.

```
static __initdata char chosen_lsm[SECURITY_NAME_MAX + 1] =
  CONFIG_DEFAULT_SECURITY;
static struct security_operations *security_ops;
static struct security_operations default_security_ops = {
  .name = "default",
};
(...)
int __init security_init(void) {
  printk(KERN_INFO "Security Framework initialized\n");

  security_fixup_ops(&default_security_ops);
```

```
  security_ops = &default_security_ops;
  do_security_initcalls();

  return 0;
}
```

Listing 5.7: Code snippet of the `security_init()` function (Linux kernel v3.11)

At first, the default module is loaded with the available routines presented in Section 5.3.2 by the `security_fixup_ops(&default_security_ops)` instruction. Then the `security_init()` function updates the kernel's security `security_ops` structure with the data earlier initialized and makes a call to `do_security_initcalls()` that implements the loop presented in Listing 5.8.

```
static void __init do_security_initcalls(void)
{
  initcall_t *call;
  call = __security_initcall_start;
  while (call < __security_initcall_end) {
    (*call) ();
    call++;
  }
}
```

Listing 5.8: Code snippet of the `do_security_initcalls()` function (Linux kernel v3.11)

The `__security_initcall_start` and `__security_initcall_end` callbacks are declared in the `include/linux/init.h` header file and the code snippet is shown in Listing 5.9.

```
/*
 * Used for initialization calls..
 */
typedef int (*initcall_t)(void);
typedef void (*exitcall_t)(void);

extern initcall_t __con_initcall_start[], __con_initcall_end[];
extern initcall_t __security_initcall_start[], __security_initcall_end[];
```

Listing 5.9: Code snippet of the `init` callbacks (Linux kernel v3.11)

Modules Registration

There are several LSM implementations adopted by the kernel, but it only runs one at a time. Therefore, there must be a way to register the desired LSM. This is achieved through the execution of

the `register_security(struct security_operations *ops)` instruction presented in Listing 5.10.

```
int __init register_security(struct security_operations *ops)
{
  if (verify(ops)) {
    printk(KERN_DEBUG "%s could not verify "
           "security_operations structure.\n", __func__);
    return -EINVAL;
  }

  if (security_ops != &default_security_ops)
    return -EAGAIN;

  security_ops = ops;

  return 0;
}
```

Listing 5.10: Code snippet of the `register_security()` function (Linux kernel v3.11)

Some rudimentary check is done on the `ops` structure by the `verify(struct security_operations *ops)` instruction. If there is already a security module registered on the kernel, an error will be returned. Otherwise, the `security_ops` structure gets the hook functions of the `ops` structure and return success.

There is another important function related to the LSM registration called `security_module_enable()`. Each LSM must pass this function before registering its own operations to avoid security registration races. This function may also be used to check if the LSM is currently loaded during kernel initialization. Listing 5.11 presents the code snippet of this function.

```
int __init security_module_enable(struct security_operations *ops)
{
  return !strcmp(ops->name, chosen_lsm);
}
```

Listing 5.11: Code snippet of the `security_module_enable()` function (Linux kernel v3.11)

At last, the security functions mentioned in Section 5.3.1 are implemented by returning the function callback present in the `security_operations` structure. A code snippet is illustrated in Listing 5.12.

```
int security_socket_create(int family, int type, int protocol, int kern)
{
  return security_ops->socket_create(family, type, protocol, kern);
```

```
}

int security_socket_post_create(struct socket *sock, int family,
        int type, int protocol, int kern)
{
  return security_ops->socket_post_create(sock, family, type,
            protocol, kern);
}

int security_socket_bind(struct socket *sock, struct sockaddr *address, int
    addrlen)
{
  return security_ops->socket_bind(sock, address, addrlen);
}

int security_socket_connect(struct socket *sock, struct sockaddr *address, int
    addrlen)
{
  return security_ops->socket_connect(sock, address, addrlen);
}
```

Listing 5.12: Code snippet of some security functions (Linux kernel v3.11)

5.3.4 *Security Functions in the Kernel*

The security functions presented in the previous section are called depending on the objective. For instance, the `socket_create()` hook is part of the socket implementation in the `net/socket.c` file. Note the code snippet in Listing 5.13.

```
int sock_create_lite(int family, int type, int protocol, struct socket **res)
{
  int err;
  struct socket *sock = NULL;

  err = security_socket_create(family, type, protocol, 1);
  if (err)
    goto out;
(...)
}

int __sock_create(struct net *net, int family, int type, int protocol,
       struct socket **res, int kern)
{
  int err;
  struct socket *sock;
```

```
  const struct net_proto_family *pf;
(...)
  err = security_socket_create(family, type, protocol, kern);
  if (err)
    return err;
(...)
}
```

Listing 5.13: Code snippet of the `socket_create()` hook in socket implementation (Linux kernel v3.11)

This hook is simply a flag in which the returned value is checked and if it is different from `0`, the kernel blocks the socket creation. That is the reason why the default capability functions always return `0`.

5.4 SUMMARY

This chapter provided a detailed description of the LSM framework. It explains how it is possible to build custom LSM, showing several code snippets to allow a better understating of the framework. The next chapter introduces several technical concepts related to the nature of the project.

6

TECHNICAL CONCEPTS

The development process of DroidGuardian required the study and understanding of some technical concepts. This chapter introduces these concepts along with relevant details to the scope of the project.

6.1 LOADABLE KERNEL MODULES

In Linux systems it is possible to develop kernel routines and run them as if they were part of the kernel. This is accomplished through Loadable Kernel Module (LKM) which are programs written specifically to the kernel that can be loaded at runtime. This feature brings a lot of advantages to developers enabling them to access low level resources from the kernel. Simple modules are easily written and installed. However, they are usually built to perform complex tasks at kernel level, leading to complex source code. When playing with kernel modules, developers must ensure that the code is not corrupted in any way, otherwise the system may stop abruptly leading to an unrecoverable state (usually designated as kernel panic).

6.1.1 *Building*

In order to get a LKM to run on the kernel, the user must provide the *entry* and *exit* points. The former is called when the module is inserted into the kernel. The last is called when the module is removed from the kernel.

The entry point is implemented as a function that is declared as `static` and should return the `int` value of 0. This *init* function may have any name, and is defined as the entry point through the `module_init()` primitive. For instance, if the init function is called *"hello"*, the entry point is defined as follows:

```
module_init(hello);
```

Listing 6.1: Defining the Loadable Kernel Module's entry point

The exit point is implemented as a function that is also declared as `static` but returns `void`. Similar to the entry point, the exit function may assume any name, but this should be passed as

argument to the `module_exit()` primitive. For instance, naming the exit function as *"goodbye"*, the exit point is defined as follows:

```
module_exit(goodbye);
```

Listing 6.2: Defining the Loadable Kernel Module's exit point

6.1.2 *Compiling*

The compilation process is accomplished using the *make* utility. The developer should build a *Makefile* providing the path to both the module's location and the kernel's libraries that will generate the `.ko` file. A simple example of such *Makefile* is presented as follows:

```
obj-m := example_module.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Listing 6.3: Example of a Makefile to compile Loadable Kernel Modules

The source code file of the module above is called `example_module.c` and the originated module file will be called `example_module.ko`. Note the `uname -r` command that will give the kernel's version so that the compiled module is able to run in the same kernel. Developers must pay attention to kernel's versions in order to build compatible modules.

6.1.3 *Inserting and Removing*

Linux provides several commands to deal with kernel modules. To insert modules into the kernel, developers use the `insmod` command that takes a `.ko` file as argument. For instance, to insert the module presented above, the following command is executed:

```
# sudo insmod example_module
```

Listing 6.4: Linux command to insert Loadable Kernel Modules

After executing this command, the module starts to run immediately by calling the entry point function. To remove the module, the following command must be executed:

```
# sudo rmmod example_module
```

Listing 6.5: Linux command to remove Loadable Kernel Modules

6.2 INTERPROCESS COMMUNICATION USING SOCKETS

Unix provides several solutions to perform IPC. Sockets use file descriptors to fulfill this task. This section introduces relevant details regarding both Internet sockets and local (Unix domain) sockets focusing on the stream socket type. Also, sockets are handled differently regarding the virtual memory of the system: user space and kernel space. Firstly we'll present a simple server-client model implementation at user space followed by some particularities of socket implementation at kernel level.

6.2.1 *Stream Sockets*

Unix systems provide a programming interface to easily carry out IPC tasks using sockets. This API is present in the `sys/socket.h` header file. Sockets follow a server-client based model, in which a sequence of primitives needs to be invoked in order to established the connection. This sequence depends on the protocol that will take place. Usually, sockets fall into the TCP or UDP protocols. Both require different primitives to settle connections. In the scope of this project, only stream sockets are used. Therefore, this section will focus on the basic behavior of stream sockets. Figure 12 illustrates a typical case and may be described as follows:

1. The server initializes the process by creating a file descriptor (socket descriptor). This process is accomplished through the `socket()` primitive:

   ```
   int socket(int domain, int type, int protocol);
   ```

   Listing 6.6: Declaration of the `socket()` function

   The returned value defines the socket descriptor. As arguments, *domain* specifies the socket family (`AF_INET`, `AF_INET6`, `AF_UNIX`, etc), *type* specifies the socket type (`SOCK_DGRAM`, `SOCK_STREAM`, etc) and *protocol* indicates a particular protocol to be used with the socket, but usually takes the value 0.

2. Once created, the socket is unnamed and needs to be bound to an address in order to be identified by the system. This address will be assigned depending on the socket family. The `bind()` primitive is presented as follows:

   ```
   int bind(int socket, const struct sockaddr *address, socklen_t address_len);
   ```

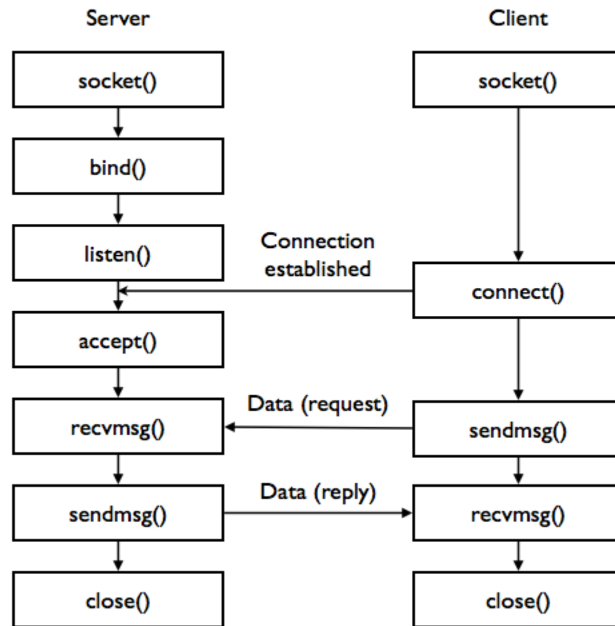

Figure 12: Typical server-client based model of stream sockets

Listing 6.7: Declaration of the `bind()` function

If the returned value is 0, the operation was successful. In case of error, returns -1. The argument *socket* specifies the socket descriptor previously created, the *address* points to the address to be bound to the socket and *address_len* indicates the length of the address structure.

3. After the binding, the server is ready to establish a connection to a client. Thus, the server is kept listening to connection requests through `listen()`:

```
int listen(int socket, int backlog);
```

Listing 6.8: Declaration of the `listen()` function

The function expresses the success or failure of the operation through the returned value, being 0 or -1, respectively. It takes as arguments the file descriptor and a *backlog* that defines the length of the socket's listen queue, where connection requests are stored.

4. At this point, the server is waiting for some request from a client. To set up a client socket, primarily it is executed the `socket()` primitive to create a file descriptor.

5. Once the socket descriptor is created, the client must specify the server address to get connected. The `connect()` primitive is used:

```
int connect(int socket, const struct sockaddr *address, socklen_t
    address_len);
```

Listing 6.9: Declaration of the `connect()` function

It returns 0 on success or -1 on error. The *socket* indicates the client socket descriptor, the *address* points to the server address and *address_len* defines the length of the address.

6. The server receives the connection request and is able to accept it through the `accept()` primitive:

```
int accept (int socket, struct sockaddr *address, socklen_t *address_len);
```

Listing 6.10: Declaration of the `accept()` function

This primitive returns a newly connected socket descriptor. The *address* is filled with the address of the client and *address_len* defines the length of this address. Both sockets are ready to start the communication.

7. The client and server may exchange data through some primitives. In this case, we'll introduce `sendmsg()` and `recvmsg()`:

```
ssize_t sendmsg (int socket, const struct msghdr *message, int flags);

ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

Listing 6.11: Declaration of the `sendmsg()` and `recvmsg()` function

These primitives use a special structure to store data in the *message* argument, that is the `struct msghdr`. Further in this section we'll inspect this structure. The *flags* argument specifies some conditions such as, for instance, blocking the function until the total amount of data requested is returned, by the flag `MSG_WAITALL`. The total amount of data exchanged is stored on the returned value.

8. At last, when all data has been exchanged both sockets need to close its connections by calling the `close()` primitive:

```
int close(int fildes);
```

Listing 6.12: Declaration of the `close()` function

The socket descriptor is passed as argument.

6.2.2 *Address Formats*

As previously mentioned, in the primitives `bind()`, `connect()` and `accept()` the argument *address* points to a `sockaddr` structure based on the socket's family. If we want to communicate through Internet sockets, the family is defined as `AF_INET` or `AF_INET6`, depending on the IP version, IPv4 or IPv6, respectively, and a `sockaddr_in` or `sockaddr_in6` structure is used to handle Internet addresses:

```
struct sockaddr_in {
    short          sin_family;
    unsigned short  sin_port;
    struct in_addr  sin_addr;
    char           sin_zero[8];
}
```

Listing 6.13: Declaration of the `sockaddr_in` structure

This structure defines the required data to create an Internet address: the port and the IP address [23]. These fields are specified by `sin_port` and `sin_addr`, respectively. The former is stored as an `unsigned short`. The last is defined by a `in_addr` structure that contains an `unsigned long` to store the IP address value:

```
struct in_addr {
    unsigned long s_addr;
}
```

Listing 6.14: Declaration of the `in_addr` structure

When it concerns the `AF_INET6` family, sockets use the `sockaddr_in6` structure:

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;
    in_port_t       sin6_port;
    uint32_t        sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t        sin6_scope_id;
}
```

Listing 6.15: Declaration of the `sockaddr_in` structure

The element `sin6_family` is defined by the `AF_INET6` macro, the `sin6_port` specifies the protocol port, `sin6_flowinfo` and `sin6_scope_id` characterize identifiers of the flow and the address, respectively and, at last, the `sin6_addr` defines the IP address through a `in6_addr` structure. This structure presents an `unsigned char` array that stores the IP address:

```
struct in6_addr {
```

```
    unsigned char   s6_addr[16];
}
```

Listing 6.16: Declaration of the `in6_addr` structure

These structures are declared in the `netinet/in.h` header file.

In local sockets the family is defined by `AF_UNIX` and the address is set using a `sockaddr_un` structure:

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;
    char     sun_path[UNIX_PATH_MAX];
}
```

Listing 6.17: Declaration of the `sockaddr_un` structure

The address is defined by the path of a file stored in `sun_path`. In the scope of this project, there are two types of paths (called namespaces) that are important to distinguish:

- *Pathname*: a null-terminated filesystem pathname is bound to the local socket.
- *Abstract*: the `sun_path[0]` is a null byte. The socket's address in this namespace is given the additional bytes in `sun_path`. The name has no connection to the filesystem pathnames[1].

The `sockaddr_un` structure is declared in the `sys/un.h` header file.

6.2.3 *Address Lookup*

Sockets store IP addresses as `unsigned longs` or `unsigned char` arrays, but they are displayed to users through the dotted notation: `x.x.x.x` in case of IPv4 or `x:x:x:x:x:x:x:x` in case of IPv6. In order to translate Internet socket addresses to the user's reading format, the `arpa/inet.h` header file provides the following function:

```
const char *inet_ntop(int af, const void *restrict src, char *restrict dst,
    socklen_t size);
```

Listing 6.18: Declaration of the `inet_ntop()` function

This function takes as arguments the Internet family in *af* (`AF_INET` or `AF_INET6`); *src* points to a buffer holding a `struct in_addr` or a `struct in6_addr`; *dst* points to the destination string and *size* indicates the maximum length of this string.

1 http://man7.org/linux/man-pages/man7/unix.7.html

Using Internet addresses is also possible to get the both the host and service name through the `getnameinfo()` function, declared on the `netdb.h` header file:

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Listing 6.19: Declaration of the `getnameinfo()` function

6.2.4 *Kernel Sockets*

In kernel space, the server-client based model is the same, but the primitives are different. In order to understand how socket primitives are handled in kernel space it was necessary to check the Linux Cross Reference[2]. Sockets are created through the `sock_create()` primitive, declared in the `linux/net.h` header file:

```
int sock_create(int family, int type, int proto, struct socket **res);
```

Listing 6.20: Declaration of the `sock_create()` function

The first three arguments are similar to the `socket()` primitive described above. Kernel creates a socket by allocating memory to a `struct socket` and filling it in with the following data:

```
struct socket {
   socket_state state;
   short type;
   unsigned long flags;
   struct socket_wq __rcu  *wq;
   struct file * file;
   struct sock * sk;
   const struct proto_ops * ops;
}
```

Listing 6.21: Declaration of the `socket` structure

From these structure's fields it is important to highlight the following: *type* that indicates the socket type (`SOCK_STREAM`, `SOCK_DGRAM`, etc); *sk* that specifies all internal networking protocol and is an agnostic socket representation, i. e. the same structure is used by any socket independently of its type or family; and *ops* that defines the socket operations. Once the `sock_create()` primitive is executed, the socket data is stored at *res*.

This socket will execute the remaining operations through the `struct proto_ops` presented in the `struct socket` by means of *ops* field:

2 http://lxr.free-electrons.com

```
struct proto_ops {
    int family;
    struct module   *owner;
    int (*release) (struct socket *sock);
    int (* bind) (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len)
        ;
    int (* connect) (struct socket *sock, struct sockaddr *vaddr, int
        sockaddr_len, int flags);
    int (* accept) (struct socket *sock, struct socket *newsock, int flags);
    int (* listen) (struct socket *sock, int len);
(...)
}
```

Listing 6.22: Declaration of the `proto_ops` structure

All primitives, `bind()`, `connect()`, `listen()`, `accept()`, and `release()`, which is the kernel implementation of `close()`, are called through this structure that belongs to the socket. They are the kernel implementation of those forementioned primitives in user space and take almost the same arguments, but instead of using the socket descriptor, they point to the socket structure in *sock*.

To send and receive data, kernel declares the `sock_sendmsg` and `sock_recvmsg` primitives, respectively:

```
int sock_sendmsg (struct socket *sock, struct msghdr *msg, size_t len);

int sock_recvmsg (struct socket *sock, struct msghdr *msg, size_t size, int flags
    );
```

Listing 6.23: Declaration of the `sock_sendmsg()` and texttttsock_recvmsg functions

These primitives also take the `struct msghdr` as argument. This structure is used to store the data that is exchanged in each sending and receiving process. It is declared in the `linux/socket.h` header file and has the following fields:

```
struct msghdr {
    void* msg_name;
    int  msg_namelen;
     struct iovec*  msg_iov;
     __kernel_size_t msg_iovlen;
     void* msg_control;
     __kernel_size_t msg_controllen;
     unsigned int msg_flags;
}
```

Listing 6.24: Declaration of the `msghdr` structure

The first two elements are normally used in datagram exchange. The *msg_flags* field indicates several characteristics of the data received. The *msg_iov* represents an array of buffers that contains or points to the data that is sent and received. The *msg_iovlen* defines the length of the `struct iovec` used.

The `struct iovec` stores data as follows:

```
struct iovec {
    void* iov_base;
    size_t iov_len;
}
```

Listing 6.25: Declaration of the `iovec` structure

The *iov_base* field points to the initial element of the data being passed and *iov_len* defines its length. This structure is used, because it allows to store data in different memory locations, providing a *scatter* feature, optimizing the use of memory [24]. Also, the read operation applies a *gather* feature, collection all spread data.

