École polytechnique de Louvain

Modifying Android for security analysis

Author: **Simon Lardinois**
Supervisor: **Axel Legay**
Readers: **Cristel Pelsser, Jeroen Beckers**
Academic year 2022–2023
Master [120] in Computer Science

Acknowledgement

I would like to thank the few people that helped and supported me throughout this thesis. First of all, I would like to thank Jeroen Beckers for all the discussions during which he provided lots of insights and pointers. Thank you also to my supervisor, Axel Legay, which was always available to quickly reply to my questions and provide feedback on the thesis.

Finally, thank you to NVISO, without which this thesis would not have seen the light of the day.

Table of contents

1 Introduction **4**

2 Background information **6**

2.1 Overview of Android 6

2.1.1 The architecture of Android 6

2.1.2 The Android boot process 11

2.1.3 Security in Android 14

2.2 Android application security 19

2.2.1 SSL pinning 19

2.2.2 Google Play Integrity 20

2.2.3 Root detection 21

2.2.4 Debugger detection 22

2.2.5 Integrity protection 22

2.2.6 Security assessments 23

3 The current state of Android security research **24**

3.1 The existing solutions 24

3.1.1 Magisk 24

3.1.2 Frida 26

3.1.3 Objection 26

3.1.4 Riru 27

3.1.5 LSPosed Framework 28

3.2 The shortcomings 29

4 InsecureOS **30**
4.1 Architecture of InsecureOS . . . 30
4.1.1 InsecureOS core . . . 31
4.1.2 InsecureOS modules . . . 33
4.2 Installation of InsecureOS . . . 35
4.2.1 The InsecureOS update package . . . 35
4.2.2 The installation flow . . . 37
4.3 Update of InsecureOS . . . 40
4.4 The InsecureOS boot process . . . 42
4.5 Default InsecureOS modules . . . 43
4.5.1 Frida-server . . . 43
4.5.2 Frida-inject . . . 43
4.5.3 Move Certificate . . . 44
4.5.4 MagiskLess-Riru . . . 44
4.5.5 Universal SafetyNet Fix . . . 46
4.5.6 MagiskLess-LSPosed . . . 47
4.5.7 Logger . . . 49
4.5.8 SSL Pinning Bypass . . . 50
4.6 Improvements over existing solutions . . . 51
4.6.1 InsecureOS vs Magisk . . . 52

5 InsecureOS in practice **55**
5.1 Setup of the tests . . . 55
5.2 Evaluation of InsecureOS . . . 57
5.2.1 Root detection . . . 57
5.2.2 Intercepting the application communications . . . 59
5.2.3 Conclusion . . . 60

6 Going further... **61**
6.1 Core improvements . . . 61
6.2 Improvements to existing modules . . . 62
6.3 Modules to implement . . . 62

A The InsecureOS sources **64**

1 Introduction

Over the last few years, smartphones have made their way in the pockets of most people around the world. Over 75% of those are powered with Google's Android operating system. Many companies therefore developed Android applications for their users to be able to manage their accounts from their smartphones. These companies range from social media platforms to banks and identity providers. Many of these applications deal with sensitive user data which the user would not want to be shared in any way.

Due to the popularity of the operating system, many threat actors started targeting it by developing Android malware. Some of these malware are very basic and simply attempt to trick the user into providing their sensitive information (e.g. their login credentials for their bank account). Others are much more advanced and will attempt to gain elevated privileges on the device and directly attack the installed applications.

With the growing number of malware targeting Android, applications dealing with sensitive data started implementing security controls to prevent such malware to obtain the sensitive data from the application. These security controls mostly aim at either detecting the presence of malware and/or preventing malicious actors from investigating them.

In addition to the implemented security controls, applications dealing with sensitive data are often audited through security research and security assessments. These aim at identifying potential security vulnerabilities that could be leveraged by malicious threat actors, in order for them to be fixed before the application is made available. Such process can also be used to evaluate the security controls of the application and improve them if necessary. This process helps at mitigating the potential impact of malware for the users of the application.

However, security research and security assessments are performed in much the same way as a malicious threat actor would investigate the application. As such, the security controls implemented by the application to impede the threat actors will often also impede the security researchers investigating the application. This therefore requires the security researcher to spend significant time to circumvent the security controls of the application before being able to investigate it.

The developers of the application may sometimes provide a specific version of the application that does not implement the security controls, however this is not always the case. If the application uses third-party libraries that implement the security controls, it is often out of the control of the application developers.

Many tools and documentation exist that can be used to circumvent the security controls implemented by the applications. Some of them may work well out of the box for most applications, however for applications implementing advanced security controls, it will often not be the case.

When conducting security research or a security assessment on such an application, it is therefore required to spend a significant amount of time to reverse-engineer the application and identify how the security controls are implemented to be able to circumvent them.

In addition, many of the tools used to evaluate the applications, while powerful, have drawbacks that may render them unusable during the investigation of specific applications.

During this thesis, a new solution, InsecureOS, was created and developed to overcome the limitations of the existing solutions in order to make security research and security assessments more efficient.

The thesis will first cover the existing solutions and their limitations and will then introduce InsecureOS, its technical details, and reasons why it makes security assessments more efficient.

2 Background information

2.1 Overview of Android

Android is a Linux-based operating system targeting embedded devices. It started as an operating system for mobile devices, but quickly gained in popularity and is now also used for tablets, smart televisions and much more.

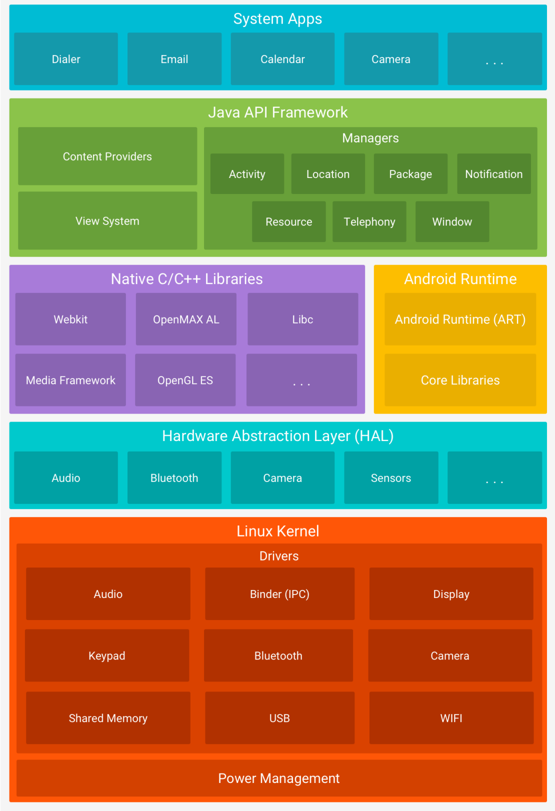
The development of Android started in 2003 by the company *Android Inc*, and was bought soon after by *Google* in 2005. Google then started promoting the operating system and pushed it to the public in 2008.

Google maintains Android under its *Android Open Source Project* (AOSP) [1]. The project contains all the different parts of the system, including external open-source tools that are included in the operating system. Although Android itself is open-source, device specific features (such as the camera) are to be implemented by the device manufacturers and are rarely made open-source. Manufacturers also often modify the system to differentiate themselves from the other manufacturers. For example, Samsung replaced the default Android GUI with its *One UI* [2] GUI.

2.1.1 The architecture of Android

Android differentiates itself from most other Linux systems as it consists of a full software stack instead of just the usual Kernel and GUI (see Figure 2.1). In particular, Android provides extensive frameworks in high-level languages that can be used by developers to easily develop in a device-agnostic manner.

The core of the system is the Linux Kernel [3]. Much like in any other Linux system, the Kernel is responsible for most of the hardware level interaction and management. In particular, the Kernel manages the network interfaces and protocols, it handles the thread management and low-level memory management. The Kernel is also responsible for the communication between processes using *Binder* [4, 5], an



Source: https://developer.android.com/guide/platform

Figure 2.1: The Android software stack

Android addition to the Linux Kernel. Binder is not the only change made to the Linux Kernel by Android. Some of them, such as *Anonymous Shared Memory*, were merged into the main-line Linux Kernel, while some did not and stayed Android specific.

