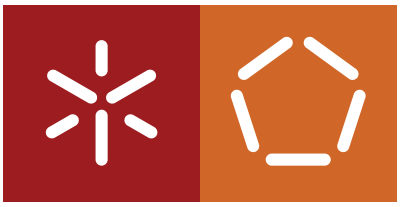

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Rui Miguel de Carvalho Videira Gonçalo

DroidGuardian: An Application Firewall for Android OS-based Devices

Master dissertation

Supervised by: Victor Francisco Fonte

Braga, January 31, 2014

ACKNOWLEDGEMENTS

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. On the other hand, Android OS-based devices could be used as a valuable on-site security auditing tool, but lack a set of coherent and fully functional applications, specially developed for this inherently resource-constrained platform.

The ultimate goal of this project is to start the development of a set of coherent and integrated tools that, ultimately, will enable Android OS-based devices to be used in network security auditing. These tools will include traffic filtering, network mapping, vulnerability assessment and intrusion detection. This project will also help identify and raise awareness to current network-based threats to Android OS-based devices.

RESUMO

Os dispositivos móveis que correm o sistema operativo Android são amplamente usados para navegar na internet e aceder um vasto leque de serviços online. No entanto, o facto de ser uma plataforma móvel usada à escala global, coloca-a como um alvo extremamente apetecido para ataques maliciosos que tentam identificar e explorar potenciais vulnerabilidades, a fim de aceder a dados privados. Por outro lado, os dispositivos Android podem ser usados como uma valiosa ferramenta de auditoria móvel.

O objetivo final deste projeto passa pelo desenvolvimento de um conjunto de ferramentas que possibilitem aos dispositivos que suportam Android avaliar redes de internet em termos de segurança. Estas ferramentas deverão incluir filtragem de tráfego, mapeação de redes, avaliação de vulnerabilidades e deteção de intrusões. Este projecto pretende também alertar os utilizadores para os perigos do uso de internet em dispositivos Android.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

Introduction

1.1 MOTIVATION

1.2 OBJECTIVES

1.3 STRUCTURE

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 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 , which is a lighter version of the 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.

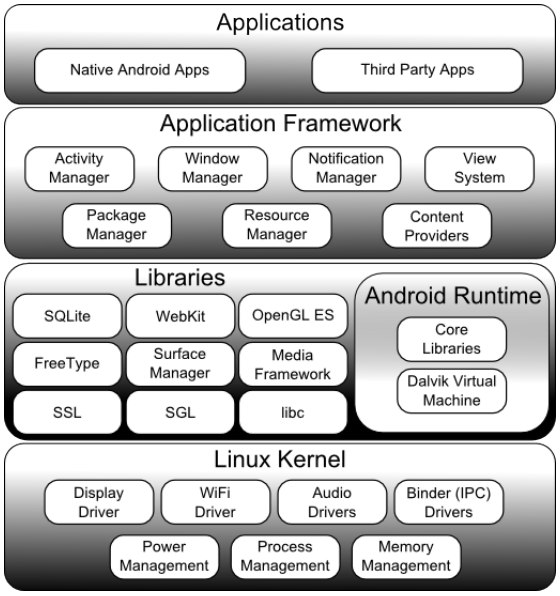

Figure 1.: Android architecture

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.: Linux kernel versions and Android realeases

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 properly modified in order to achieve exceptional results in an embedded environment. 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.

Google created the [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 manufactures 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 Yaghmour (2013). 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 stored in RAM and the remain hardware turns off. This allows the laptop to save battery power.

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

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 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 levels to processes depending on the components they are running and applies a threshold for each type of process. Android avoids the state by reaching this threshold and killing tasks.

- **Binder** is an mechanism adopted by Android that was based on OpenBinder. By 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 Gargenta (2013).

- **Anonymous Shared Memory (ashmem)** is another 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 Yaghmour (2013). 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 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 to fire an alarm when the earliest event is to run Stultz (2011).

- **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 eLinux webpage (2012). 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, Android inherited the user-based permission model from Linux that will be explained further. 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 of the calling process Dubey and Misra (2013).

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 . 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 world wide 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”. We are used to the . However, it is heavy to mobile devices. 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. The is designed to achieve good results 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` contains the android jar file for the targeted version of Android, denoted by `x` (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 Android webpage (2013a).

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 just stops being visible when another activity comes to the foreground. When this happens, `onPause()` is called. If the system needs memory to execute activities with higher priority, the activity is killed. If the activity is requested to run again, it can continue the previous task. The activity may also be stopped through `onStop()`. While stopped it cannot go back to the previous task, but might be restarted through `onRestart()`. At last, the activity is destroyed calling `onDestroy()`. The activity's lifecycle ends.

2.2.2 *Services*

When developers intend to launch some task that has no visible elements they 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.

Services may be called in two distinct ways. An application component, 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, if 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