6.3 ANDROID TOOLS

The Android Software Development Kit (SDK) provides useful tools regarding the development and study of Android apps. It is the case of the *emulator*, placed at `tools/` and the Android Debug Bridge (ADB), placed at `platform-tools/`. This section describes both tools regarding their valuable features to this project.

6.3.1 *Android Emulator*

Android supplies a mobile device emulator based on the Qemu virtual machine that runs on the computer. This emulator provides a real Android environment, being able to run any application. It is very useful to developers, because avoids the need of having a real device in order to run applications. However, depending on the computer's characteristics, the performance of the Android emulator may be considerable low when compared to real devices.

The Android emulator boots an Android image according to the Android Virtual Device (AVD) configuration file. The AVD allows to define hardware and software characteristics of a specific model to run on the Android emulator. Figure 13 shows a snapshot of the AVD window configuration. For instance, in the *Device* option it is possible to choose an Android model, as *Nexus 4*, *Nexus 7*, *Nexus 10*, *Galaxy Nexus*, *Nexus S*, etc. The *Target* element defines the Android version and the corresponding API, as *Android 2.3.3 - API Level 10*, *Android 4.4 - API Level 19*, etc.

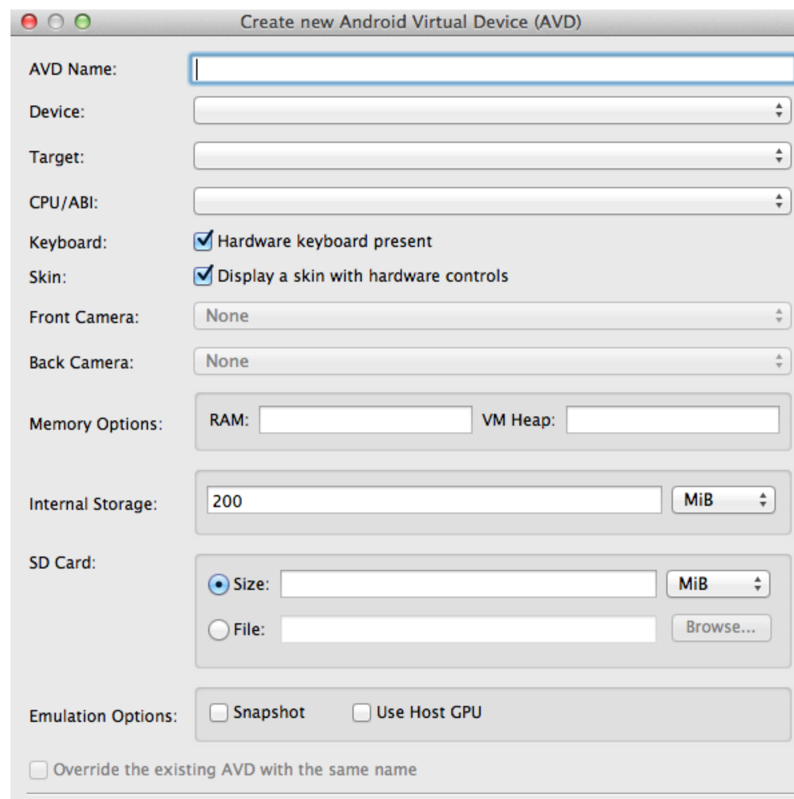

Figure 13: Android Virtual Device configuration

Once the AVD is created, the emulator may be launched through the AVD Manager or using the command line. Some useful commands[3] are presented as follow.

```
# emulator -avd <avd_name>
```

Listing 6.26: Command to start the Android emulator

This command launches the emulator with the AVD image called `avd_name`. AVD files are usually stored at `.android/avd/` within the Android SDK folder.

```
# emulator -avd <avd_name> -kernel <kernel_path>
```

Listing 6.27: Command to provide a kernel to the Android emulator

In order to choose a kernel of our own, to run on the emulator, it is used the `-kernel` flag followed by the path of the image kernel file.

```
# emulator -avd <avd_name> -kernel <kernel_path> -show-kernel -verbose
```

Listing 6.28: Command to provide kernel prints of the Android emulator

3 http://developer.android.com/tools/help/emulator.html

To follow what is happening during the boot and to inspect kernel prints, the emulator provides both `-show-kernel` and `-verbose` flags.

6.3.2 *Android Debug Bridge*

ADB[4] is another useful tool that connects the computer to Android devices (real or emulated). This connection brings powerful features that will be described in this section. The ADB tool, mention as adb from now on, is available as a command line. It is a client-server program that comprises three components:

- A client, that runs on the development computer;
- A server, that runs as a background process on the development computer. The server handles communication between the client and the daemon;
- A daemon, that runs in background on the mobile device (real or emulated).

Once Android emulator is started, adb provides a means of communication. The following command shows all Android devices running on the computer:

```
# adb devices
```

Listing 6.29: Command to show Android devices running on the computer

If there is an Android emulator running on the computer, the output returned is similar to the following:

```
List of devices attached
emulator-5554 device
```

Listing 6.30: Example of the output from the `adb devices` command

With adb it is possible to:

- install an Android application on the emulator/device;

  ```
  # adb install <path_to_apk>
  ```

  Listing 6.31: Command to install Android apps using the adb utility

- copy a specific file from the emulator/device to the development computer;

4 http://developer.android.com/tools/help/adb.html

```
# adb pull <remote> <local>
```

Listing 6.32: Command to pull files using the adb utility

- copy a specific file from the development computer to the emulator/device;

```
# adb push <local> <remote>
```

Listing 6.33: Command to push files using the adb utility

- print the logcat output;

```
# adb logcat
```

Listing 6.34: Command to get logcat's prints using the adb utility

- start a remote shell in the target emulator/device:

```
# adb shell
```

Listing 6.35: Command to start a remote shell using the adb utility

There are more options to execute with adb that can be checked on the Android online page[5].

6.4 ANDROID NDK AND JNI

Android provides a powerful toolset that has multiple purposes, called Native Development Kit (NDK). The Android NDK was built to supply developers the capability to exploit the full power of mobile devices using native code. This is achieved through the JNI, which is a programming framework that provides connection between Java code that runs on the virtual machine and native code, as C/C++. Native code is accessed by the Java side as a static library, declared through the following statement:

```
static {
  System.loadLibrary("native");
}
```

Listing 6.36: Declare a JNI library in Java

This library called `native` implements a set of native methods called in Java. For instance, considering that the library implements two native methods named `nativeMethodA()` and `nativeMethodB()`. The following statements declare these methods:

5 http://developer.android.com/tools/help/adb.html

```
public native void nativeMethodA();
public native String nativeMethodB(String str);
```

Listing 6.37: Declare native methods

At this point, Java knows that in order to execute the `nativeMethodA()` and the `nativeMethodB()` methods it has to inspect the `native` library stored as `libnative.so` and placed at `libs/armeabi/` in the Android project folder.

Developers are advise to create a new folder under the Android project directory, called `jni`. The native library consists of, at least, three files that should be placed at a this `jni` folder:

- the `Android.mk` configuration file;
- the header file;
- the C/C++ file.

The `Android.mk` file comprises several configurations required by the *ndk-build* tool. This tool is brought by the Android NDK and allows to compile native code generating library files as well as executable files. The minimum set of instructions in the `Android.mk` file is presented as follows:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := native
LOCAL_SRC_FILES := native.c

include $(BUILD_SHARED_LIBRARY)
```

Listing 6.38: The minimum set of instructions in the `Android.mk` file

This file specifies:

- the native source files location in `LOCAL_PATH`;
- the name of the library in `LOCAL_MODULE`;
- and the name of the native code file in `LOCAL_SRC_FILES`.

The statement `CLEAR_VARS` indicates that no dependent configuration disrupts compilation [25]. At last, the `include $(BUILD_SHARED_LIBRARY)` statement instructs the `ndk-build` tool to build a shared library. As additional note, if the native code was intended to generate an executable file, the last instruction would be replaced by `include $(BUILD_EXECUTABLE)`.

The header file name follows the pattern: `<package_name>_<class>.h`. For instance, considering that the Java class responsible for loading the library is called `LoadLibrary` and the Java package that contains this class is called `com.android.droidguardian`. The name of the header file would be: `com_android_droidguardian_LoadLibrary.h`. This file is automatically generated by a tool called *javah* provided by the Java Development Kit (JDK). It was designed to build header files to the JNI and may be used as follows:

```
# javah -jni -d <path_to_jni_folder> -classpath <path_to_class_files> \
com.android.droidguardian.LoadLibrary
```

Listing 6.39: Example of use of the javah tool

This tool operates over `.class` files which means that the Java code must be previously compiled.

The native code goes on regular `.c/.cpp` files. The next section will introduce basic JNI concepts in order to get a native library running on Java.

6.4.1 *Java Native Interface concepts*

The JNI englobes a wide set of features to handle native code by means of a Java library. Starting with data types, the JNI maps primitive Java data types to native data types as shown in Table 2.

Java Type	**JNI Type**	**C/C++ Type**	**Size**
Boolean 1.5	Jboolean	unsigned char	Unsigned 8 bits
Byte	Jbyte	char	Signed 8 bits
Char	Jchar	unsigned short	Unsigned 16 bits
Short	Jshort	short	Signed 16 bits
Int	Jint	Int	Signed 32 bits
Long	Jlong	long long	Signed 64 bits
Float	Jfloat	float	32 bits
Double	Jdouble	double	64 bits

Table 2: JNI primitive data types mapping

Regarding reference types, the JNI also maps Java and native types, as shown in Table 3.

Java Type	JNI Type
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.String	jstring
Other objects	jobjects
java.lang.Object[]	jobjectArray

Table 3: JNI reference types mapping

Strings

Strings cannot be converted directly from Java into its native type. Therefore, there are specific methods to handle Strings, as well as other reference types. For instance, to convert a C string into a Java string the following instructions need to take place:

```
jstring javaString;
javaString = (*env)->NewStringUTF(env, "Hello World!");
```

Listing 6.40: Creating a new Java string from a given C string

Note that along with UTF strings, also Unicode strings are supported by the JNI. The `javaString` variable will be used by Java as a string that contains the value *"Hello World!"*.

When the conversion needs to occur in the reverse order, the following steps need to be executed:

```
char* str;
str = (*env)->GetStringUTFChars(env, javaString, NULL);
(*env)->ReleaseStringUTFChars(env, javaString, str);
```

Listing 6.41: Creating a new C string from a given Java string

After converting the `javaString` into a new C string called `str`, the Java string needs to be released by calling the `ReleasingStringUTFChars()` function.

Java fields

The variables of Java classes, also called fields, may be accessed through the JNI. There are two types of fields: instance fields and static fields. When a class is instantiated, its instance variables are copied, but its static variables are not, which means that all instances of the class share the same static fields. As example, considering that a certain Java class presents the following fields:

```
private String instanceField = "Instance Field";
private static String staticField = "Static Field";
```

Listing 6.42: Declaring Java fields

The first variable, `instanceField`, is accessed in the native code as follows:

```
jclass clazz;
clazz = (*env)->GetObjectClass(env, instance);
jfieldID instanceFieldId;
instanceFieldId = (*env)->GetFieldID(env, clazz, "instanceField", "Ljava/lang/
    String;");
```

Listing 6.43: Accessing a Java instance field

At first, it was necessary to get the class object in order to inspect its fields. Then, the native type `jfieldID` assumes the native form of the Java field `instanceField`. When using static fields, the function is very similar:

```
jclass clazz;
clazz = (*env)->GetObjectClass(env, instance);
jfieldID staticFieldId;
staticFieldId = (*env)->GetStaticFieldID(env, clazz, "staticField", "Ljava/lang/
    String;");
```

Listing 6.44: Acessing a Java static field

Once Java fields are converted into native fields, it is possible to get or set its values:

```
jstring jstr;
char *str;
jstr = (*env)->GetObjectField(env, clazz, instanceFieldId);
str = (*env)->GetStringUTFChars(env, jstr, 0);
(*env)->ReleaseStringUTFChars(env, jstr, str);

 jstr = (*env)->NewStringUTF(env, "Hello World!");
  (*env)->SetObjectField(env, clazz, fid, jstr);
```

Listing 6.45: Getting and setting a Java instance field value

Java methods

Like Java fields, Java methods can be handled by the JNI. Two types of methods are distinguished: instance methods and static methods. As example, considering that a certain Java class presents the following methods:

```
private String instanceMethod() {
        return "Instance Method";
 }

 private static String staticMethod() {
        return "Static Method";
}
```

Listing 6.46: Declaring Java methods

The related functions to access these methods are the following:

```
jmethodID instanceMethodId;
instanceMethodId = (*env)->GetMethodID(env, clazz, "instanceMethod", "()Ljava/
    lang/String;");

jmethodID staticMethodId;
staticMethodId = (*env)->GetStaticMethodID(env, clazz, "staticMethod", "()Ljava/
    lang/String;");
```

Listing 6.47: Acessing Java methods

Note the last argument of both functions, that, in the example, took the value `"()Ljava/lang/String;"`. This defines the method descriptor which represents the method signature in Java. The Table 4 presents Java types and the corresponding signature.

The JDK provides an useful tool named Java Class File Disassembler, available as a command line called *javap*. This tool can be used to extract the method signature from the compiled class files:

```
# javap -classpath bin/classes -p -s com.android.droidguardian
```

Listing 6.48: Example of use of the javap tool

The source code and tables present in this section was inspired by [26].

6.5 ANDROID DEVELOPMENT

The Android OS was designed to achieve the best performance when running in mobile devices. This goal brings particular characteristics that developers need to bear in mind when creating Android apps. Some of these characteristics are introduced in this section.

Java Type	Signature
Boolean	Z
Byte	B
Char	C
Short	S
Int	I
Long	J
Float	F
Double	D
type[]	[type
fully-qualified-class	Lfully-qualified-class;
method type	(arg-type)ret-type

Table 4: Java types and signatures

6.5.1 *Application Not Responding*

Android is built to be highly responsive, which means components may not block on I/O operations on the main thread (also known as the User Interface (UI) thread). The system must always be available to respond to user input events. Whenever an application is using the main thread, Android triggers an internal clock that fires when one of the following conditions occurred:

- No response to an input event within 5 seconds.
- A BroadcastReceiver hasn't finished executed within 10 seconds.

Then, the system launches an Application Not Responding (ANR) dialog window allowing the user to stop the application's execution, as illustrated in Figure 14.

To avoid ANR dialogs, it is recommended to use worker threads in order to execute heavy and long tasks that would block the main thread. The following section introduces the use of worker threads on Android.

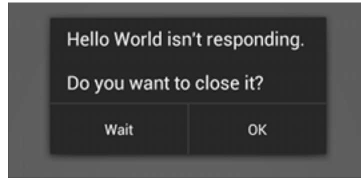

Figure 14: Application Not Responding dialog window

6.5.2 *Worker Threads*

Android consists of four main components that were presented earlier: Activities, Services, Broadcast Receivers and Content Providers. When an application starts executing, Android initiates a new process and, usually, all components run in the same process and thread (main thread). This thread is also called the UI thread, because it supports the Android UI toolkit components (from the `android.widget` and `android.view` packages) that applications interact with. When users provide UI events, such as touching a button on the screen, the event is enqueued until the UI thread is ready to dispatch it. If some application's component starts a heavy task, by default, it will run on the UI thread which may lead to a blocking situation where user input events get stuck on the queue [27]. This is the typical case where ANR dialogs are prompted.

Worker threads are used to overcome this problem. In order to take off heavy tasks from the UI thread so that it can be ready to handle UI events, developers must create other threads to execute such tasks. Note that UI events cannot be handled by worker threads, which means that developers may not access the Android UI toolkit outside the main thread.

6.6 SUMMARY

This chapter presented valuable descriptions regarding different matters. Each topic was very useful in the development process of DroidGuardian. The next chapter presents the technical approach to the implementation phase.

7

IMPLEMENTING DROIDGUARDIAN

The previous chapters introduced fundamental concepts regarding the development of DroidGuardian. This chapter presents their practical application on the implementation process. At first, it is given a detailed description of each component preceded by the general architecture. Follows a topic related to the data structures used to exchange data between layers. Then, it is highlighted some relevant implementation decisions. The last section introduces the environment in which DroidGuardian was built concerning operating systems and tools.

7.1 CONCEPTION

In Section 1.2, it is described the main goals that DroidGuardian should achieve. The starting point was to handle the LSM framework in order to intercept socket connections at kernel level. As described in Chapter 5, the LSM framework allows to use callback functions placed in strategic points in the kernel code to provide a fine-grained access to kernel objects. Since sockets are considered kernel objects and the LSM framework provides a set of callback functions to handle them, the solution to intercept outgoing Internet connection requests would imply these hook functions. It is important to stress the fact that the implementation of a custom LSM did not follow a different approach due to the circumstance of building a mechanism to the Android system. The Linux kernel that acts as the foundation of the Android system includes the same LSM framework that the mainstream Linux kernel has. Moreover, the custom LSM built to DroidGuardian would run perfectly in any Linux kernel, within the same version.

At this point, there was a module running inside the kernel that was intercepting every socket connect function call. It was necessary to provide a mechanism able to communicate with the kernel module in order to receive the socket connection request data and to send the permission that would either allow or deny the socket connection. Furthermore, this mechanism should reach out the end user so that this could provide his intention regarding the pending network request. Considering the Android system described in Chapter 2, to interact with the user through a GUI it is imperative to build an Android application. The Android application framework is designed to deal with Java programs, in contrast with kernel modules that are written in C. Fortunately, Android allows its applications to use the JNI where the potential of native code may bring considerable enhancements, as mentioned in

section 6.4. In this case, it was very important to have a program running as an Android application able to communicate with the kernel module in the same language. Therefore, DroidGuardian uses a native library to establish a communication between the kernel module and the Java application.

The GUI is implemented in the same application that calls the native library. It uses the main Android components to run in background indefinitely, and to display an alert window when it is required a response from the user.

The following section presents the general architecture of DroidGuardian, introducing the three layers that enables the communication between the kernel module and the end user. Further, it is discussed each layer in detail.

7.2 DROIDGUARDIAN ARCHITECTURE

DroidGuardian consists of three layers presented in Figure 15. At the bottom lays the *Kernel Module* that uses the LSM framework to place hook functions in the Linux kernel. The remaining layers comprise the Android Package (APK): the *Native Layer* and the *Java Layer*. DroidGuardian uses a native library to communicate with the kernel module and to process some data before it arrives to the Java layer.

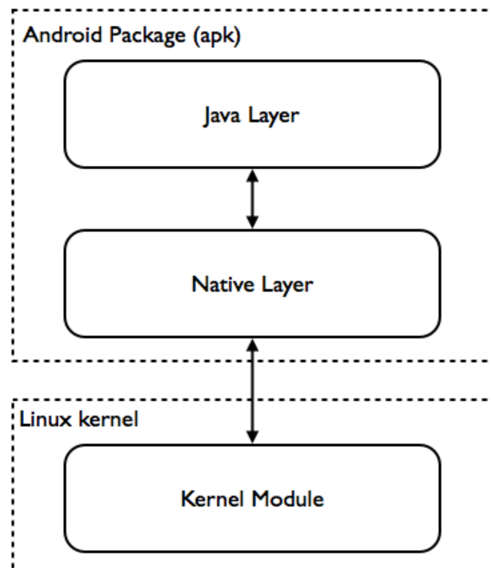

Figure 15: DroidGuardian architecture

The kernel module uses local domain stream sockets to communicate with the native layer. This local socket is stored in the `/data/data/` directory of the Android filesystem. The native layer also uses local domain stream sockets to communicate with the Java layer, due to the requirement of exchange data synchronously. This communication uses abstract sockets, mentioned in Section 6.2, thus there is no physical representation of the socket file in the filesystem. In Section 7.6 it will be

described the DroidGuardian Protocol that presents the data structures used to exchanged data between layers.

7.3 KERNEL MODULE

It took a long time-consuming process to analyze and understand the LSM framework, because there is no official documentation and hook functions are constantly being changed. Even though, it was possible to successfully deploy a custom LSM that fulfills the provided goals.

Basically, this module uses default hook functions to every set of operations, except for those that involve sockets. Recalling Chapter 5, the LSM framework provides default functions in case any module is loaded, or in case the loaded module don't implement all hook functions. DroidGuardian is included in this last case, where just socket related hook functions have their own custom implementation.

It was created a C file that declares the `security_operations` structure in which all socket operations are pointed to their custom functions, having the prefix '`droidg_`'. Listing 7.1 presents a code snippet of the structure.