On top of the Kernel lies the Android *Hardware Abstraction Layer* (HAL) [6], a powerful concept in Android allowing device manufacturers to provide an interface between the device hardware (fingerprint sensor, camera, ...) and the Android framework APIs. This allows Android to use the device hardware seemlessly, regardless of how the manufacturers implemented them, due to the generic interface that the manufacturers have to follow.

Then come the native C and C++ libraries, and the *Android Runtime*. The native libraries, much like on any other Linux system, provide a set of core C and C++ libraries that native applications can use. While most Linux systems use

GLibC, Android uses *Bionic* [7], an implementation of the standard C libraries, optimized for low-memory and low-power systems.

The Android runtime consists of the Android Java virtual machine, aptly named *ART* [8, 9] for *Android Runtime*, and core libraries. Prior to Android 7.0 (Nougat), the *Dalvik* [9, 10] virtual machine was used instead of *ART*. All applications on the device run inside this virtual machine, which allows Android to manage the applications more easily, perform some optimizations, and provide the powerful Android Framework.

The Android Framework is a set of Java APIs made available to all the applications on the system through the Android Runtime. It is a very complete set of APIs that allows application developers to efficiently manage their application life cycle and interactions with the system, without having to consider the differences between the devices or low-level implementation details. The extensiveness of the Android Framework is what sets Android apart from most operating systems, and is the main reason of the popularity of the operating system for embedded devices.

Finally comes the applications installed on the device. These applications run inside the Android Runtime (*ART*) and use the Android Framework to provide most of their functionalities. Each application runs in its own process, and as a separate Linux user, which allows to prevent applications from accessing the system resources or the other applications resources using the Linux permission system.

2.1.1.1 The Android partitions

Unlike desktop operating systems which often use partitions scarcely, Android makes heavy use of partitions. It is a key part of the Android architecture and security. In particular, it makes updates of the system rather simple, while making it difficult for the device to be compromised [11].

Android devices come with many partitions, each with their designated role and configuration. Some of them are common across all manufacturers (such as `/system` or `/data` [12]), while others are vendor specific (such as `blcmd` on Samsung devices, or `batt_tp_para` on Huawei devices). Table 2.1 lists some of the key Android partitions which should be present on all devices running Android.

The `/`, `/system` and `/vendor` partitions (as well as some others such as `/product`) are mounted read-only to ensure that no modifications (legitimate or not) are made to them. Additional security features, such as *dm-verity* further ensure during the mounting process that the partitions have not been modified (see Section 2.1.3.1).

Partition	Permissions	Description
/	ro	This is the root of the Android file system. It contains the mount points of the other partitions. In the case of *System-as-root* devices, this is also the /system partition.
/system	ro	This partition contains the crux of the operating system. It contains most of the Android binaries, the service definition and other key components of the Android system. It also contains the Android framework (in the form of JAR files) and the system applications. On *System-as-root* devices, the system partition also contains the root of the Android file system and will be mounted to / instead of /system. /system will therefore just be a regular directory on the partition, containing what was described above.
/vendor	ro	Similar to the system partition, but for the vendors. Vendors may add additional binaries or system applications in this partition.
/data	rw,nosuid	This partition contains all the user data (applications, configuration, ...). A factory reset of the device essentially removes all data from this partition.
/sdcard	rw,noexec	This partition represents the physical SD card of the device. When the device does not support external SD card, it maps to a specific directory in the /data partition.

Table 2.1: Some of the key Android partitions common to all devices

Before Android 8, some of the above partitions (such as /vendor) were included in the system partition. This meant that device manufacturers would place all their modifications alongside the core Android components, which sometimes made the update process difficult.

Since Android 8, Google introduced *Project treble* [13], which aimed at segregating the manufacturer modifications from the core Android components, using different partitions (/vendor being the one used by manufacturers). This significantly simplified the update process, as updates to the core Android components could be performed by simply modifying the /system partition without risking breaking the manufacturer components.

On most new Android devices, each of the system partitions will be present twice, with a suffix (`_a` or `_b`) appended to the name (e.g. `system_a` and `system_b`). This indicates that the device uses Google's *Seamless update* mechanism [14]. On such devices, the system critical partitions will be duplicated, but the system will only use one of the two. When an update is in progress, it will modify the unused version, before switching to it through a restart of the device.

2.1.1.2 The boot image

While the above partitions contain most of the components of Android, the Kernel is not included in any of them. Instead, the Kernel is packed into a *Boot image* along with the ramdisk, the Android Recovery (see Section 2.1.1.3) and information for *Android Verified Boot* (see Section 2.1.3.1).

During the boot process, the boot image will be verified by the bootloader, which will then load and hand over the execution to the Android Kernel. The boot process is further explained in Section 2.1.2

2.1.1.3 Android Recovery

Android Recovery is a key component of Android's update mechanism and integrity protection. It is essentially an alternative minimal operating system to which the device can boot to during an update or in the event the device can no longer boot properly to the primary system.

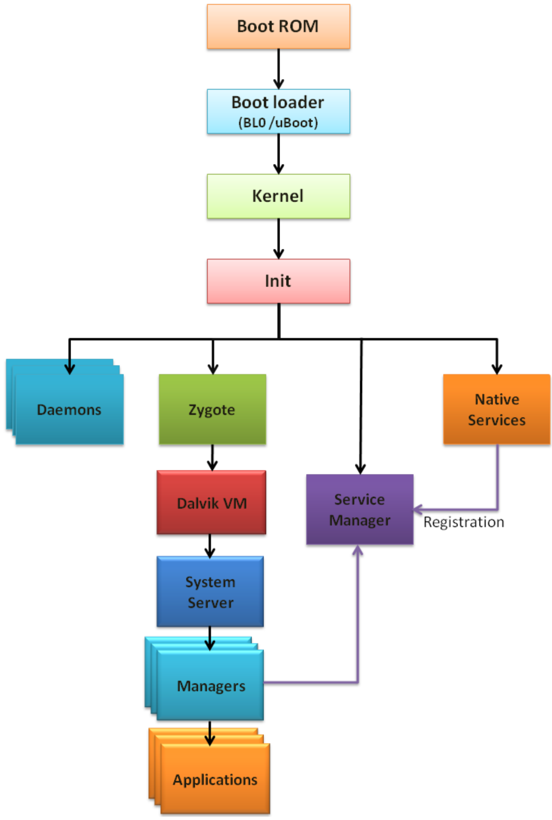
The Android recovery (often referred to as *recovery mode*) contains a simplified version of the Android Kernel, as well as a minimal GUI and a limited collection of tools used to manipulate the partitions and the system.

The recovery allows for applying update to the device by flashing the different partitions. This is usually done by the updater daemon of Android, which reboots the device with specific parameters in order to reboot to recovery and flash the recently downloaded update (*Over-The-Air* (OTA) updates [15]). It is also possible to manually provide an update package to the recovery, which will be flashed to the system.

During the update process, the Android Recovery is responsible for verifying the integrity and the authenticity of the update to apply by verifying its signature using a key derived from a key burned on the device by the manufacturer (see Section 2.1.3.1 for the details on the verification process). It is therefore only possible to apply updates that were signed by the manufacturer of the device, unless a third-party recovery was installed on the device.

2.1.2 The Android boot process

When turning on an Android device, the device will have to locate and execute the Android Kernel which will in turn initialize the system components and finalize the startup of the device. This is done through a complex flow of several components each verifying and executing the next one, starting from the Boot ROM until the Kernel which then executes Android `init` (see Section 2.1.2.1) to finalize the startup [16].



Source: https://www.embien.com/blog/android-boot-process-and-optimization/

Figure 2.2: The Android boot process

The Boot ROM (sometimes called *Primary Boot Loader* or *First stage Boot Loader*) is the very first software that is executed when powering on an Android device. It lies in a read-only section of the memory, ensuring that it will never be modified and that the device can (almost) always boot to it.