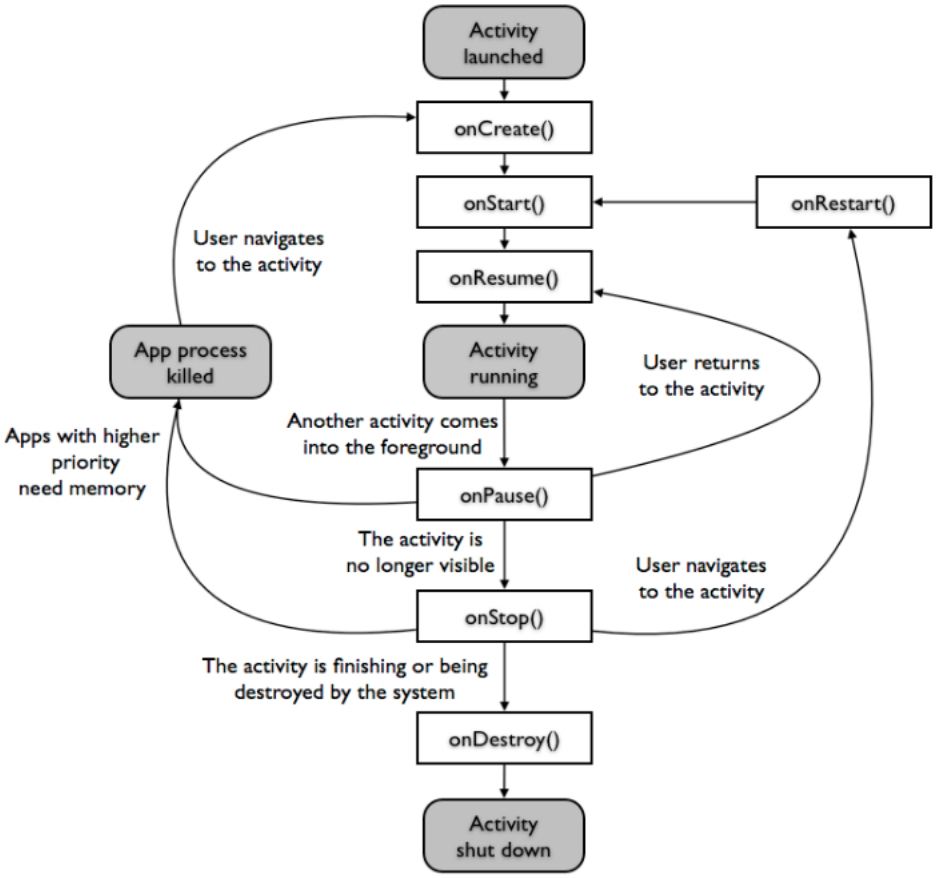

Figure 2.: The activity lifecycle

is bound to some application component, being destroyed after that. Note that the same service may assume both forms, unbound and bound.

The service lifecycle shows these two approaches in Figure 3 Android webpage (2013h). 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 Droidguardian is able to perform indefinitely in the background, being started when the device boots, as we will explain further.

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

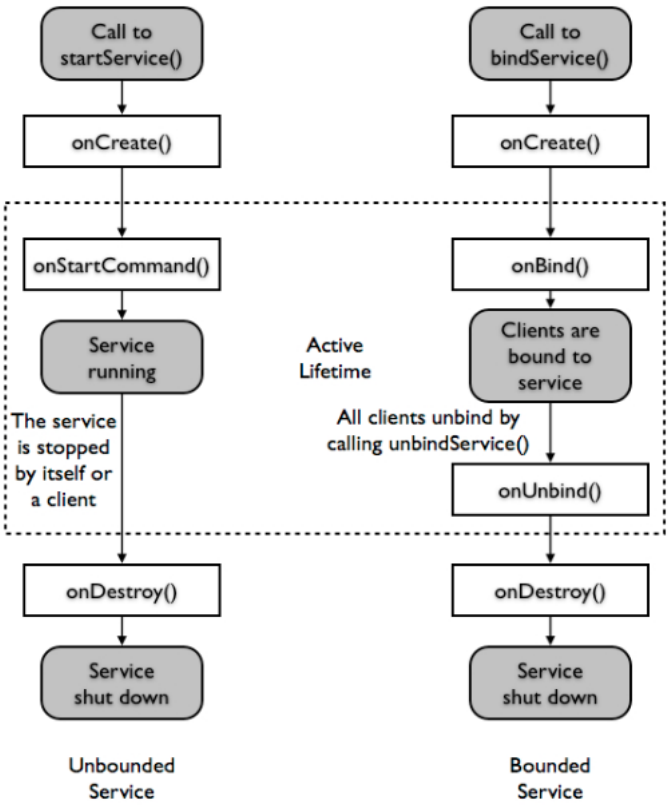

Figure 3.: The service lifecycle

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 share some data Android webpage (2013c). Figure 4 exhibits the way application components use intents to communicate.

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, as videos, audio, images, contacts, etc. Android supplies a consistent standard interface to data that also handles 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

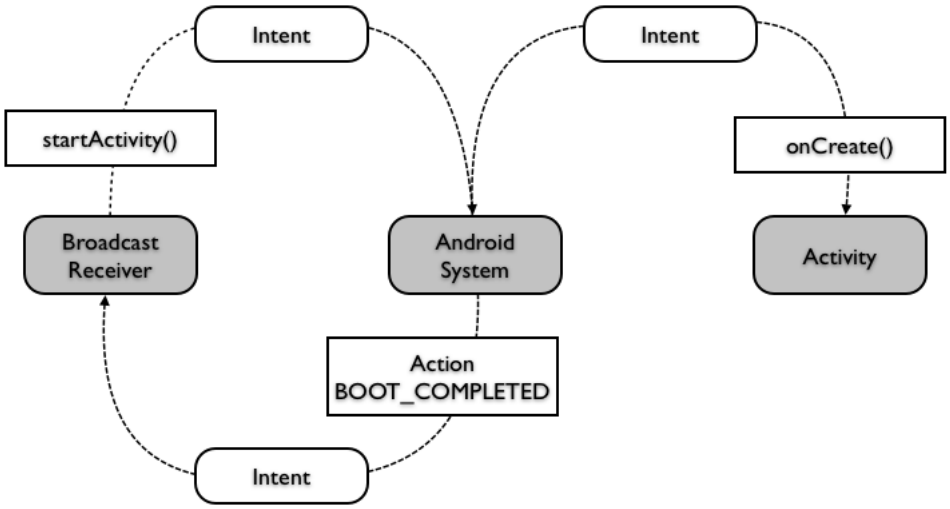

Figure 4.: Broadcasting an intent to start an activity

to external applications using a relational database like interface, providing CRUD (create, retrieve, update and delete) functions and a system Android webpage (2013e).

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 Android webpage (2013g). 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 . The user-based permissions model was originally developed for Unix environments, thus Linux takes advantage of it. In fact, the user-based permissions model has proven its good design concerning security issues over time. Every user registered in the system has an unique identifier number known as . Along with users, there are groups that are identified by its unique . 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 . 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 . Each file on a Linux system has three sets of permissions: owner, group and world. The owner and the group are those mentioned before. The world is considered to be 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 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 , and/or by every member of the file's group, and/or by all other users that have an account on the system Gollmann (2011).

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 that we'll see later). When an application is installed on the system, an new unique is assigned to it and the application runs under this . In addiction, all data stored by that application is assigned the same . 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 Marko Gargenta (2013).