```
static struct security_operations droidg_ops = {
  .name =       "droidg",
#ifdef CONFIG_SECURITY_NETWORK
  .unix_stream_connect =  droidg_unix_stream_connect,
  .unix_may_send =    droidg_unix_may_send,
  .socket_create =    droidg_socket_create,
  .socket_post_create = droidg_socket_post_create,
  .socket_bind =      droidg_socket_bind,
  .socket_connect =   droidg_socket_connect,
  .socket_listen =    droidg_socket_listen,
  .socket_accept =    droidg_socket_accept,
  .socket_sendmsg =   droidg_socket_sendmsg,
  .socket_recvmsg =   droidg_socket_recvmsg,
  (...)
#endif /* CONFIG_SECURITY_NETWORK */
};
```

Listing 7.1: Code snippet of the DroidGuardian's `security_operations` structure

However, the implementation of almost all socket functions is actually the same as their default implementation. They just return a default value and have no instructions. In order to intercept the outgoing traffic and to extract the desired data - IP address, port, process name and PID - only one function fulfills these requirements: `socket_connect()`. This function has the declaration presented in Listing 7.2.

```
static int droidg_socket_connect(struct socket *sock, struct sockaddr *address,
    int addrlen)
```

Listing 7.2: Declaration of the `droidg_socket_connect()` function

The following items explain how those four elements were extracted from the LSM.

- As described in Section 6.2, the `connect` primitive has as arguments the `socket` structure of the socket that intends to get connected, the `sockaddr` structure of the remote address and its length. Through the second argument, `address`, DroidGuardian is able to extract both the remote **IP address** and **port**.
- At kernel level, it is possible to use the `current` global variable that points to the `task_struct` structure provided by the `include/linux/sched.h` header file. This structure contains the data related to the running process. Using `current->pid` and `current->comm` it is possible to get the **PID** and **process name**, respectively.

Figure 16 presents the kernel module process flow.

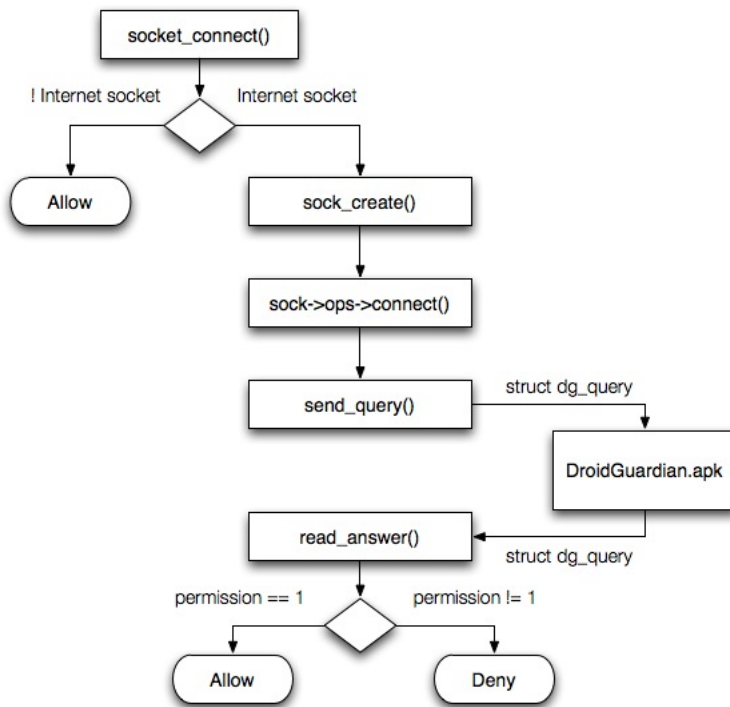

Figure 16: Kernel Module process flow

The process flow illustrated in Figure 16 is described as follows:

1. Whenever a process (or Android application) executes the `connect` system call, the `droidg_socket_connect` hook function is called;

2. The socket family type is analyzed, by accessing the element `address->family` in the `sockaddr` structure. If it is different from `AF_INET` or `AF_INET6`, the hook function immediately returns 0 allowing the connection, because DroidGuardian is only interested in Internet sockets.

3. Once passed the family type check, IP versions are distinguished: if the socket uses IPv4 (`AF_INET`), a `sockaddr_in` structure is filled in by casting the `sockaddr` structure; otherwise the socket uses IPv6 (`AF_INET6`) and a `sockaddr_in6` structure is filled in, in the same way. These structures will store the remote address data and they will be sent to the application.

4. A new Unix domain stream socket is created to communicate with the application, and the subsequent steps to establish a stream connection are executed. This socket plays the client role (it is only created and connected to the server socket).

5. The `sock_sendmsg()` function is called to send the data to the application.

6. The `sock_recvmsg()` function is called to get the value that will define the hook as permissive (Allow) or blocker (Deny).

If the output of the hook function is different from 0, the socket will not be connected. If the connection request was accepted by DroidGuardian, the hook function returns 0, otherwise returns `EPERM`, which is the macro to *"Operation not permitted"*. Note that the kernel module bypasses two system processes, called *netd* and *Captive Portal*, in order to get the network stack always functional. Blocking some system processes may lead to irreversible errors.

The `struct dg_query` used to exchange data between the kernel module and the native layer will be presented on Section 7.6.

7.4 NATIVE LAYER

Android provides an interface to take advantage of the powerful native libraries that was introduced in Section 6.4. DroidGuardian uses this library to establish the communication between the kernel module and the Java layer. The native code is executed as a library by the Java layer that calls the native method implemented in this library. Figure 17 presents the process flow of the native layer.

The process flow illustrated in Figure 17 is described as follows:

1. The main native function starts its execution;

2. It follows TCP protocol to establish a stream socket connection (creating, binding, listening and accepting). Recall that the kernel module produces client sockets that get connected to a server socket. The native layer implements this server socket;

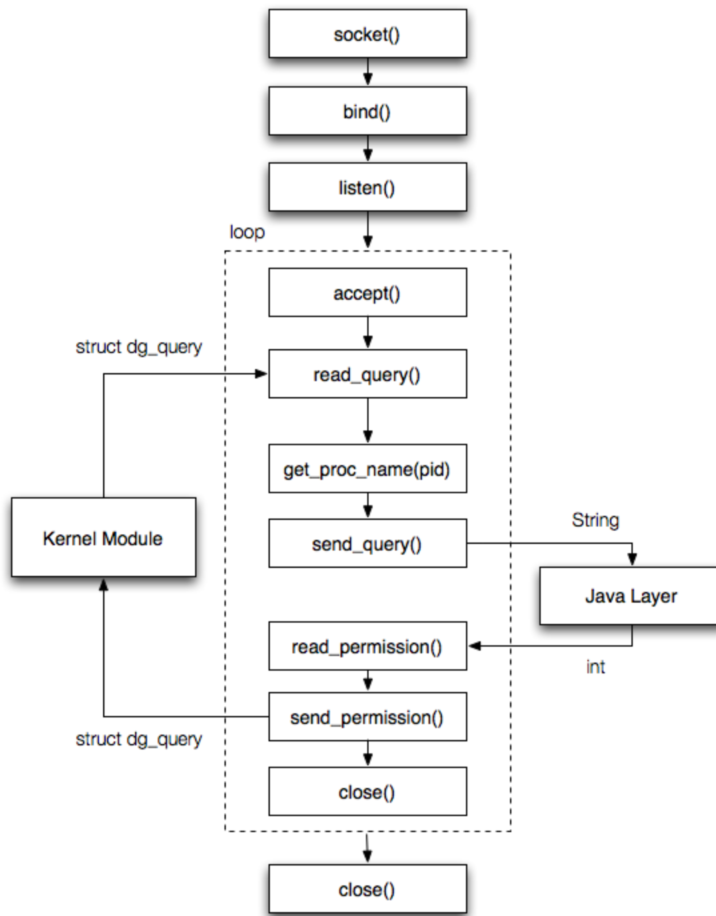

Figure 17: Process flow in the Native layer

3. Within an infinite loop, connection requests from the kernel module are attended, one by one;

4. When a request is accepted, the server socket receives the data sent by the client socket through the `read` primitive;

5. This data comes in the form of a C structure (`struct dg_query`) containing the remote address and PID generated in the kernel module;

6. The native layer uses the PID to read the `/proc/<pid>/cmdline` file in order to obtain the Android package name associated to the process that launched the connection request. Further in this chapter, it will be explained why the `current->comm` value did not fit the need to handle the process name;

7. There are two distinct functions that handle the address structures, depending on the IP version. If the socket family relates to IPv4, it is inspected the `sockaddr_in` structure in order to get the IP address and port. The IP address is translated into the dot format using the `inet_ntop()`

function available on the arpa/inet.h header file. The port value is obtained using the ntohs() function available on the netinet/in.h header file. If the socket family relates to IPv6, it is inspected the sockaddr_in6 structure in order to get the IP address and port. Both values are translated into the corresponding formats using the same functions;

8. At this point, all data regarding the connection request is in the right format to be sent to the Java layer. However, the communication between the native layer and the Java layer concerns two different languages. To ensure that the data sent by the native layer is correctly read by the Java layer, the four elements - IP address, port, process name and PID - are wrapped up in a single string;

9. The native layer creates a new socket to communicate to the Java layer. This initiates an Unix domain server socket that waits for connection requests from the native layer.

10. After sending its message, the native layer receives the answer from the Java layer;

11. According to the permission value received, the query structure is updated and sent back to the kernel module.

This process is repeated whenever a new connection request arrives from the kernel module. Note that after the binding operation, the native layer changes the socket file permissions using chmod so that DroidGuardian has the necessary read/write permissions, as presented in Listing 7.3.

```
#define SOCK_PATH "/data/data/com.rmgoncalo.droidg/dg_daemon_server"

chmod(SOCK_PATH, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

Listing 7.3: Changing socket file permissions

7.5 JAVA LAYER

The topmost layer of DroidGuardian has the main purpose of displaying the query data to the end user so that he can decide whether to accept or reject the connection. The Android application comprises two sets of classes. One set is constituted by the classes that extend Android components, presented as follows:

- BootReceiver: extends a BroadcastReceiver that starts DroidGuardian after the device booting process;
- Daemon: extends a Service that runs in background indefinitely;
- QueryActivity: extends an ActivityFragment that allows to display the dialog window;

- `DialogWindow`: extends a `DialogFragment` to exhibit the dialog window.