The Boot ROM is usually quite small in size, which greatly limits its capabilities. For this reason, most devices (if not all) implement a Secondary Boot Loader (sometimes called *Second Stage Boot Loader*). The role of the Boot ROM then

becomes to locate, validate and execute this Secondary Boot Loader.

The Secondary Boot Loader will be responsible for locating, verifying and executing the next element in the chain, which is often TrustZone (on ARMv7 devices) or Hypervisor (on ARMv8 devices), in the same way as the Boot ROM did for the Secondary Boot Loader. This process continues until the Android Boot Loader is loaded and executed.

The Android Boot Loader [17] (often the *Little Kernel* bootloader), is a more advanced bootloader with limited UI, hardware support and which provides utilities for flashing partitions. This is where the user ends up when rebooting to the bootloader with the command `adb reboot bootloader` or using a device specific key combination. One of the purposes of this bootloader is to finally execute the Kernel by locating the boot image of the device and loading the Kernel from it.

The Android Kernel boot process is not very different of that of Linux and will therefore not be discussed in details in this thesis. In short, it is responsible for initializing all of the subsystems through different stages. After the Kernel boot process is completed, the user-mode startup will begin with Android `init`.

2.1.2.1 Android init

`init` [18] is a key component of the Android boot process. It is the very first process started after the initialization of the Kernel, started with PID 1. It is responsible for initializing all other processes and services of the system as well as mounting the different partitions, initializing *SELinux* (see Section 2.1.3.2) and launching *Zygote* (see Section 2.1.2.3).

Among all of its other responsibilities, `init` will start all of the numerous daemons used by Android. Example of those are *installd* or *vold*, which are used to manage the installation of Android applications and the device partitions respectively.

Android `init`, unlike the init of most Linux systems, supports the powerful Android *system properties* (see Section 2.1.2.2), and uses .rc files to describe the services to launch on startup using the init.rc language [19]. This makes it rather simple to launch a service. By creating a .rc file describing the service, `init` will know which program to run and under which configuration. The .rc file provides extensive control over the context in which the service will run. In particular, it is possible to choose to run the service with root privileges or a specific SELinux context.

`init` defines several boot stages, which can be used to control when the services will be executed. An overview of the different stages can be found in Table 2.2.

Stage	Description
early-init	The first in the sequence, triggered after cgroups has been configured but before ueventd's coldboot is complete.
init	Triggered after coldboot is complete.
charger	Triggered if ro.bootmode == "charger".
late-init	Triggered if ro.bootmode != "charger", or via healthd triggering a boot from charging mode.
early-fs	Start vold.
fs	Vold is up. Mount partitions not marked as first-stage or late-mounted.
post-fs	Configure anything dependent on early mounts.
late-fs	Mount partitions marked as latemounted.
post-fs-data	Mount and configure /data; set up encryption. /metadata is reformatted here if it couldn't mount in first-stage init.
zygote-start	Start the zygote.
early-boot	After zygote has started.
boot	After early-boot actions have completed.

Source: https://android.googlesource.com/platform/system/core/+/master/init/README.md#trigger-sequence

Table 2.2: The boot stages of Android `init`

2.1.2.2 System properties

Android *system properties* are a set of key-value pairs that can be accessed and modified globally on Android. They are mainly used to store configuration settings and device information that can then be accessed by the relevant components of Android.

System properties can be grouped in different *namespaces*, which are denoted by dots in the name (e.g. `ro.buildnumber`). Some namespaces (such as `ro`) are special namespaces in which the system properties can only be modified by authorized processes. This authorization is enforced by the *property_service* started by `init`.

Several system property files are present on Android devices which contain a list of predefined properties that will be loaded by `init` during the boot process. The file `/system.prop` is an example of such a file.

2.1.2.3 Zygote

Zygote is a key process in the Android architecture. It is the parent process of all Android applications running on the device, and allows to greatly reduce the startup time of the applications.

During the device startup, `init` will start Zygote, which will initialize the Android Runtime (ART) and load most of the Android framework. When a user then opens an application, Zygote will fork itself to spawn the new application process, before dropping its privileges to prevent applications from running with elevated privileges.

As the Android framework and the application runtime have already been initialized in Zygote, the newly spawned application does not need to do so, which greatly reduces its startup time.

During the boot process, Zygote will also fork itself and specialize into the Android `system_server`. `system_server` is a process that manages all of the Android system services, and makes them available to the Android framework through the `Context.getSystemService` methods. Examples of such system services are the fingerprint service and the location service.

2.1.3 Security in Android

On Android, most applications will store sensitive information about their users on the local storage, a directory on the `/data` partition to which an application can place all of its data. This information can range from authorization tokens to access the APIs used by the application, to the banking information of the user.

In order to prevent applications or other unauthorized actors from accessing the sensitive information stored by the different applications, Android provides a number of security mechanisms, ranging from low-level integrity verification with Android Verified Boot (see Section 2.1.3.1) to high-level application permissions.

One way Android prevents unauthorized actors from accessing sensitive data of applications is by not making the root Linux user accessible. While the root user still exists and many processes (such as Zygote) are running in its context, Android does not provide a way for users or application to elevate their privileges. In practice, where other Linux systems would provide the `su` binary to elevate privileges, Android does not provide any such binary.

Another important concept in the Android security architecture is the *Application Sandbox* [20]. When installing an application in Android, a new Linux user is created, under which the application's directories are created (installation directory, local storage and external storage) and the processes run. This allows for the use the Linux permission system to prevent applications from accessing the directories and processes of the other applications. In addition, the user under which an application runs has very limited permissions, greatly limiting the access

of an application on the device. The permissions of the users (and therefore of the applications) can be extended using *Android permissions* [21], which adds the user to different Linux groups depending on the permission.

2.1.3.1 Android Verified Boot

Android Verified Boot (AVB) [22] is a security feature of Android used to verify the integrity of the system during the boot process of the device, ensuring that no illegitimate modifications have been made to the system. It relies on cryptographic signatures and hashes to verify the integrity of each component during the boot process.

AVB was introduced in Android 4.4 (KitKat). In its early days, it was simply called *Verified Boot* (version 1.0), and was re-branded to *Android Verified Boot* (version 2.0) [23] in Android 8.0 (Oreo) when it was made compatible with Google's Project Treble.

The verification of the integrity of the system with AVB starts as soon as the device starts, with Boot ROM. The Boot ROM contains a root certificate containing the public key of the vendor, which will be used as the root of the chain of trust for validating the integrity of the device. Indeed, this certificate located in a read-only section of the memory, can never be tampered with, which, in turn, ensures that the components verified with it have not been tampered with. This certificate is the basis of the integrity of the device.

Each component in the boot process must be signed by the vendor using this key, or a key contained in one of the previous components. This allows each stage of the boot process to verify the signature of the next stage using either the certificate of the Boot ROM, or a certificate embedded in them.

Concretely, the Boot ROM will verify the integrity of the Secondary Boot Loader before executing it, which in turn will verify the integrity of TrustZone or Hypervisor, ... This process continues until the Android Kernel is verified and executed.

Once the Kernel is booted-up, it will start the `init` process and the user-mode start-up will begin. At some point during the `init` boot process, `init` will have to mount the different system partitions. In order to make sure that those partitions have not been tampered with, `init` will first have to verify the integrity of those partitions. However, unlike for the verification of the boot image, the partitions are too big to be loaded entirely in memory and have their signature verified. Android `init` will therefore use the `dm-verity` (device-mapper-integrity) Kernel feature [24] to verify the integrity of the partitions.

`dm-verity` is a Linux device-mapper, which allows for the mapping of virtual block devices to physical ones. When mounting a block device (i.e. a partition),

`dm-verity` will verify the integrity of the blocks as they are read from the disk. This allows to verify the integrity of the partitions in blocks rather than all at once, which avoids the need of loading the full partitions in memory.

`dm-verity` works by calculating the hashes of the loaded blocks and validating them against hashes that were pre-calculated during the build of the system. If one of those hashes does not match the pre-calculated ones, the system will stop the boot process and show an error message to the user.

2.1.3.2 SELinux in Android

SELinux (Security Enhanced Linux) [25] was developed and incorporated in Linux with the goal of providing a Mandatory Access Control (MAC) framework with great granularity. It assigns labels to resources and processes, and defines rules describing how labels can interact with other labels.