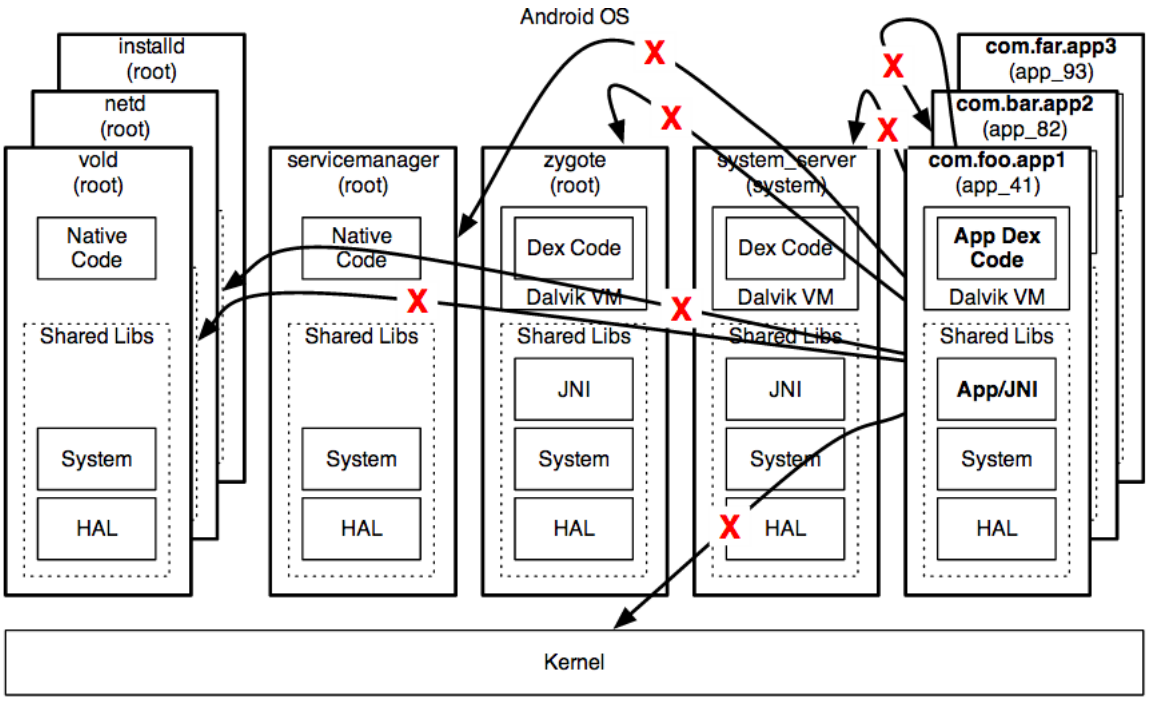

Figure 5.: Application sandboxing

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 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 , providing isolation access from other applications. This access control is enforced to all applications. However, if a user access the Linux kernel using the root will break down the sandboxing mechanism and be able to access any data stored in any application.

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 grants protection from the Android foundations. However this model enforces a mechanism that increases the risk of harm, as we will see later in the section. To overcome the related threats, Android began to use a component that has been in the Linux kernel in the last years, . This mechanism applies a model Smalley and Craig (2013) that reduces the effect of malicious software and protect users from potential flaws in code Android webpage (2013f).

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 world wide used mobile platform as Android, therefore a set of artifacts were developed granting 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 new permissions model know 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 their applications use when executing. Each resource requires a permission that must be declared on the Manifest file. At installation time, permissions 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 Android webpage (2013d) 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 falls into, 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 were required by other 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; if not, the app being installed will no be granted the permission. The user is never questioned about these permissions in order to start an installation;
- **SignatureOrSystem** These permissions follows 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.

An important rule that follows the Manifest permission is the Principle of Least Privilege Six (2011), which states that each application should keep permissions at its minimum, using the weak permission instead of a strong one that allows the application to execute tasks 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's trying to deploy his application into the market and establishes trust relationships between applications. Signing an application does not require a . 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 Android webpage (2013b).

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 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 . 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 Ludwig et al. (2013) . 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;
- 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 dynamic behaviors. Heuristics and similarities on-device data are applied. 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* 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.

4

RELATED WORK

Droidguardian was inspired by a powerful tool called *Little Snitch*, 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[1] 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 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 reading.

Since Android is an embedded Linux environment product, in the initial research phase to design Droidguardian we stumbled upon a very interesting tool, quite similar to Little Snitch, although much simpler, called *TuxGuardian*. This tool aims to exhibit in real time all outgoing internet connection attempts, allowing users to accept or reject such connections. TuxGuardian 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 SnitchObjective Development Software GmbH (2013b) 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.

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

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 address and a port. 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 might be seen as a triplet that includes a server, a port and a protocol. It has associated the process that triggered the connection and the owner of the process. The server is the remote internet address and Little Snitch handles them using numeric sets, hostnames and domains. The port points to services. Protocols (, or) states the behavior of the internet connection. Processes are applications, as Safari, Mail, etc, and UNIX processes, as *storeagent*, *ntpd*, etc. These processes are owned by an entity, as System, root, etc. There are two other properties that belongs to conditions: connection 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 an off enabled 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.

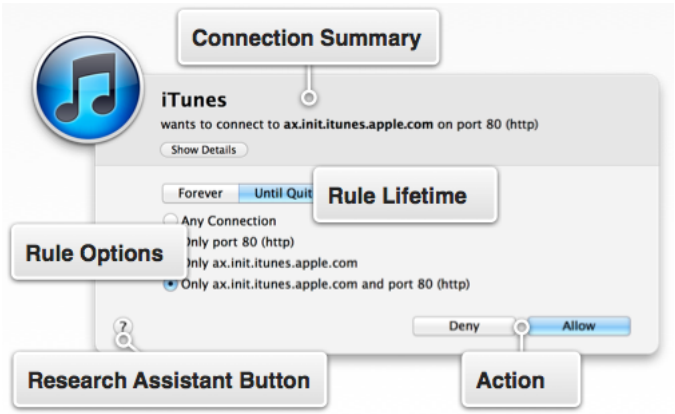

Figure 6.: Little Snitch Connection Alert window

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;
- *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 setted 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. The ability to refuse an internet connection cannot be performed at user level. Therefore, Little Snitch developers was forced to operate at kernel level, building a Kernel Extension Objective Development Software GmbH (2013a). The collected data from the bottom layer is sent to the layer above, the Network Filter. The matching process is done in this layer. At the top is placed the user interface that permit users to check information, define rules, etc.