These classes comprise about 70% of the total amount of Java code. The remaining classes are used to define the following Java objects:

- `Query`: declares the four elements exchange in queries as instance variables - IP address, port, process name and PID;
- `Rule`: defines the rule object composed of those four elements along with the permission and lifetime values;
- `RulesList`: implements a `TreeMap` to store the rules that are being created while DroidGuardian is running;
- `Protocol`: defines some macros used by several classes.

The process flow of the Java layer is illustrated in Figure 18.

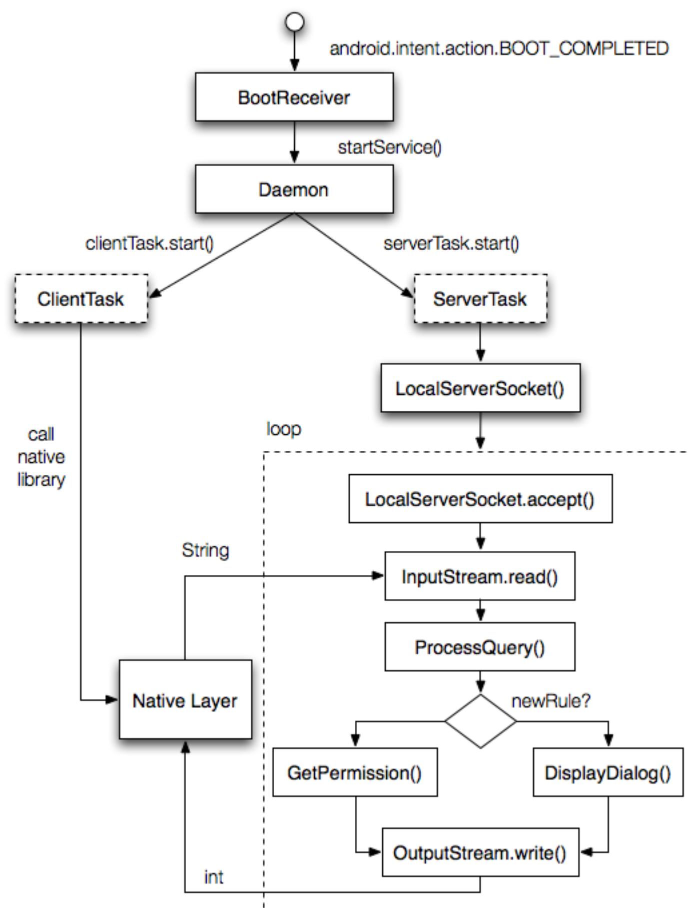

Figure 18: Process flow in the Java layer

DroidGuardian was design to run as a daemon, silently and unnoticed, until new outgoing Internet connection requests fall into the kernel module. The following steps describe the process flow of the Java layer:

1. Android sends the `BOOT_COMPLETED` intent action after the booting process and `BootReceiver` grabs it;

2. `BootReceiver` starts the `Daemon` through the `startService()` method;

3. The `Daemon` initiates two threads: `ServerTask` and `ClientTask`. The former operates as a server in the stream socket protocol and will run until an external perturbation, such as low memory, destroys the service. If nothing happens, this socket server will run while the device is on. Android provides an interface to implement Unix domain server sockets, `LocalServerSocket`, and Unix domain client sockets, `LocalSocket`. The later acts as the client socket in this connection, by calling the native library described above. This thread calls the native method `startDaemon()` that kicks off the native engine;

4. Once executing, the server socket starts an infinite loop and accepts the pending requests, one by one;

5. The server receives the query string through the `read()` method of the `InputStream` interface;

6. This string feeds a new instance of the `Query` class that will parse it using `split` and assign its variables - `address`, `port`, `pid` and `processName` - the corresponding values;

7. The `processQuery` method checks if it is being displayed a dialog message regarding the same process that just arrived to the Java layer. If the process is the same, i.e. has the same name, it waits until a response is provided by the user indicating the permission to that process to access the Internet. When this happens, it is inspected the `RulesList` to get the permission value regarding this process. Once obtained this value, it is sent to the native layer. If the process is not being prompted to the user, neither is on the `RulesList`, that means it is the first time this process tries to access the Internet;

8. In this case, the `QueryActivity` is launched. Despite it is an activity component that should be used to provide a visual interface, it is invisible to the user and it's purposes is just to launch the dialog fragment. Due to the need of a synchronous communication between the dialog and the daemon, it is used Unix domain stream sockets to establish this transmission.

9. The `DialogWindow` is displayed, presenting the query information through the following message: *"Process `processName` (`pid`) wants to connect to address `address` on port `port`."*. Along with this message, two buttons are available, *"Allow"* and *"Deny"*, so that the

user may easily provide his decision. Also, a spinner element is presented to assign the time tag to the rule, being *"Forever"* or *"Once"*.

10. Depending on the user's decision, a new rule may be created. In any case, the permission value is always stored and sent to the native layer.

The following section describes the protocols used to exchange data between layers, and how the permission and lifetime values are handled.

7.6 DROIDGUARDIAN PROTOCOL

The three layers use different protocols to exchange data with each other. The following subsections describe the data structures used by each layer to send and receive data.

7.6.1 *Exchanging data between the Kernel module and the Native layer*

Both the kernel module and the native layer declare the same C structure, named `dg_query`, as follows:

```
#define DG_INET 1
#define DG_INET6 2

#define DENY 0
#define ALLOW 1

#define ONCE 0
#define FOREVER 1

struct dg_query {
  int family;
  struct sockaddr_in addrin;
  struct sockaddr_in addrin6;
  int pid;
  int permission;
  int lifetime;
}
```

Listing 7.4: Declaration of the `dg_query` structure

The `dg_query` structure's elements are described as follows:

- `family` defines the IP version: it is used the macro `DG_INET` to characterize IPv4, and the macro `DG_INET6` to characterize IPv6;
- `addrin` is used to store the remote address structure when IPv4 is used;

- `addrin6` is used to store the remote address structure when IPv6 is used;
- `pid` gives the process identifier;
- `permission` indicates the query's permission: it is used the macro `ALLOW` to grant permission, and the macro `DENY` to deny permission;
- `lifetime` assigns a time tag of the rule: it used the macro `ONCE` to state that the query's permission was assigned only once in time, and the macro `FOREVER` to state that the query's permission was assigned forever in time.

The actual implementation of the kernel module does not check the lifetime value. It just receives the permission value in order to allow or deny the connection.

7.6.2 *Exchanging data between the Native layer and the Java layer*

These layers use different structures to exchange data. As mentioned in Section 7.4, the native layer creates a single string to store each set of data derived from the kernel module's queries. This string is sent through Unix domain sockets, and the Java layer uses the `InputStream` interface to read the string. This string is then parsed so that the Java layer can extract and process each element of the query.

The data sent by the Java layer to the native layer follows a different approach. At first, it is important to understand the possible outcome from the user interaction with the dialog interface. Figure 19 presents all possible paths regarding the *Spinner* and *Button* input events.

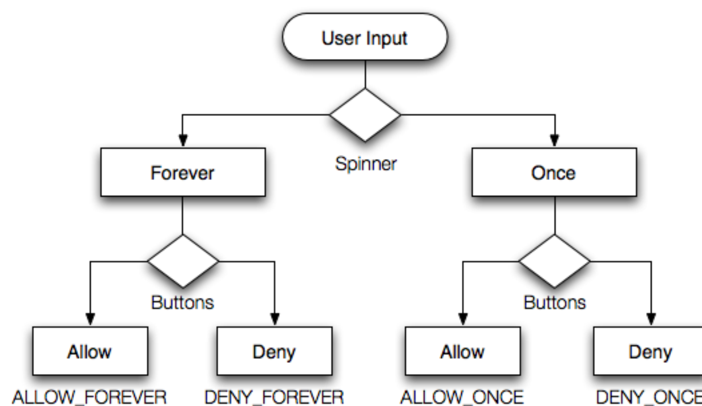

Figure 19: Dialog results diagram

The macros illustrated in Figure 19 are defined with the following values:

```
#define ALLOW_FOREVER 1
#define DENY_FOREVER 2
#define ALLOW_ONCE 3
```

```
#define DENY_ONCE 4
```

Listing 7.5: Declaration of macros to the dialog results

For each macro, the values assigned to the variables `permission` and `lifetime` are the following:

- ALLOW_FOREVER: `permission = 1` $\wedge$ `lifetime = 1`
- DENY_FOREVER: `permission = 0` $\wedge$ `lifetime = 1`
- ALLOW_ONCE: `permission = 1` $\wedge$ `lifetime = 0`
- DENY_ONCE: `permission = 0` $\wedge$ `lifetime = 0`

Using this method, the Java layer sends an `int` value to the native layer, with range [1-4] specifying both the permission and lifetime condition of the query.

7.7 DISCUSSION

While developing DroidGuardian, various doubts and questions came out regarding the best way to implement certain features. This section presents those cases along with the decision's discussion.

7.7.1 *Process Name*

At first, the kernel module was storing the value provided by `current->comm` in the `dg_query` structure in order to use it as the process name on the dialog message. However, the names received were not clear to the end user. DroidGuardian was printing names like *"Async Task #2"*, *"Thread-73"*, *"WebViewCoreThre"*, etc. But, it is normal that the names stored in `current->comm` do not necessarily point to Android applications. After all, one application may run several processes.

In order to provide friendly names that would explicitly indicate the user what application was trying to access the Internet, it was discarded the process name used before, and it was implemented an auxiliary function on the native library to read the `/proc/<pid>/cmdline` file, using the process identifier value supplied by `current->pid`. This file gives the full package name of each application. For instance, the process name *"WebViewCoreThre"* used by the browser application was replaced by *"com.android.browser"*.

7.7.2 *Dialog vs Notification*

The dialog window don't follow the correct rules that Android states when it comes to alert the user that some event occurred. Dialogs exist for this purpose, but in a different context. A dialog alert

should be used inside an activity that the user intentionally invoked. For instance, when the user triggers an action to delete data from a certain folder it is expected that a dialog window pops up asking if he intends to delete that data. This is a consequence of the user's action.