Since Android 4.3 (JellyBean) SELinux was increasingly incorporated into Android. Nowadays with newer versions of Android, SELinux has become a key component of the Android security architecture and is sometimes referred to as SEAndroid [26].

In essence, SELinux is quite simple. It works by assigning *labels* (also called *contexts*) to the files and the processes, and by defining *policies* (rules) regulating the interaction between the defined labels. A label assigned to an object (such as a file) is usually called *type*, while a label assigned to a process is usually called a *domain*.

In addition to the types and domains, SELinux also defines *permissions*, which are the set of actions that domains can perform on types, if allowed by the SELinux policies.

A SELinux label is of the form `user:role:type[:level]`. If two processes have the same label (domain), they are virtually identical with regards to SELinux. Similarly, if two objects have the same label (type), they are identical with regards to SELinux.

In SELinux, it is also possible to group types together into *classes*. This allows to define rules on the class level instead of having to define the policies for each and every type.

A SELinux policy is usually written in the form `allow domain type:class permission`. For example, the policy `allow user_t bin_t:file read` allows the processes from the domain `user_t` to execute the action `read` on objects of type `bin_t` belonging to the class `file`.

On Android, SELinux works in the same way, although it was extended slightly to support system properties and the Binder.

Android only makes use of the role and type part of the SELinux contexts, choosing to use static values for the user (`s`) and the level (`s0`). The role is also limited to distinguish between processes (`r`) and objects (`object_r`). As such, processes in Android would be assigned a context of the form `s:r:domain:s0`, while objects will be assigned a context of the form `s:object_r:type:s0`.

During the boot process, Android will have to assign SELinux contexts to the different objects and processes, as well as load all the defined policies. This is done by `init` using several SELinux configuration files that can be found in a SELinux folder in most of the key system partitions.

The SELinux policies loaded during the boot process are pre-compiled during the build of the system, and are located in `/vendor` or `/odm` partitions.

2.1.3.3 TLS Certificate Authority stores

Similarly to any other system that performs TLS communications, Android requires a set of pre-installed certificate authorities to validate the TLS certificate of the server it connects to.

Android provides two different Certificate Authority (CA) stores: The system CA store, and the user CA store.

The system CA store contains all the pre-installed certificate authorities. It is located in the `/system` partition and can therefore not be modified by the user. This CA store will be used by default when an application performs an HTTPS request, to validate the TLS certificate of the server.

The user CA store on the other hand contains all the certificate authorities that are installed by the user. It is located in the `/data` partition and can therefore be modified by the user. Applications installed on the device can be configured during their development to use the user CA store to validate the TLS certificate of the server, but it will not be the case by default.

This distinction between the system CA store and the user CA store allows Android to protect the user of the device. Should a malicious actor be able to trick the user into installing a certificate they control, the malicious actor would still not be able to intercept the HTTPS communications of the applications, unless the application specifically allowed the use of the user CA store.

2.1.3.4 Rooting

Rooting is the process of making the root user available on an Android device. This is usually done by adding a binary such as `su` allowing the user to obtain root privileges. It is done by Android users that want more control over their devices, by security researchers who need privileged access, or by advanced malware aiming to gain full control over the device.

Rooting a device typically requires disabling Android Verified Boot (AVB) (see Section 2.1.3.1) in order to modify the boot image or the system partitions. Disabling the verification of the boot image by AVB is called *unlocking the bootloader* and will wipe the content of the `/data` partition.

Due to the lack of integrity verification following disabling AVB, and to the availability of the root user, the device and the applications installed on it are at greater risk of compromise by malware. For this reason, security critical applications (such as banking application) will often attempt to verify if the device is rooted and refuse to run if they detect that it is.

There exist several rooting solutions, but the most popular one by far is *Magisk* (see Section 3.1.1), which also provides support for modules and is able to hide itself to most applications.

2.2 Android application security

Due to the popularity of Android, it quickly became a prime target for malicious actors. Many Android users will have their device with them at all time, which increases the likelihood of the device being lost or be compromised.

Applications dealing with sensitive user data will therefore often rely on several security controls on top of the ones provided by default by the operating system. These security controls will be more or less advanced depending on the application, with security-critical applications often relying on a combination of all of them.

This section will discuss the most common security controls that applications can implement and that will be enforced at runtime, called *Runtime Application Self-Protection* (RASP).

2.2.1 SSL pinning

As an Android device is meant to be carried around by its user, the device will often connect to numerous networks, be it cellular network or Wi-Fi. Some of these networks may be insecurely configured or without any protection, which would allow the communications of the device to be intercepted and potentially tampered with.

To prevent the communications from being intercepted, applications should use encrypted protocols such as HTTPS whenever communicating with their back-end. To protect the user, Android requires the applications to use such encrypted protocols by default, although it is possible for applications to remove that restriction.

While using HTTPS greatly reduces the likelihood of the communications being intercepted, it is still possible for an attacker to intercept them if they somehow managed to obtain a valid TLS certificate (e.g. if an attacker controls one of the root certificate authorities installed on the phone). The attacker will simply have to intercept the TLS handshake and provide their TLS certificate instead of the legitimate server certificate.

SSL pinning is a security feature that will prevent an attacker from intercepting the communications, even if they managed to obtain a valid certificate for the domain. It is commonly used by security-critical applications such as banking applications, where the interception of the communications would have significant impact on both the user and the application.

When using SSL pinning, an additional verification will be performed after the usual validation of the TLS certificate of the server. This additional verification will rely on a hard-coded certificate in the application and compare it with the received

server certificate. During the development of the application, the server certificate (or its hash) will be embedded in the application, which will then only accept TLS connections with that exact certificate. This way, even if a valid certificate is presented, the TLS connection will be aborted if the certificate does not exactly match the hard-coded one.

Android provides a built-in mechanism to perform SSL pinning, using the Network Security Config [27]. In addition, many other third-party libraries provide ways to implement SSL pinning in an application, notably the popular OkHttp3 library [28].

Since SSL pinning prevents intercepting the HTTPS communications of the application, it is a protection that needs to be circumvented during a security assessment in order to be able to investigate the communications with the back-end of the application.

2.2.2 Google Play Integrity

Malicious actors (such as malware developers) will often rely on modified devices to investigate and identify issues in Android applications. They will root their device and modify the system to add many tools to help them identifying vulnerabilities and behavior that they could then exploit. Some advanced malware will also root the device they are installed on and use their elevated privileges to retrieve sensitive data from installed applications.

Security-critical applications or applications relying on local security controls will therefore want to prevent such malicious actors from investigating them and prevent malware from obtaining their sensitive information. Such applications can use Google's*Play Integrity API* [29] (formerly *SafetyNet Attestation API* [30]) to verify the integrity of the device.

Google Play Integrity is a security feature developed by Google and is part of the Google Play Services. It monitors the integrity of the device at regular intervals by collecting several key pieces of information in different ways. Google then provides the Play Integrity API that applications can use to retrieve the integrity status of the device from Play Integrity.

In practice, Play Integrity will monitor the value of several system properties related to AVB (see Section 2.1.3.1) and device integrity, such as `ro.boot.flash.locked` or `ro.boot.verifiedbootstate`, and will validate the collected data with the Play Integrity server, which knows the expected values for the device model.

On some devices, Play Integrity also supports *hardware-backed key attestation* [31]. When using key attestation, Play Integrity will use a hardware-backed

key to verify the state of the bootloader (locked or unlocked) and will use this information as indication that the device has been modified.

Once an application determines the device it is running on is a modified or rooted device, it is up to the application to decide what to do with that information. Some applications will terminate themselves, preventing malicious actors from running them, while others will display a warning message to the user, to inform them of the dangers involved in running the application on a rooted device.

In case the application decides to terminate itself, the protection will need to be circumvented during a security assessment in order to be able to run the application on a modified device and investigate it.

2.2.3 Root detection

While Google Play Integrity API (see Section 2.2.2) can be used to verify the integrity of the device and therefore to verify if the device is rooted, it is not without flaws. Indeed, Google Play Integrity can be circumvented without too much difficulties with a rooted device (i.e. by modifying the system properties to hide the AVB state). Several tools exists that can be found online for that purpose, including Magisk modules [32].