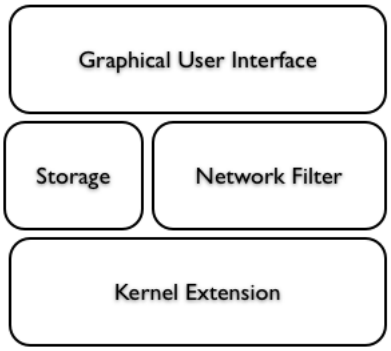

Figure 7.: Little Snitch architecture

4.2 TUXGUARDIAN

TuxGuardian da Silva (2006) 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 presents 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.

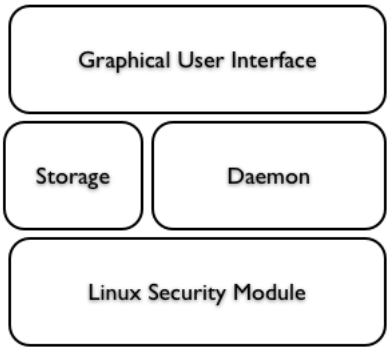

Figure 8.: TuxGuardian architecture

The *Security Module* is the bottom layer and takes advantage of the 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[2]. 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.

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 of the process that created the connection request. This communication process is established through local sockets.

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

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 component, denying all connections that are not placed in the storage file. In order to enforce a security measure, TuxGuardian keeps the MD5 hash of each process path. Through s, the daemon gets the process path name in the `/proc` directory and calculates its MD5 hash so that modified programs cannot access internet.

The or *Frontend* displays the notification windows. The user receives the process name (the complete process path, for instance `/bin/ping`) that created 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 da Silva and Weber (2006). 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 or the permission values: either *yes* or *no*.

5

TECHNICAL CONCEPTS

The development process of DroidGuardian comprised several parts and required some technical concepts. This chapter introduces those concepts along with relevant details that were fundamental to the development process.

5.1 LINUX SECURITY MODULES

Typically, developers find every resources they need in the Android . When some resources are missing, Android provides the so that they can bring the potential of native code to their applications. But, sometimes there are situations where it is necessary to go deeper into the Android software stack. This is one of those situations. In order to be able to reject an internet connection in real time, the operation must be executed at kernel level. The framework is the basis of DroidGuardian, providing the ability to deny an internet connection. This section explains in detail the framework and how it allows to operate with internet connections requests.

5.1.1 *Introduction*

In 2001, Peter Loscocco and Stephen Smalley wrote an article introducing the Loscocco and Smalley (2001). The main reason that led to the development of such mechanism was the flawed assumption that adequate security should reside in applications, leaving the role of the operating system behind Loscocco et al. (1998). They supported the idea that secure applications require secure operating systems. A strong concept related to operating systems security is *access control policy*. In a simple manner, this term specifies what operations associated with an object are authorized to perform. Linux kernel inherited from the UNIX security model the 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 advantages. For instance, every program executed by a certain user receives all of the privileges associated with that user. Therefore it is able to change the permissions of all user's objects, creating potential security threats. In this sense, a was purposed to protect the system against vulnerabilities left by other access control models. In the operating system

constrains the ability of a subject to perform an operation on an object, depending on the security attributes. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system kernel checks these security attributes in order to allow or deny the access.

At the Linux Kernel 2.5 Summit, the , based on the security issues previously mentioned, presented their work on , a security mechanism of a flexible access control architecture in the Linux kernel. reiterated the need for such support in the mainstream Linux kernel. Other projects were presented to enforce access policies, namely , and POSIX.1e capabilities. Given these projects, Linus Torvalds decided to provide a general framework for security policy, called . This framework allow many different access control models to be implemented as loadable kernel modules. Linus enforced that should be truly generic, where using a different security model was a question of loading a different kernel module. The framework should also 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 Wright et al. (2002).

This security framework has motivated developers and gave them freedom to build their own according to how they consider that kernel objects should be accessed. [1] was originally developed by the and has been in the mainstream kernel since version 2.6 (December 2003). presents three forms of access control, , and . It uses the filesystem to mark executables when keeping track of permissions.

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 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 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 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, 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 `http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison`.

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

Recently, Yama has been added to the mainstream kernel since version 3.4 (May 2012). Yama is a that collects a number of system wide security protections that are not handled by the core kernel itself.

Since the first release of the framework that new updates are committed in almost every new version of the Linux kernel. It is important to refer that between version 2.6.25 and 2.6.27, the boot engine changed and became no longer a removable module. Since then, the is loaded at compile time.

5.1.2 *Design*

The basic abstraction of the interface is to intercede in 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 allow modules to execute this task lies in *hook* functions that are placed in the kernel code, as shown in Figure 10.

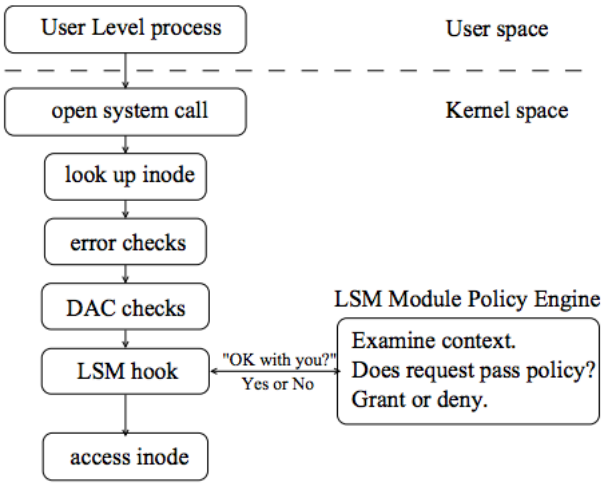

Figure 10.: LSM hook functions architecture

Immediately before the kernel access the object, represented as *inode* in Figure 10, the hook makes a call to a function that the must provide. The module, based on policy rules, either allow or deny the access, forcing an error code return in the last case.

5.1.3 *Implementation*

The framework comprises a few files in the kernel. Figure 11 highlights the relevant files that implement the security mechanism.