In situations where an event occurs outside the context of the application that the user is interacting with, Android offers the *Notification* interface. Notifications are messages displayed on the notification bar, placed at the top (or bottom) of Android devices, by icons. When a new icon appears on the notification bar, it means that some event took place as a result from a background action. The user is able to expand the notification bar to check all notifications that, usually, comprise some short information text. By clicking on the notification area, it may fill the screen with data related to the event that occurred, depending on how the notification was developed. Users are free to keep notifications unread for as long as they want, without lose performance.

Considering both elements, dialogs and notification, the DroidGuardian case fits better in the last, because the event that triggers an alert to the user happens in background. However, taking the Internet connection request to the notification bar would lead to a longest response time when compared to the dialog. The time the user takes to provide his input is included in the total amount of time that the socket waits in the kernel in order to accept or reject the connection. It is known that kernel operations should be executed as fast as possible and that keeping the kernel stuck could bring several damages to the system. Even though it is kept waiting a considerable amount of time using dialogs, compared to notifications this time would increase.

It was decided that disrupt the user from whatever he is doing, with an alert pop up was better than keeping the kernel waiting long periods of time.

7.7.3 *Service and Dialog Communication*

Android provides methods to allow components to exchange data. Actually, there are simple ways to launch components and pass values within the same method. For instance, a service can start an activity calling `startActivity` and passing an intent object as argument. This intent may store several values using the `putExta` method. When the activity is running, it is able to inspect the intent through the `getIntent` method and to get its values using the `get_type_Extra` method, where *type* defines the data types.

However, when a service and an activity are both running, it is not simple to exchange data. Android provides the Android Interface Definition Language (AIDL) to solve these cases, where components are allowed to exchange data. This method would provide the `Dameon` service and the `DialogWindow` activity a mean to exchange data. But, DroidGuardian presents a different scenario. When the kernel module sends a query to the layers above, it blocks until a response is received. This forces the native and Java layers to communicate synchronously in order to attend the kernel module's request and provide an answer. Android dialogs are not designed to behave synchronously. The sys-

tem cannot block until the user provides some input event. For this reason DroidGuardian was forced to implement socket communication, to apply a synchronous system of exchanging messages.

7.7.4 *Starting DroidGuardian on Boot*

DroidGuardian is designed to start running as soon as possible in order to dispatch the kernel requests. Android allow applications to start executing after the booting process by requesting a Manifest permission defined as `android.permission.RECEIVE_BOOT_COMPLETED`, and by grabbing the intent, which action is `ACTION_BOOT_COMPLETED`.

Using a receiver it is possible to grab that intent and execute a certain operation. DroidGuardian implements a BroadcastReceiver class which goal is to start the `Daemon` service:

```
public class BootReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
            Intent i = new Intent(context, Daemon.class);
            context.startService(i);
        }
    }
}
```

Listing 7.6: BroadcastReceiver responsible to start the service after the booting process

This receiver must be declared on the Manifest file by defining the `BOOT_COMPLETED` intent's action that falls under the `DEFAULT` category:

```
 <receiver
    android:name="com.rmgoncalo.droidg.BootReceiver"
    android:exported="true"
    android:enabled="true"
    android:permission="android.permission.RECEIVE_BOOT_COMPLETED">>

    <intent-filter >
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</receiver>
```

Listing 7.7: Declaring the receiver component on the Manifest file

7.8 SETTING UP THE ENVIRONMENT

Building DroidGuardian comprised several stages that are directly related to the layer that was being handled. For instance, the kernel module layer requested a completely different implementation environment when compared to the Java layer.

In order to fully understand the LSM framework it was necessary to manipulate a real Linux kernel, as well as compile it and install it. Since the development computer used was a *MacBook Pro*, which runs the *OS X* operating system, a new disk partition was created to install the *Xubuntu* operating system. It is a different flavor of the *Ubuntu Linux* operating system that provides a light user interface. Since the only needed program was the console, because all required steps could be executed through the command line and using *Vim*, a lighter user interface was good enough. The new disk partition was created using the *rEFIt*[1] tool.

Running Linux on a new partition provided speed and efficiency when setting up the Android environment in order to build and launch a new image on the emulator. However, handling loadable kernel modules on a separate partition proved to be a mistake, due to the system's blocking when kernel failures were reached by programming errors. To overcome this inconvenience, programming tests with loadable kernel modules started to be done in a virtual machine. This way, if the code contained flaws that could led to a kernel panic, the virtual machine could easily be restarted causing no harm to the host operating system. *VMWare* was used to virtualize a *Xubuntu* operating system, being *OS X* the host operating system.

Regarding the Android applications development environment, *Eclipse* was chosen as the Integrated Development Environment (IDE), because is widely used, well documented and almost all issues an user may face are solved in internet forums, books and other sources.

Application testing was conducted on both the Android emulator and a real device. The device was a *Commtiva z71* running Android 2.3.3, API level 10.

7.9 SUMMARY

This chapter provided a thorough description of the implementation of DroidGuardian. Starts with the theoretical conception that led to the practical software development. Each layer is fully covered, being explained its process flow and the relevant operations it executes. It also presents a section of discussion where the most important topics regarding the implementation are provided. The next chapter exhibits a show case.

1 http://refit.sourceforge.net

8

USING DROIDGUARDIAN

This chapter presents a show case of DroidGuardian, providing some screenshots that allow to visualize how data is displayed in both the kernel module and the Android app.

8.1 THE BROWSER SHOW CASE

DroidGuardian starts running automatically after the device's booting process. The kernel module prints messages on the `/var/log/messages` file using the `printk` function. These messages indicate whether a query was not successfully sent to the DroidGuardian app, and the related error. In case the communication was established without problems, the kernel module prints the permission assigned to the corresponding query.

When it is clicked the browser icon, the browser starts running and tries to connect to the Internet. DroidGuardian exhibits the dialog window presented on Figure 20.

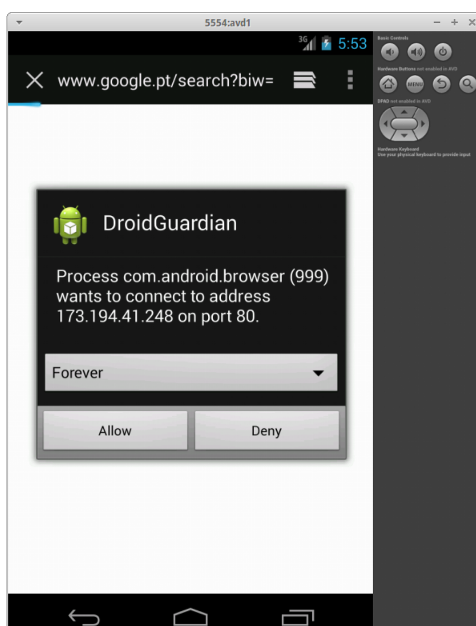

Figure 20: DroidGuardian window dialog

According to the Figure 20, the browser, which package name is *com.android.browser* and PID *999*, was trying to access the remote address *173.194.41.248* on port *80*. Using a simple lookup IP address tool, we find that this IP points to the hostname *mad01s15-in-f24.1e100.net* belonging to the organization *Google*.

8.2 BENCHMARKING DROIDGUARDIAN

This section provides the results obtained from a simple benchmarking procedure in order to assess the performance of DroidGuardian. Since this tool runs its own code inside Android's kernel by intercepting all socket's `connect()` primitives invocations and executing the business logic responsible for sending the socket's data to the native module, it is important to address what overhead is associated to the changing of socket connection callbacks, taking into account that they might be invoked several times per second in Android devices. Therefore, it is expected that a custom kernel running DroidGuardian's kernel module takes a bit longer to establish a socket connection than the original Android kernel. However, the act of exchange data between sockets shouldn't register any additional overhead, because DroidGuardian's kernel module doesn't affect the `read()` and `write()` primitives invocations.

In order to assess the execution load associated to the kernel module, it was run DroidGuardian without the native and java components. Instead of sending all internet sockets's data to the native module and wait for a response, it was forced an internal reply so that the kernel module could run independently. This response was defined as permissive, which means that all socket connections were accepted.

The results achieved are shown in Table 5. It is concluded that DroidGuardian takes, in fact, a bit longer to establish a socket connection, but the overhead is insignificant. Therefore, the kernel module doesn't affect the performance of the entire system.

The native module is responsible for receiving a message sent by the kernel module, extracting some data from this message, forwarding this data to the java layer and waiting for a response so that it may reply back to the kernel module. It is possible to define the native module as a router. In order to estimate what overhead this router causes, it was modified to run independently from the java layer and to communicate only to the kernel module. Like the previous procedure, the response was permissive. Thus, the kernel module intercepted a socket connection, sent its data to the native module that answered back the permissive response. The kernel module received the response and executed the instruction according to it. Once more, the overhead should be noticed in socket connections and not in data exchanging.

By the analysis of Table 5, it is clear that this second procedure extends the `connect()` execution time a bit longer, which is acceptable, because there are two modules being executed, one of which that runs outside the kernel space. But the additional overhead is considerably low and doesn't bring negative effects to the global performance.

At last, the java layer uses the Android SDK to run as a mobile application so that the user can interact with DroidGuardian. The time the user takes to provide an input to the dialog is not measured, because this doesn't depend on the system. However, this input is simulated as a permissive response, once more, by replying automatically to every request that comes from the native module. Through this procedure it is possible to estimate the overhead that DroidGuardian applies on the system, by measuring all modules working together. It is expected that a socket connection invocation takes longer than the two previous procedures to complete, because this one runs all three layers: the kernel module, the native module and the java layer. Thus, when a socket connection is intercepted on the kernel module, a message goes across all DroidGuardian's modules reaching the start point in the end, when the kernel module is able to allow or deny the connection request. The time this process takes is an important benchmark in order to assess the global performance of DroidGuardian. In case of data exchanging between sockets, once again, it is not expected an additional overhead.

The last column in Table 5 shows that the performance of DroidGuardian as an application that runs during the entire lifetime of the device doesn't bring heavier overhead and that it may be used as an useful protection mechanism.