Security-critical applications may therefore decide not to rely on Google Play Integrity and instead use more robust methods to detect if a device is modified and/or rooted. Some applications will implement their own detection mechanisms, or use one of the many available third-party (often commercial) libraries specializing in this aspect (some notable ones are GuardSquare's *DexGuard* [33] or OneSpan's *Mobile Security Suite* [34]).

Applications or libraries implementing root detection will perform a series of verifications on the device, more or less advanced, to determine whether the device has been rooted or modified. The most basic root detection will check for the existence of the `su` binary in several places on the system (such as `/system/bin/su` or `/xbin/su`) and/or for the existence of specific applications on the device, known to provide rooting solutions (such as SuperSU or Magisk).

More advanced root detection mechanisms will perform much more involved verifications, such as verifying the SELinux configuration of the device, checking the mounted partitions and their configuration, verifying the existence of specific processes or if specific ports are opened, and much more.

Once again, once an application determines the device it is running on has been modified, it is up to the application to decide what to do with that information.

If the application decides to terminate itself, the protection will need to be circumvented during a security assessment in order to be able to run the application on a modified device.

2.2.4 Debugger detection

In addition to root detection, security-critical applications will also often implement *debugger detection*. Debugger detection aims at identifying if a debugger is currently attached to the application. This once again makes it more difficult for malicious actors to investigate the application and identify potential issues. It is most often used to strengthen the other security controls (root detection and SSL pinning) by making it much more difficult to circumvent them using a debugger.

Much like for root detection, application developers may develop their own debugger detection, or rely on third-party libraries (once again, GuardSquare's *DexGuard* or OneSpan's *Mobile Security Suite* are notable example).

There are many ways to detect if a debugger is currently attached to the application. The most basic method is to use `ptrace` [35], while more advanced methods will include enumerating the modules loaded in memory or to inspect the code section of the application in memory.

Once an application detects that it is currently being debugged, it will often terminate itself. In that case, it will be necessary to circumvent the protection during security assessments if the security researcher needs to use a debugger to manipulate the application.

2.2.5 Integrity protection

To circumvent all the RASP discussed so far, malicious actors will often attempt to repackage a modified version of the application, in which they removed or disabled the security controls. In addition, malware developers will often use existing applications which they will modify to include the malware, and convince users to install it by advertising it as the normal application or a better version of it. Applications willing to prevent these issues can implement some *integrity protection* mechanisms.

Integrity protection consists of verifying the integrity of the package of the application during its runtime. It can be done through the verification of the signature of the application, or additional signatures and hashes included in the application.

If an application detects that it has been modified illegitimately, the application will often terminate itself. During security assessments, it will therefore often become necessary to circumvent the protection in order to be able to modify the application package.

2.2.6 Security assessments

All the security controls discussed above can be implemented by an application to make it harder for a malicious actor to investigate and identify vulnerabilities in it. However, given enough time and effort, such protections can always be circumvented. It is therefore crucial for application developers to ensure that no security vulnerabilities are present in their application in the first place. This is done by regularly conducting security assessments on the application.

A security assessment, aims at investigating and identifying potential security issues in the application, much in the same way as a malicious actor would. The identified issues will then be reported to the application developers, which will then resolve the issues.

The security assessment can also be used to test the strength of the implemented security controls and provide feedback to make them more difficult to circumvent.

Security assessments on applications implementing advanced RASP are often difficult and time-consuming, requiring a lot of effort to circumvent the protections and be able to test the underlying application.

3 The current state of Android security research

When performing security research or a security assessment on an Android application, the application is usually run on a rooted or heavily modified device. In addition, debuggers and runtime manipulation tools will be used during the assessment to observe the behavior of the application and to circumvent the runtime protections, if any.

While this approach works for most assessments, it can be difficult if the application implements advanced runtime protections. The different protections will need to be circumvented, one after another, before being able to test the underlying application. Circumventing the runtime protections of such applications will often require reverse-engineering of heavily obfuscated code, which is very time consuming, leaving less time for the security researcher to investigate the underlying application.

3.1 The existing solutions

Security researchers have a lot of tools at their disposal to circumvent potential application runtime protections to be able to test the application.

This section will cover the *state of the art* of the available tools that can be used for this purpose.

3.1.1 Magisk

Magisk [36] is the current de facto solution to root Android devices. It is a very powerful suite of software that allows the user to customize their Android devices.

Being a rooting solution, Magisk provides a utility to access the root user of the device. This utility will require explicit authorization from the user before granting the root privileges to the application requesting them, which makes it safer than simply providing the su binary. In practice, it works by running a daemon with root privileges, with which the utility will communicate and send the requested command to.

Magisk supports the installation of *modules* [37], which can be used to achieve almost anything on the device. Magisk modules can modify any partition on the device, including read-only ones, register services to run at various stages of the boot process, modify system properties, including the ones in reserved namespaces (such as the `ro` namespace) or modify the SELinux policies of the device. Due to the popularity of Magisk, there exist many modules, ranging from simple modules modifying UI elements, to advanced ones that can be used to modify the Android framework at runtime.

Magisk also includes *Zygisk*, a powerful tool to inject in the runtime of any running application. Magisk modules can leverage Zygisk to modify the behavior of applications installed on the device. Magisk itself uses Zygisk to implement its *deny list*, which allows to hide Magisk from the applications for which the deny list is enforced. This can be used to circumvent root detection in most cases.

In practice, Zygisk uses the `LD_PRELOAD` environment variable from *ld.so* to load a shared library in the Zygote process during startup. This Zygisk shared library will hook several key methods of Zygote that are triggered whenever Zygote forks itself to spawn a new application, allowing Zygisk to inject itself in every application as soon as they are spawned. Zygisk will then load the libraries of the various Magisk modules which provide one, allowing those modules to be loaded in all the applications on the system.

In addition to Zygisk, Magisk also provides a *deny list* feature, which allows the user to select applications for which Zygisk should not load libraries in their process. This will prevent both Zygisk and any Magisk modules to be loaded in the applications process, making it more difficult for an application to verify if the device it is running on is rooted with Magisk.

Through the many features Magisk provides, it leaves a lot of artifacts on the device that can be picked up by the most advanced RASP libraries. Magisk being so popular, most such libraries will be tailored specifically for it, some of which being able to detect Magisk even when the deny list is enforced.

3.1.2 Frida

Frida [38] is a dynamic instrumentation tool compatible with Windows, macOS, Linux, iOS, Android and more. Security researcher can use Frida to inject JavaScript code into the process of an application, and interact with the process memory, native methods and, in the case of Android, Java methods.

In the context of Android security research and security assessments, Frida is often used to bypass the security controls (root detection, SSL pinning, ...) of the investigated application or to manually trigger flows inside the application and observe the application's behavior.

Due to its support for hooking native libraries, it is also possible to use Frida to hook methods from cross-platform mobile development frameworks (such as the C# methods of Xamarin [39]). This can be achieved by hooking the engine of the framework which is usually compiled in a native library loaded by the application.

Frida works by injecting a complete JavaScript runtime into the process of an application which can then be interacted with by the Frida client to inject JavaScript scripts in the context of the application. These scripts can then be used to hook the methods of the application, interact with the application's keystore or explore its memory.

To inject the JavaScript runtime into the application the process of the application, Frida will hijack the process using `ptrace` and load a shared library containing the runtime and a Frida agent that can be communicated with through the Frida client [40].

Despite Frida being very powerful, using it for security research can be quite time consuming, especially with applications that implement advanced runtime protections. Since Frida uses `ptrace` to hijack the process of an application and load additional modules in its memory, an application can prevent the use of Frida by `ptrace`-ing itself and check if it is being manipulated by Frida by inspecting the modules loaded in its memory. Security researchers will then need to first circumvent the debugger detection before being able to use Frida.

In addition, when using Frida, it is required to specify the exact method and/or memory address that needs to be hooked. This requires a fair amount of reverse-engineering, which can be difficult on applications implementing heavy obfuscation.