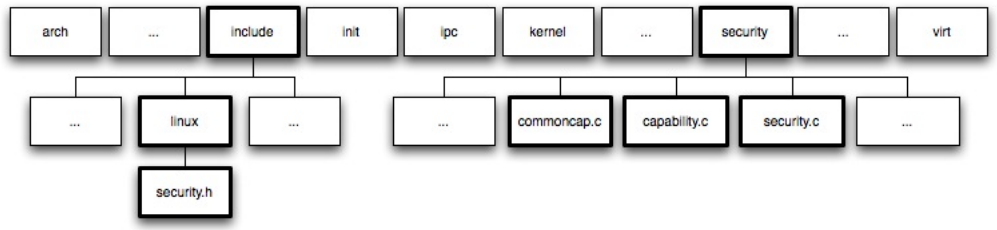

Figure 11.: framework files in the Linux kernel

Header file

The `include/linux/security.h` file contains the hook functions declarations. The source code may be divided into two parts, depending on the value of the conditional group `CONFIG_SECURITY` being true or false. In the first case, an extensive structure with pointers to all hook functions is declared. If false, only default functions are declared and the kernel loads the default security module. The code snippet in Listing 5.1, extracted from the Linux kernel v3.11, presents the initial function pointers in the structure `security_operations`.

```
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 (*capget) (struct task_struct *target,
          kernel_cap_t *effective,
          kernel_cap_t *inheritable, kernel_cap_t *permitted);
  int (*capset) (struct cred *new,
          const struct cred *old,
          const kernel_cap_t *effective,
          const kernel_cap_t *inheritable,
          const kernel_cap_t *permitted);
  int (*capable) (const struct cred *cred, struct user_namespace *ns,
      int cap, int audit);
  int (*quotactl) (int cmds, int type, int id, struct super_block *sb);
  int (*quota_on) (struct dentry *dentry);
  int (*syslog) (int type);
  int (*settime) (const struct timespec *ts, const struct timezone *tz);
  int (*vm_enough_memory) (struct mm_struct *mm, long pages);
```

Listing 5.1: Security structure declaration (Linux kernel v3.11)

Along with the structure, the functions prototypes are declared, as shown in Listing 5.2. Some security hooks are declared depending on conditional groups:

- `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 XFRM framework, that implement 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.

```
int security_ptrace_access_check(struct task_struct *child, unsigned int mode);
int security_ptrace_traceme(struct task_struct *parent);
int security_capget(struct task_struct *target,
        kernel_cap_t *effective,
        kernel_cap_t *inheritable,
        kernel_cap_t *permitted);
int security_capset(struct cred *new, const struct cred *old,
        const kernel_cap_t *effective,
        const kernel_cap_t *inheritable,
        const kernel_cap_t *permitted);
int security_capable(const struct cred *cred, struct user_namespace *ns,
      int cap);
int security_capable_noaudit(const struct cred *cred, struct user_namespace *ns,
           int cap);
int security_quotactl(int cmds, int type, int id, struct super_block *sb);
int security_quota_on(struct dentry *dentry);
int security_syslog(int type);
int security_settime(const struct timespec *ts, const struct timezone *tz);
int security_vm_enough_memory_mm(struct mm_struct *mm, long pages);
```

Listing 5.2: Security functions declaration (Linux kernel v3.11)

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 allow access to all kernel internal objects. Listing 5.3 exhibits some of these hooks' source code.

```
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);
```

```
}

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

static inline int security_quota_on(struct dentry *dentry)
{
  return 0;
}

static inline int security_syslog(int type)
{
  return 0;
}
```

Listing 5.3: 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.

Linux capabilities

Linux capabilities were designed to provide a solution to the UNIX-style user privilege set 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. Unfortunately, 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 Wikipedia (2013).

In the scope of , a set of functions, called *common capabilities*, were developed to give the security framework a default behavior in the case no other is loaded. These functions are plugged in the kernel to overcome the problem mentioned above. In `security/commoncap.c` we can see the source code of these functions.

If no is loaded, there must be a default function hook that does not execute any operation and let the process access kernel internal objects. The file `security/capability.c` have all hook functions with the default code. If the return type is `void`, functions have no operations, otherwise is `int` and functions just `return 0`, which is the value to turn the hook permissive. Listing 5.4 shows some of these 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;
}

static int cap_bprm_check_security(struct linux_binprm *bprm)
{
  return 0;
}

static void cap_bprm_committing_creds(struct linux_binprm *bprm)
{
}
```

Listing 5.4: Capability functions (Linux kernel v3.11)

These functions are called in the structure `security_operations` if the respective 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: `security_fixup_ops` function (Linux kernel v3.11)

Framework initialization

The header file mentioned in Figure 5.1.3 declares some functions in charge of getting the 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: Framework initialization functions (Linux kernel v3.11)

These functions are implemented in `security/security.c`. The first function being executed is `security_init`. The source code is present in Listing 5.7.

```
/* Boot-time LSM user choice */
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: `security_init` function (Linux kernel v3.11)

At first, the default module is loaded with the available routines cited in 5.1.3, by `security_fixup_ops(&def`
Then `security_init()` updates the kernel's security structure `security_ops` with the data ear-

lier initialized and makes a call to `do_security_initcalls()` that implements a 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: `do_security_initcalls` function (Linux kernel v3.11)

The callbacks `__security_initcall_start` and `__security_initcall_end` are declared in `include/linux/init.h` 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: `init` callbacks (Linux kernel v3.11)

LSM registration

There are several implementations adopted in the kernel, but this only runs one at a time. Therefore, there must be a way to register the desired . This is achieved through the execution of `register_security(stru security_operations *ops)`, exhibited 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: `register_security` function (Linux kernel v3.11)

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

There is other important function related to the registration, that is `security_module_enable`. Each must pass this method before registering its own operations to avoid security registration races. This method may also be used to check if the is currently loaded during kernel initialization. Listing 5.11 presents this function.

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

Listing 5.11: `register_security` function (Linux kernel v3.11)

At last, the security functions declarations previously mentioned in Figure 5.1.3, are implemented by returning the function callback present in the structure `security_operations`. A code snippet is available at 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: `register_security` function (Linux kernel v3.11)

Security functions in the kernel

Security functions presented in the previous subsection are called depending on each objective. For instance, the `socket_create` hook is part of the socket implementation, in `net/socket.c`. 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: `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`.

5.2 LINUX KERNEL MODULES

5.3 SOCKETS

Sockets are a well known mechanism that provides through Unix file descriptors (since this project is based on the Unix environment). This section introduces relevant details regarding both internet sockets and local (Unix domain) sockets. 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 implemented at user space followed by some particularities of socket implementation at kernel level.