Primitives	Original kernel	Kernel module	Kernel and Native modules	Three layers
Connect	0.00134	0.00198	0.00205	0.00212
Read and Write (10b)	0.01469	0.01601	0.01742	0.01636
Read and Write (5Kb)	0.01723	0.01722	0.01711	0.01737

Table 5: Benchmarking DroidGuardian

In order to achieve these results, it was used an Android emulator with the following properties:

- Kernel: Android goldfish 2.6.29
- CPU/ABI: ARM (armeabi-v7a)
- Target: Android 4.3 (API Level 18)

The emulator ran on a MacBook Pro with the following properties:

- 2.3GHz Intel Core i5
- 8 GB 1333 MHz DDR3,
- Operating System: Xubuntu 14.04 64

It is important to address that if the procedures were conducted on a real device, the execution time of each primitive would be lower, due to a higher processing speed. Unfortunately, this was not possible in the defined schedule, because running a custom kernel on a real device envolves a

set of time consuming tasks, as rooting the device, find an appropriate ROM that fits that specific device, compile this ROM with the custom kernel and successfully deploy it and having it run without problems. However, the results obtained using an Android emulator provide a reliable source of comparison between an original kernel and DroidGuardian's custom kernel.

8.3 SUMMARY

This chapter presented a very simple practical case of the use of DroidGuardian, illustrating a screenshot of the dialog message. Also, it introduced some details regarding a simple benchmark that assessed the performance of DroidGuardian. The next chapter gives an overview of the project and the conclusions about its achievements, along with the future work.

9

CONCLUSION AND FUTURE WORK

This document comprises a project in which it was designed a new mechanism to the Android OS able to intercept outgoing Internet connection requests providing end users the ability to either accept or reject such requests in real time. It took a long time-consuming process to conceive this technology, due to several factors:

- The network stack of the Linux kernel is not a simple system. It requires high levels of expertise to handle network components in order to filter or extract related data;
- The LSM framework is very complex, because it handles a large set of kernel objects and comprises more than a hundred callback functions. The main disadvantage is the lacking of official documentation to clarify how to build custom LSM modules;
- The implementation of sockets at kernel level brings some peculiarities that may turn it into a complicated process;
- Also, the communication through Unix domain sockets when it involves kernel space and user space requires solid knowledge;
- A reliable implementation of a native library using the JNI demands the compliance of certain rules what makes it a hard task;
- DroidGuardian runs out of the standard of regular Android applications. At first, it presents a daemon that is supposed to run indefinitely. Usually, Android services are used to carry out operations that may take a while, but eventually will end, for instance, Internet downloads, music playing, etc. Second, it uses alert dialogs that are launched out of the context of the application the user may be interacting with, what disrupts a possible task he might be carrying out. At last, Android is designed to respond asynchronously to a large set of events, including dialog messages. However, DroidGuardian needs a synchronous response from the dialog in order to send it to the kernel module before the following request is attended.

Nonetheless, DroidGuardian succeeded to achieve the outlined goals.

At first, it was purposed the development of a mechanism able to intercept all outgoing Internet connection requests from any process running in the system. In order to handle network outgoing

traffic it was used the LSM framework that applies a fine-grained access policy regarding kernel objects, such as sockets, through callback functions that may be implemented and inserted into the kernel at compile time. Using an IPC mechanism to establish a communication between kernel space and user space it is possible to exchange data within these hook functions. DroidGuardian's kernel module uses Unix domain stream sockets to provide synchronous communication to user space.

Along with the socket interception system, it was intended to extract some data regarding each connection request, namely the IP address and port of the remote server to which the process wanted to get connected, as well as its name and identifier. DroidGuardian implements the `socket_connect` hook function that is executed every time a process calls the `connect` system call. In Linux, every process that intends to get connected to the Internet is forced to use this system call. This requires the `sockaddr` structure of the desired remote server as argument. By inspecting its elements, it is possible to extract the IP address and port. Since these operations are being executed in kernel space, it is available an useful structure, called `task_struct`, that among other properties, provides the identifier of the running process. Thereby, the desired data is extracted and sent to user space, fulfilling this requirement.

Another feature the technology should have would be to display the aforementioned data related to connection requests to the end user. Taking into account that the mechanism were developed to the Android OS, the effective way to exhibit data would be through the user interface the Android SDK provides. Therefore, it was implemented an Android application that receives the connection requests's data and prints it to the screen using Android components designed to that purpose.

Furthermore, the user should also be prompted an interface whereby he could provide his decision regarding the request: to either allow or deny it. At this point, DroidGuardian were able to display the requests's data through the app. It was a question of providing an input object so that users could interact with to pass their decision. Therefore, it was implemented a dialog system that displays a message and two buttons. Each one of these buttons has a `OnClickListener` interface implemented that executes a certain operation when the button is clicked. Using a simple method, the user is able to click *"Allow"* in case of accepting the request, or *"Deny"* in case of rejection. DroidGuardian translates this events into values that are passed to the kernel module, which is waiting the response since it first intercepted the connection request.

At last, it was required a practical way to use this technology by means of a rule-based model able to filter connection requests. Since each application may launch several socket connect requests at a time, there must be a filter to handle this case that could easily become overwhelming to the app. DroidGuardian provides a filter placed at the Java layer, that controls the incoming requests from the native layer. To each request, it is assigned a time tag that the user provides through the dialog input objects. Along with the text message and buttons, also a spinner containing the values *"Forever"* and *"Once"* is shown. Thus, the user provides two values: a permission value through the buttons and a lifetime value through the spinner. By assigning a request as *forever*, DroidGuardian creates a new rule stating that to every request that has the same process name, its permission is automatically obtained

as being the same that the user provided by clicking the button. In case a request is assigned as *once*, no rule is created and every time the same process launches a request it is received by DroidGuardian as if it was the first time.

This way, all goals have been achieved and DroidGuardian proves to be an useful mechanism that helps to raise awareness regarding the risks related to the network traffic in Android devices.

9.1 FUTURE WORK

DroidGuardian is presented as a proof-of-concept to state that it is useful to have a mechanism able to filter Internet connections in order to inform users regarding what is going on under the hood. However, it is only the first step towards a powerful mechanism that can really help users. This section describes a set of improvements that would bring many advantages to Android consumers when using DroidGuardian.

This first version of an application firewall for Android devices intercepts relevant data concerning remote addresses to which processes intend to get connected, namely the IP addresses and ports. In order to provide users detailed information, DroidGuardian should execute a lookup system over each address. This way it would be able to present the hostname, organization, country, etc, of the address, which would help the decision of accepting or rejecting the connection.

An interface to manipulate saved rules becomes mandatory so that users may check their previous decisions and be aware of what happened regarding outgoing traffic. This interface should permit to edit and delete rules.

The deployment of DroidGuardian requires the use of a custom kernel, because it uses its own LSM that do not come in Android kernel releases. This requirement turns DroidGuardian really hard to use due to the complexity of installing a custom kernel. New ways of intercepting socket connections that do not imply installing kernel modules at compile time would bring a huge advantage. This solution should include the use of netfilter hooks or acting over the *libc* library, as mentioned on Chapter 4.

The use of netfilter hooks may also be useful through the *iptables* interface. For instance, when DroidGuardian sets a rule's lifetime as *forever* it may be useful to add an iptables rule using the corresponding PID to allow or deny the access to the Internet, according to the permission's value. This way, netfilter hooks would filter that connection, instead of DroidGuardian hooks.

BIBLIOGRAPHY

[1] Gartner, Inc., "Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013." Press Release, February 2014. http://www.gartner.com/newsroom/id/2665715.

[2] Slashdot, "Rovio denies knowledge of nsa access, angry birds website defaced anyway," 2014. http://goo.gl/CV8ybP.

[3] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., 2013.

[4] A. Gargenta, "Deep dive into android ipc/binder framework," Android Builders Summit, 2013.

[5] J. Stultz, "Waking systems from suspend." https://lwn.net/Articles/429925, 2011.

[6] eLinux webpage, "Android logging system." http://elinux.org/Android_Logging_System, 2012.

[7] A. Dubey and A. Misra, *Android Security: Attack and Defense*. CRC Press, 2013.

[8] Wikipedia, "Dalvik (software)," 2014. http://en.wikipedia.org/wiki/Dalvik_(software), last modified on 19 February 2014.

[9] D. Gollmann, *Computer Security*. Wiley, 2011.

[10] Marko Gargenta, "Android security underpinnings." https://thenewcircle.com/s/post/1518/Android_Security_Underpinnings.htm, 2013.

[11] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," tech. rep., Trusted Systems Research - National Security Agency, 2013.

[12] J. Six, *Application Security for the Android Platform*. O'Reilly Media, Inc., 2011.

[13] A. Ludwig, E. Davis, and J. Larimer, "Android: Practical security from the ground up," Presented at Virus Bulletin Conference 2013, 2013.

[14] Wikipedia, "Netfilter," 2013. http://en.wikipedia.org/wiki/Netfilter.

[15] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2012.

[16] Objective Development Software GmbH, *Little Snitch 3 Documentation*, 2013.

[17] B. C. da Silva and R. F. Weber, "Tuxguardian: Um firewall de host voltado para o usuário final," tech. rep., Instituto de Informática - Universidade Federal do Rio Grande do Sul, 2006.

[18] P. Loscocco and S. Smalley, "Meeting Critical Security Objectives with Security-Enhanced Linux," in *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[19] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," in *Proceedings of the* 21st *National Information Systems Security Conference*, pp. 303–314, October 1998.

[20] R. Farrow, "Linux 2.5 kernel developers summit." Conference Report, March 2001.

[21] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the* 11th *USENIX Security Symposium*, August 2002.

[22] Wikipedia, "Capability-based security," September 2013.

[23] B. Hall, "Beej's guide to network programming: Using internet sockets," tech. rep., 2012.

[24] R. W. Stevens and S. A. Rago, *Advanced Programming in the UNIX(R) Environment (3Nd Edition)*. Addison-Wesley Professional, 2013.

[25] S. Ratabouil, *Android NDK Beginner's Guide*. Packt Publishing, 2012.

[26] O. Cinar, *Pro C++ with the NDK*. Apress, 2012.

[27] S. Komatineni and D. MacLean, *Pro Android 4*. Apress, 2012.