3.1.3 Objection

Objection [41] is a Frida-based tool that can be used to manipulate the runtime of Android and iOS applications. It contains a number of powerful and versatile

Frida scripts that can be used to manipulate and investigate the application. It also provides a command-line interface that can be used to load and execute the various Frida scripts dynamically.

Due to the collection of Frida scripts it provides, Objection makes it easier to investigate applications as it allows to quickly enumerate and trace methods (including printing their arguments, return values and stack-traces), or circumvent runtime protection of applications. In particular it can be used to bypass the root detection and SSL pinning of the most popular libraries.

Being Frida-based, Objection can also be detected in the same way as Frida, making it difficult to use it on applications implementing advanced runtime protections. In addition, while the Frida scripts can be used to bypass the most common runtime protections, it will often not be sufficient against more advanced runtime protection libraries, or on applications that are obfuscated.

3.1.4 Riru

Riru [42] is a Magisk module that allows other Magisk modules to inject shared libraries into any application on the device. Magisk modules leveraging Riru (called Riru modules), simply need to place the shared library that needs to be loaded into a specific folder in their module directory. Riru will then take care of loading the library into all the applications as soon as they are forked from Zygote.

With Magisk version 24 and the advent of Zygisk, Riru was deprecated since it provides the same functionalities as Zygisk which was included in Magisk by default. Most maintained Riru modules also migrated to Zygisk.

While Riru provides the same features as Zygisk, it works slightly differently. Instead of using the `LD_PRELOAD` environment variable used by *ld.so* as does Zygisk, Riru uses the *native bridge* feature of Zygote. By specifying a shared library into a specific system property (`ro.dalvik.vm.native.bridge`) before Zygote is started, Zygote will load the library while it starts. This allows to load arbitrary libraries into the Zygote process.

Riru leverages this feature to load its *riruloader* library, which will hook several methods in Zygote that are called when a new application is forked from it. Whenever an application is forked from Zygote, the registered hooks will be called, and Riru will load the shared libraries of the different installed Riru modules.

While Riru is quite powerful, Riru modules need to be developed to inject into the applications and achieve anything. Developing a Riru module is often a difficult task, requiring low-level C programming and managing the Zygote forking process. To develop a Riru module that can be used to investigate applications

by, for example, circumventing runtime protections, it also requires a lot of reverse engineering to identify how to circumvent the protection.

Since developing a Riru module takes a lot of time and effort, it is often easier to rely on existing Riru modules. However, while there exist modules to circumvent runtime protections (such as *Universal SafetyNet Fix* by *kdragon* [32] which can be used to circumvent Google SafetyNet), there are no Riru module that will work for every application.

3.1.5 LSPosed Framework

The *LSPosed Framework* [43] is a Zygisk/Riru module (and therefore Magisk module) that provides APIs to hook into the Android Runtime (ART). It supports modules in the form of regular Android APKs, which can therefore be installed as any other application on the device. LSPosed modules can use the LSPosed APIs to hook the Android framework or the application's specific code, monitor them, or modify their implementation.

LSPosed also provides a manager application, which can be used to enable/disable installed LSPosed modules and control in which application the module will be loaded.

The LSPosed API is based on the *Xposed Framework* API. Xposed is a legacy hooking framework that was popular before Magisk was developed, and which provides powerful APIs to manipulate the Java methods of the Android framework and the applications.

LSPosed uses *LSPlant* [44] to replace the Android runtime (ART) binaries on the system to allow to dynamically register hooks using the *Java Native Interface* (JNI). LSPosed then provides a system service that exposes the Xposed Framework API to the LSPosed modules and which handles the registration of the JNI hooks.

Since LSPosed patches the Android Runtime directly instead of the application, it has the advantage of being more difficult to detect than other tools such as Frida. Indeed, it does not change the application's signature, nor does it use `ptrace`-based mechanisms to hook the application.

LSPosed can easily be used to add monitoring to all applications on a device or to circumvent basic security mechanism as it allows to hook directly into the Android framework. However, LSPosed requires either Zygisk or Riru in order to be loaded into the targeted application. As such, LSPosed cannot be used when the Zygisk deny list is enforced on the targeted application. When working with an application that implements advanced runtime protections, this limitation may render the use of LSPosed impractical, as Magisk will no longer be hidden from the application.

3.2 The shortcomings

The tools discussed in Section 3.1 are powerful and can be used to circumvent the runtime protections of most applications. For example, simply enforcing the Magisk deny list will be enough to circumvent the most basic root detection mechanisms, while the default SSL pinning bypass script included in Objection will be enough to circumvent the most common SSL pinning.

However, most of the solutions discussed in Section 3.1 have drawbacks which make it possible for applications implementing advanced runtime protections to detect the use of such tools. In such cases, it is up to the security researcher to reverse-engineer the application (which is usually heavily obfuscated), identify the security mechanisms, and circumvent them using debuggers and runtime manipulation tools such as Frida.

This process is very time consuming and often difficult to implement in practice. For security research and security assessments which are bound in time, the time spent to circumvent the runtime protections is time that could not be spent investigating the underlying application for security vulnerabilities.

In addition, some tools in Section 3.1 are not compatible with all the features of the other tools. For example, LSPosed cannot be used in conjunction with Magisk deny list. Therefore, even for applications with basic runtime protections, it may not always be possible to easily manipulate the application while at the same time circumventing its runtime protections.

Such limitations are showing the need for a more robust solution which can circumvent most runtime protections with minimal effort, while at the same time provide powerful tools to investigate applications on the device.

4 InsecureOS

InsecureOS is a new solution that was crated and developed in this thesis and which aims at overcoming the limitations of the currently available solutions used during security assessments of applications implementing advanced runtime protections (see Section 2.2).

InsecureOS is a solution to root an Android device which is very similar to Magisk, which it is meant to replace. It differs and improves on Magisk in its goal to remain invisible to installed applications.

To achieve this, InsecureOS modifies the Android system as little as possible. For example, unlike Magisk, no new permissive SELinux context is added. Instead, SELinux policies are added to existing contexts as they are required by InsecureOS and the installed modules.

The modifications that must be made to the device are kept to a minimal and are done in a way that allows to modify them to evade detection. In practice, files that are added to directories accessible by applications can be renamed dynamically to prevent applications from verifying their existence.

In addition, InsecureOS comes with a number of default modules which aim at making security research and security assessments more efficient.

4.1 Architecture of InsecureOS

InsecureOS was designed with a modular architecture with support for the dynamic installation and removal of *InsecureOS modules* (see Section 4.1.2).

The main components of InsecureOS are included in *InsecureOS core*, which only provides the main functionalities of InsecureOS which are strictly required for the solution to function properly. Any other features are added through the modules which can be installed either during or after the installation of InsecureOS.

InsecureOS places all required files in the /data/adb/insecureos/ directory, which is a symbolic link to the actual installation directory of InsecureOS that was chosen during installation. In future versions of InsecureOS, the installation directory would typically be the local storage of the InsecureOS manager application.

The /data/adb/insecureos/ directory was chosen to be the main directory used by InsecureOS since it is located in a writable location (the /data partition) while also not being accessible by applications installed on the device. This prevents applications from detecting the presence of InsecureOS by verifying if the directory exists.

4.1.1 InsecureOS core

InsecureOS core contains the main components of InsecureOS. It is located in the /data/adb/insecureos/core directory, and follows the same structure as any other InsecureOS module (see Section 4.1.2). The content of the folder is shown in Figure 4.1.

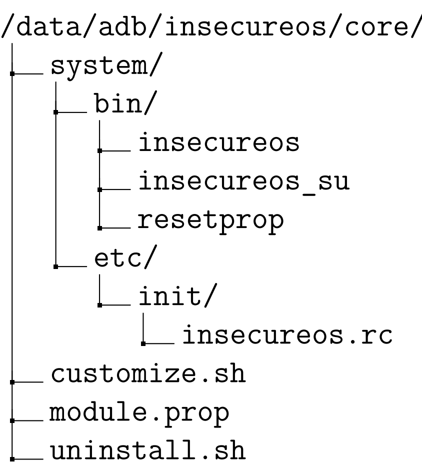


Figure 4.1: Content of the InsecureOS core directory

The file system/bin/insecureos is an executable ELF file that will be copied to /system/bin. It is the main binary of InsecureOS, developed in C++. It can be used to install a new InsecureOS module using the --update argument, and to execute the post-fs-data.sh and boot.sh scripts of installed modules through the --post-fs-data and --boot arguments respectively. It also supports the execution of commands with root privileges through the su argument.