5.3.1 *User space sockets*

Unix systems provide a programming interface to easily carry out tasks using sockets. This 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 or . 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);
   ```

   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);
   ```

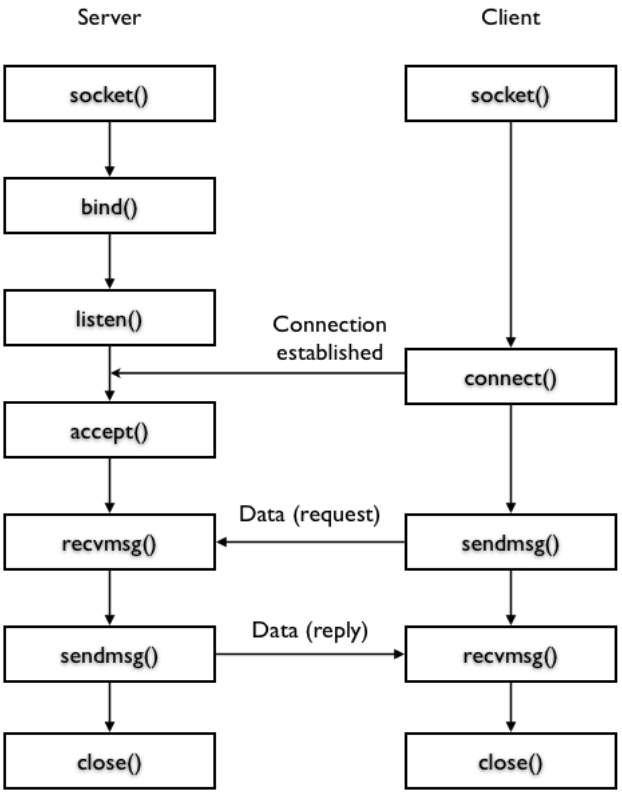

Figure 12.: Typical server-client based model of sockets

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);
```

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);
```

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);
```

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);
```

This 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, calling the `close()` primitive:

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

The socket descriptor is passed as argument.

5.3.2 *Address Formats*

As previously mentioned, in the primitives `bind()`, `connect()` and `accept()` the argument *address* points to a `struct sockaddr` 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 version, IPv4 or IPv6, respectively, and a `struct sockaddr_in` 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];
}
```

This structure defines the required data to create an internet address: the port and the address Hall (2012). 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 `struct in_addr` that contains an `unsigned long` to store the address value:

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

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

In local sockets, where the family is defined by `AF_UNIX`, the address will be set using `struct sockaddr_un`:

```
#define UNIX_PATH_MAX    108

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

In local sockets, the address is defined by the path of a file, in `sun_path`. In the scope of this project, there are two types of path (called namespaces) that is 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[5].

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

5.3.3 *Address Lookup*

Sockets store addresses as `unsigned long`, but they are displayed to users through the dotted notation: `x.x.x.x` in case of v4, or `x:x:x:x:x:x:x:x`, in case of v6. 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);
```

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.

With internet address is also possible to get the host name and service name, using `getnameinfo`, 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);
```

5.3.4 *Kernel space 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[6]. 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);
```

5 http://man7.org/linux/man-pages/man7/unix.7.html
6 http://lxr.free-electrons.com

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;
}
```

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:

```
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);
(...)
}
```

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
    );
```

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;
}
```

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;
}
```

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 Stevens and Rago (2013). Also, the read operation applies a *gather* feature, collection all spread data.

5.4 ANDROID TOOLS

Useful tools that allow the development of Android applications are found in the Android . It is the case of the *Emulator*, placed at `tools/` and the , placed at `platform-tools/`. This section describes both tools regarding valuable their features to this project.

5.4.1 *Android Emulator*

Android provides 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, the performance of the Android Emulator may be considerable low when compared to a real devices.

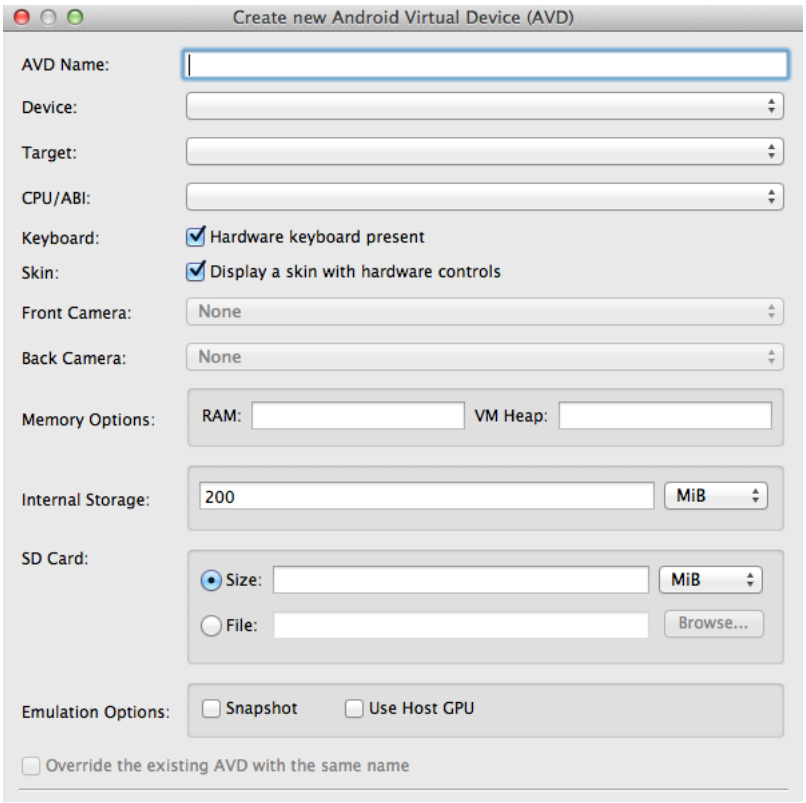

Figure 13.: Android Virtual Device configuration

The Android Emulator boots an Android image according to the configuration file. The allows to define hardware and software characteristics of a specific model to run on the Android Emulator. Figure 13 shows a snapshot of the 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 , as *Android 2.3.3 - API Level 10*, *Android 4.4 - API Level 19*, etc.

Once the is created, the Emulator may be launched through the Manager or using the command line. This last provides more options and fits better our needs when developing DroidGuardian. Some useful commands are presented as follow:

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

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

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

In order to choose a kernel of our own to run on the Emulator, it is used the *kernel* flag providing the image file system path of the kernel.

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

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

5.4.2 *Android Debug Bridge*

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 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 we start an Android Emulator, this becomes available to connection through adb. The following command shows all Android devices running on the computer:

```
# adb devices
```

If we have one Android Emulator running on the computer, the output returned is the following:

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

With adb it is possible to:

- install an Android application on the emulator/device;

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

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

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

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

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

- print the logcat output;

```
# adb logcat
```

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

```
# adb shell
```

There are more operations and options to perform with adb, that can be checked on the Android online page[7].

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

5.5 ANDROID NDK AND JNI

Android provides a powerful toolset that has multiple purposes, available at http://developer.android.com/tools/sdk/ndk/index.html. The Android was built to supply developers the capability to exploit the full power of mobile devices using native code. This is achieved through the , 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");
}
```

This `native` library implements a set of native methods called in Java. For instance:

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

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

This library consists of, at least, three files that should be placed at a folder called `jni`:

- 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 and allows to compile native code generating library files as well as executable files. The minimum intructions of an `Android.mk` file are presented as follows:

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

include $(CLEAR_VARS)

LOCAL_MODULE    := native
LOCAL_SRC_FILES := native.c

include $(BUILD_SHARED_LIBRARY)
```

This file specifies the native source files location, in `LOCAL_PATH`, the name of both the library and the native code file, in `LOCAL_MODULE` and `LOCAL_SRC_FILES`. The statement `CLEAR_VARS` indicates no dependent configuration disrupts compilation Ratabouil (2012). At last `BUILD_SHARED_LIBRARY` instructs to build a shared library.

The header file name follows a pattern: `<package_name>_<class>.h` . Imagine that the Java class that loaded the library is called `LoadLibrary` and the Java package that contains this class is called `com.android.droidguardian`. The name of the header file will be:
`com_android_droidguardian_LoadLibrary.h`
This file is automatically generated by a tool called *javah* provided by the . It was designed to build header files to the and may be called as follows:

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

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

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

5.5.1 *JNI concepts*

5.6 ANDROID SDK

6

IMPLEMENTING DROIDGUARDIAN

6.1 SETTING UP THE ENVIRONMENT

The development process of DroidGuardian comprised several stages that corresponding to the layer level that was being handled. The kernel module requested a completely different environment when compared to the Java layer implementation environment.

In order to fully understand the framework it was necessary to manipulate a real Linux kernel, as well as compile and install. Since the work machine used was a *MacBook Pro*, a new partition was created to install the *Xubuntu*. This is a different flavor of *Ubuntu Linux* operating system that provides a light user interface. Basically, the only used program was the console, because all required steps could be executed through the command line and using *Vim*. This is why a lighter and simpler user interface was enough to carry on the desired tasks on the Linux environment. The new partition was created using *rEFIt*[1].

Running Linux on a new partition provided speed and efficiency when setting up the Android environment, to build and launch a new image on the emulator. However, handling loadable kernel modules on a separated 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 pain to the host operating system. *VMWare* was used to virtualize a *Xubuntu* operating system, being *OS X* the host operating system.

Regarding the Android application development environment, *Eclipse* was chosen as the , 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, level 10.

1 http://refit.sourceforge.net

6.2 KERNEL MODULE

The main challenge in this project consisted in the manipulation of hook functions to properly handle socket connections at kernel level.

6.3 NATIVE LAYER

Users control internet connection attempts through an Android application. This application may be divided into two layers: *Native layer* and *Java layer*. This section introduces the former and the following section presents the last.

6.4 JAVA LAYER

The topmost layer of DroidGuardian has the only purpose of displaying the data to the user. This is achieved through an Android application, that is provided with both the necessary components and a powerful . Android components, presented in a previous chapter, were carefully studied in order to ensure that DroidGuardian was being built with the proper pieces. However, when compared to the usual Android applications lifecycle, DroidGuardian may be seen as a different kind of application that was not though to follow the good tips when it concerns the behavior of applications in mobile environments. But, more on that later.

The process flow of the Java layer is illustrated in Figure 14. It presents the following classes:

- `BootReceiver`: operates as a *BroadcastReceiver* that starts DroidGuardian after the device booting process.
- `Daemon`: acts as a *Service* to run in background while the device is on.
- `QueryActivity`: is the *Activity* responsible for displaying the data of queries to the user.
- `Query`: does not extend an Android component, being used to translate a native query into a Java query.
- `DialogWindow`: is bounded to the `QueryActivity`, building a fragment to display the dialog window.

DroidGuardian was design to run as a daemon, silently and unnoticed, until internet connection requests arose to trigger the dialog window. Also, applications may launch internet requests at any time since the device starts running. Therefore, DroidGuardian needs to start listening as soon as possible. Android fires an intent immediately after the booting process, allowing applications to receive it:

`android.intent.action.BOOT_COMPLETED`

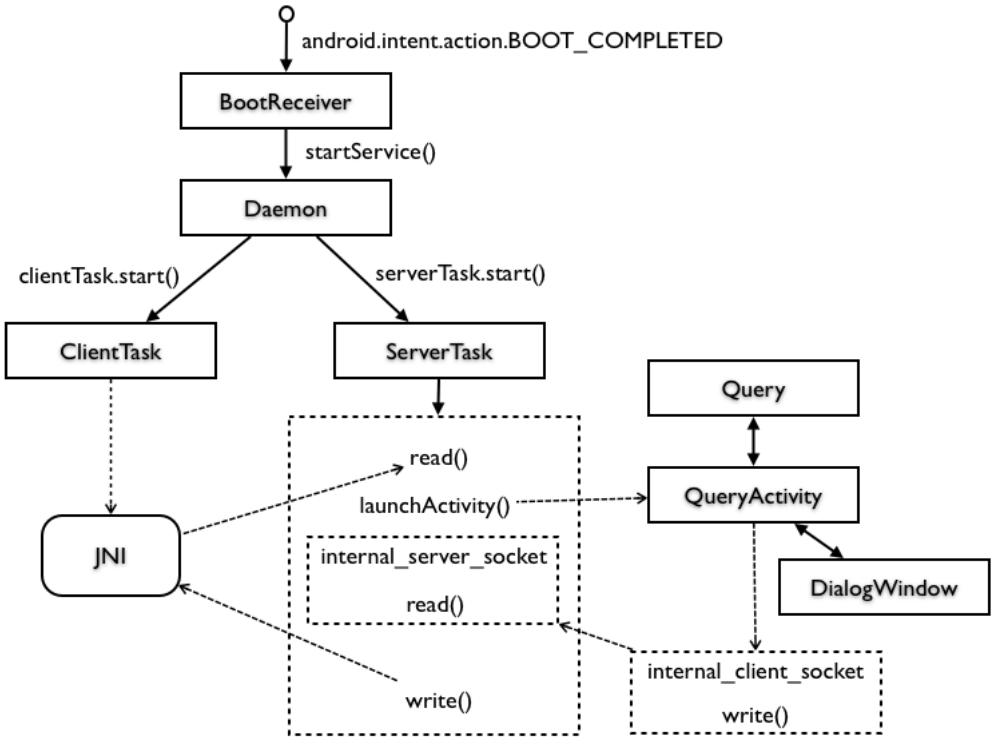

Figure 14.: Process flow in the Java layer

This intent is received trough *Receivers* or *BroadcastReceivers* that overwrite a callback named `onReceive()`. This method is responsible for grabbing intents and triggering whatever action the developer wants. In this case, the `BootReceiver`'s `onReceive()` method starts the daemon allowing DroidGuardian to connect to the kernel module and start listening internet requests.

The following steps describe the process flow of the Java layer:

1. Android sends `BOOT_COMPLETED` intent action and `BootReceiver` grabs it.

2. `BootReceiver` starts the service `Daemon` after grabbing the intent.

3. The `Daemon` service launches two threads: `ServerTask` and `ClientTask`. The former operates as a server in the stream socket protocol and will run until an external perturbation, as low memory, destroys the service. If nothing happen, this socket server will run while the device is on. The last acts as the client socket in this connection. However, the client code is not implemented in this class but in the native library. This thread calls the native method `startDaemon()` that kicks off the native engine.

4. Once executing, the server socket starts a `while(1)` loop in which queries' data will be exchanged between the server and the client.

5. When the client gets a query sends it to the server, that receives it through the read() method of the `InputStream` interface.

6. Immediately after reading the query, the server invokes an instance of the `QueryActivity` class through intents, transmitting the query's data.

7. Along with this call to `QueryActivity`, the server also initializes a new server socket, called internal server, that will handle the communication between the `Daemon` and the `QueryActivity`.

8. The `QueryActivity` gets the query's data in a special format. Then, creates an instance of the `Query` class, which has as instance variables the fields that will ultimately display the information to the user.

9. This process culminates with the execution of the `DialogWindow`. The `QueryActivity` instantiates a new Activity Fragment and exhibits it through the `show()` method.

10. The `DialogWindow` fills itself with a View that contains a text, a spinner and buttons. The text displays the internet connection request information so that the user may decide what option to chose in the spinner and what button to click on.

11. Once the user clicks a button, the `DialogWindow` executes a method that provides from a Java Interface and is implemented in the `QueryActivity` class. This method starts the internal client socket that will send the user's action to the internal server, listening on the `Daemon`'s server loop. After sending the message, the internal client closes itself.

12. The internal server gets the information, stores it in a variable and closes itself.

13. At last, the server reads this variable's value and sends the message back to the native client.

14. This process is repeated every time a new connection request reaches the Java layer.

6.5 DECISIONS

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 taken and its explanation.

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 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.

Service and Activity communication

The communication between the `Daemon` and the `QueryActivity` is established through local sockets. This is the third nested socket connection that takes place since DroidGuardian intercepts an internet connection attempt in the kernel and displays it to the user.

7

USING DROIDGUARDIAN

8

CONCLUSION

Conclusion and Future Work

BIBLIOGRAPHY

Android webpage. Activities. http://developer.android.com/guide/components/activities.html, 2013a.

Android webpage. Signing your applications. http://developer.android.com/tools/publishing/app-signing.html, 2013b.

Android webpage. Intents and intent filters. http://developer.android.com/guide/components/intents-filters.html, 2013c.

Android webpage. Manifest permissions. http://developer.android.com/reference/android/Manifest.permission.htm,, 2013d.

Android webpage. Content providers. http://developer.android.com/guide/topics/providers/content-providers.html, 2013e.

Android webpage. Validating security-enhanced linux in android. http://source.android.com/devices/tech/security/se-linux.html, 2013f.

Android webpage. Android security overview. http://source.android.com/devices/tech/security,, 2013g.

Android webpage. Services. http://developer.android.com/guide/components/services.html, 2013h.

Bruno Castro da Silva. Tuxguardian webpage. http://tuxguardian.sourceforge.net/, 2006.

Bruno Castro da Silva and Raul Fernando Weber. Tuxguardian: Um firewall de host voltado para o usuário final. Technical report, Instituto de Informática - Universidade Federal do Rio Grande do Sul, 2006.

Abhishek Dubey and Anmol Misra. *Android Security: Attack and Defense*. CRC Press, 2013.

eLinux webpage. Android logging system. http://elinux.org/Android_Logging_System, 2012.

Aleksandar Gargenta. Deep dive into android ipc/binder framework. Android Builders Summit, 2013.

Dieter Gollmann. *Computer Security*. Wiley, 2011.

Brian Hall. Beej's guide to network programming: Using internet sockets. Technical report, 2012.

Peter Loscocco and Stephen Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the* 21^{st} *National Information Systems Security Conference*, pages 303–314, October 1998.

Adrian Ludwig, Eric Davis, and Jon Larimer. Android: Practical security from the ground up. Presented at Virus Bulletin Conference 2013, 2013.

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

Objective Development Software GmbH. *Little Snitch 3 Documentation*, 2013a.

Objective Development Software GmbH. Little snitch. http://www.obdev.at/products/littlesnitch/index.html, 2013b.

Sylvain Ratabouil. *Android NDK Beginner's Guide*. Packt Publishing, 2012.

Jeff Six. *Application Security for the Android Platform*. O'Reilly Media, Inc., 2011.

Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. Technical report, Trusted Systems Research - National Security Agency, 2013.

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

John Stultz. Waking systems from suspend. https://lwn.net/Articles/429925, 2011.

Wikipedia. Capability-based security, September 2013. URL http://en.wikipedia.org/wiki/Capability-based_security.

Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., 2013.

INDEX

Part II

APENDICES

A

SUPPORT WORK

Auxiliary results which are not main-stream

B

DETAILS OF RESULTS

Details of results whose length would compromise readability of main text.

C

LISTINGS

Should this be the case

D

TOOLING

(Should this be the case)

Anyone using LaTeX should consider having a look at TUG , the TeX Users Group