In future versions of InsecureOS, it will be possible to rename the binary through the use of a configuration file. This would prevent applications from verifying if

InsecureOS is installed by verifying if the `insecureos` binary exists. However, this will also have the side effect of making it more difficult for installed InsecureOS modules to access the binary. For this reason, a symbolic link of the binary is created at `/data/adb/insecureos/insecureos`. Modules will then always be able to access the binary through its symbolic link.

The file `system/bin/insecureos_su` is an executable ELF file that will also be copied to `/system/bin`. It is the `su` binary from *LineageOS* [45], an open-source Android distribution. It is used to provide the ability to the user to obtain root privileges on the device which is not the case by default in Android (see Section 2.1.3). In future versions of InsecureOS, `su` will be merged into the main binary of InsecureOS.

The binary was named `insecureos_su` instead of `su` in order to prevent applications from detecting the rooted nature of the device by verifying if the `su` binary exists. This will no longer be necessary once it is merged into the main `insecureos` binary.

Like the two previous files, the file `system/bin/resetprop` is an executable ELF file that will be copied to `/system/bin`. It is the `resetprop` binary from Magisk [46] which allows for the modification system properties on the device, even those in protected namespaces (see Section 2.1.2.2). Once again, in future versions of InsecureOS, `resetprop` will be merged into the main binary of InsecureOS to prevent applications from verifying if it exists.

The file `system/etc/init/insecureos.rc` is a .rc file that will be copied to `/system/etc/init`. The file describes several actions that will be executed by `init` (see Section 2.1.2.1) during the boot process of the device.

In particular, it defines the `su_daemon` system service, which is used by `insecureos_su` to provide access to the root user. The .rc file also executes the main InsecureOS binary with the `--post-fs-data` and `--boot` arguments when the relevant boot stage is reached during the boot process of the device (see Section 2.1.2.1).

The `customize.sh` file is a shell script that is executed during the installation of InsecureOS. It is used in the InsecureOS core module to create the symbolic link of the main InsecureOS binary, and to add the following two SELinux policies which are required for `insecureos_su`:

```
allow shell socket_device:sock_file write
allow shell devpts:chr_file ioctl
```

The remaining files will be discussed in Section 4.1.2.

4.1.2 InsecureOS modules

InsecureOS modules are located in the `/data/adb/insecureos/modules` directory. Each folder in the directory represents a different installed module, and has a strict structure as shown in Figure 4.2.

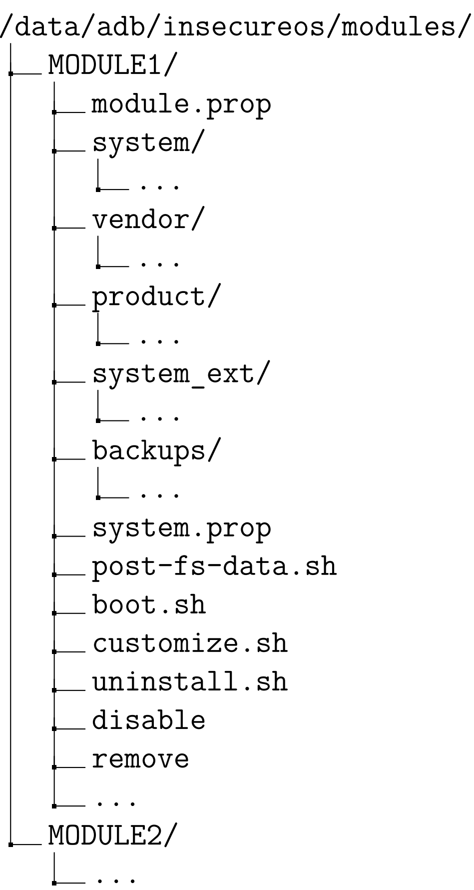


Figure 4.2: InsecureOS modules directory structure

The file `module.prop` is the only strictly required file in a module. It contains information about the module such as its unique identifier, its name, its author information, its version number, ... The file **must** follow the format shown in Figure 4.3.

The `system`, `vendor`, `product` and `system_ext` folders are optional folders. If any of them exists, their content will be copied to the `/system`, `/vendor`, `/product` and `/system_ext` partitions respectively by InsecureOS during the installation of the module. These folders can be used by the module to add files to the system.

```
id=<string>
name=<string>
version=<string>
versionCode=<int>
author=<string>
description=<string>
```

Figure 4.3: `module.prop` file format

For example, a module can register a new system service by defining a .rc file in the `system/etc/init` folder.

Already existing files that would be overwritten through this process will be copied to the `backups` folder in the module directory. During the uninstallation of the module (i.e. when it is disabled or removed), files in the `backups` folder will be restored.

The `system.prop` file is an optional file in which a module can specify values of system properties (see Section 2.1.2.2), which will be set by InsecureOS during the startup of the device. Each line of the file must be of the form `system.property.name=value`.

The `post-fs-data.sh` and `boot.sh` files are optional shell scripts which will be executed by InsecureOS during the startup of the device. They will be executed during the `post-fs-data` and `boot` stages of the boot process respectively (see Section 2.1.2.1). They can be used by the module to execute specific actions at different stages of the boot process.

The `customize.sh` and `uninstall.sh` files are optional shell scripts which will be executed by InsecureOS at the end of the installation and uninstallation of the module respectively. The `customize.sh` script can be used to perform additional modifications to the system if needed, while `uninstall.sh` will be responsible to revert such changes when the module is disabled or removed.

A module can also add new SELinux policies in the `customize.sh` script, using the `inject_selinux_policy` shell function. Note that added SELinux policies will be automatically removed when uninstalling the module. `uninstall.sh` should therefore **not** remove SELinux policies that were added during the installation of the module.

The `disable` and `remove` files are optional files. If they exist, the module will be disabled or removed respectively during an update of InsecureOS. These files should be managed by InsecureOS and not by the module itself.

Finally, modules can also include any number of additional files and folders that they can use to work correctly.

4.2 Installation of InsecureOS

To install InsecureOS on a device, the InsecureOS update package (see Section 4.2.1) must be applied through the Android recovery of the device (see Section 2.1.1.3).

Since the InsecureOS update package is not signed, it is required to use a third-party recovery such as TWRP [47] to install it. Therefore, the bootloader of the device must be unlocked (see Section 2.1.3.4). The process of unlocking the bootloader and installing a third-party recovery is device-specific and will therefore not be discussed in this thesis.

In addition, since installing InsecureOS will modify system partitions, Android Verified Boot (AVB) (see Section 2.1.3.1) must be fully disabled. This will be the case on third-party Android ROMs such as LineageOS [48], which was used in this thesis.

4.2.1 The InsecureOS update package

The structure of the InsecureOS update package is shown in Figure 4.4 and follows the structure of Android Over-the-Air (OTA) update packages [15].

The `core` folder contains InsecureOS core which will be copied to `/data/adb/insecureos/core`, while the `modules` folder contains all the modules to install and which will be copied to `/data/adb/insecureos/modules`. When installing InsecureOS, the `core` folder must be present and contain a valid `module.prop` file, while the `modules` folder is optional. If the `modules` folder is present, all the modules in it must have a valid `module.prop` file.

The `META-INF` folder contains files that are required for the Android recovery to recognize the ZIP file as an update package [49]. Normally, the `update-binary` file is a binary file that is executed by the Android recovery, while the `updater-script` file is a Edify script [50] that will apply the update. In practice however, `update-binary` can be replaced by a shell script, while `updater-script` can be left empty, which was done in InsecureOS.

In the InsecureOS update package, `update-binary` is responsible for extracting the different files from the update package and placing them in `/tmp/INSTALLDIR`. It will then source several additional shell scripts included in the `utils` folder, before sourcing the `insecureos.sh` shell script where the installation will continue.

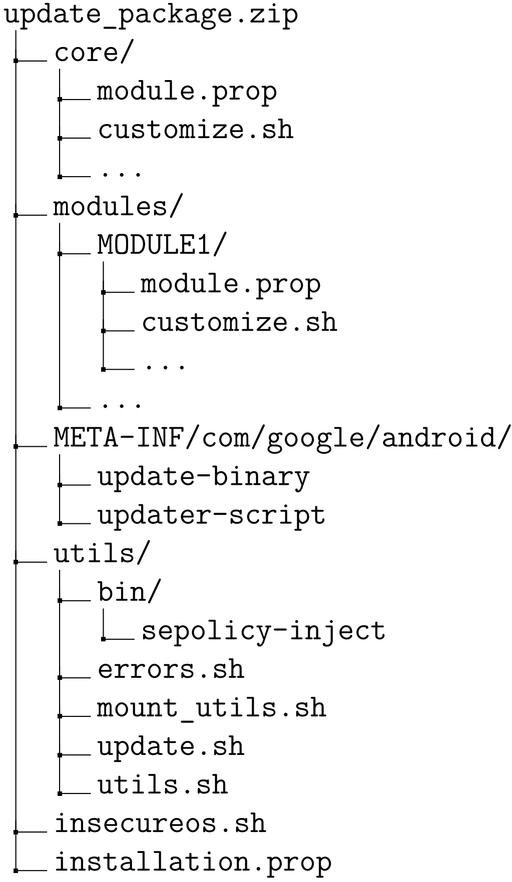


Figure 4.4: InsecureOS update package structure

The `utils` folder contains several shell scripts and binaries which provide helper functions that are used in `insecureos.sh` and that the different modules can use in their `customize.sh` shell script.

The `insecureos.sh` file is a shell script sourced from `update-binary` which contains all the installation, update and removal logic of InsecureOS. The different flows will be detailed in Sections 4.2.2 and 4.3

Finally, the `installation.prop` file contains parameters that are used during the installation process. Currently, it only contains the installation path of InsecureOS, with its default value being `/data/insecureos`. In future versions of InsecureOS, the installation path will be set to the local storage of the InsecureOS manager application.

4.2.2 The installation flow

When applying the update through the Android recovery, execution will start in the `META-INF/com/google/android/update-binary` shell script.

The script will start by defining the `INSTALLDIR` shell variable with the value `/tmp/INSTALLDIR`. This is where all the files will be extracted to, and where modules can put temporary files. The script will then extract and source the files in the `utils` folder before mounting the `/system`, `/vendor`, `/product`, `/system_ext` and `/data` partitions. If the partitions cannot be mounted in read-write mode, the update will be aborted.

Finally, it will extract the content of the `core` and `modules` folders to `INSTALLDIR` and source the `insecureos.sh` shell script where the execution will continue.

The `insecureos.sh` script will first verify if InsecureOS is already installed by verifying if the `/data/adb/insecureos` directory already exists on the device. If it does, it will update InsecureOS (see Section 4.3), otherwise it will install it.

When installing InsecureOS, the script will verify that the `core` folder exists in the update package and that it is a valid InsecureOS module (i.e. that its `module.prop` file has the correct format). If it is not the case, installation will be aborted.

The script will then obtain the installation path from the `installation.prop` file, create the directory in the system, and create a symbolic link at `/data/adb/insecureos`. All subsequent access to the installation directory, be it from the installer script or by modules, should use the symbolic link instead of the true installation path.

The `core` and `modules` folders will be copied to the InsecureOS directory (`/data/adb/insecureos`), then the SELinux context of all the files within it will be set to `u:object_r:system_file:s0`.

Finally, InsecureOS core will be installed using the module installation flow (see Section 4.2.2.1), after which every module in the InsecureOS modules directory will also be installed using the same flow.

Figure 4.5 depicts the complete InsecureOS installation flow.

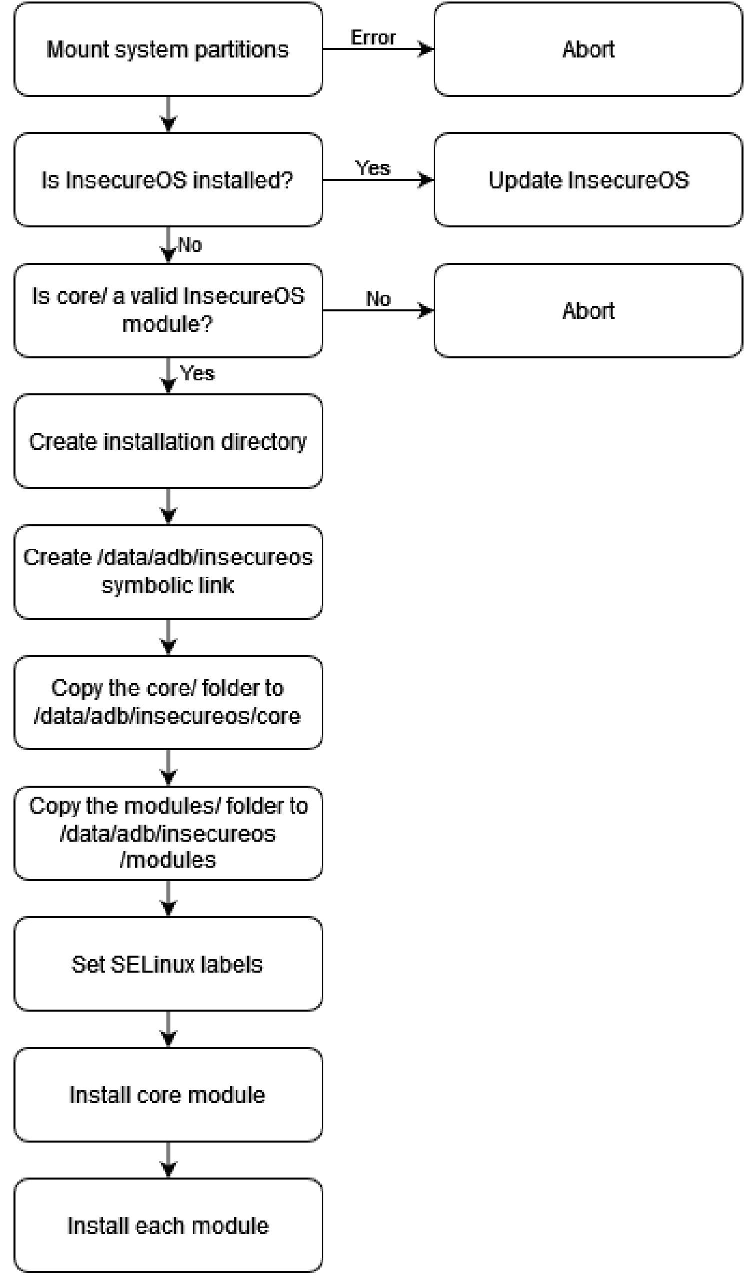


Figure 4.5: The InsecureOS installation process

4.2.2.1 InsecureOS module installation flow

When installing a module, the installer will first verify if the `disable` file exists in the module directory. If it exists, it will abort the module installation.

After this, the installer will iterate over the `system`, `product`, `vendor` and `system_ext` folders in the module directory. If the folder exists, the permissions of all its files and directories will be set to `644`, while the permissions of the files in its `bin` folder will be set to `755`. Each file and folder will then be copied to the corresponding system partitions (e.g. files in the `system` folder will be copied to the `/system` partition).

If the destination of the file already exists in the partition, the `backups` folder will be created in the module directory, and the original file will be copied in it.

Finally, the `customize.sh` script of the module will be executed if it exists. In the script, modules can use the `INSECUREOSDIR`, `MODULEDIR` and `INSTALLDIR` shell variables, which respectively represent the `/data/adb/insecureos` folder, the module directory, and the temporary installation directory.

Modules can also use all the helper functions defined in the `utils/utils.sh` shell script, including the `inject_selinux_policy` functions which allows to define new SELinux policies.

Figure 4.6 depicts the complete InsecureOS module installation flow.

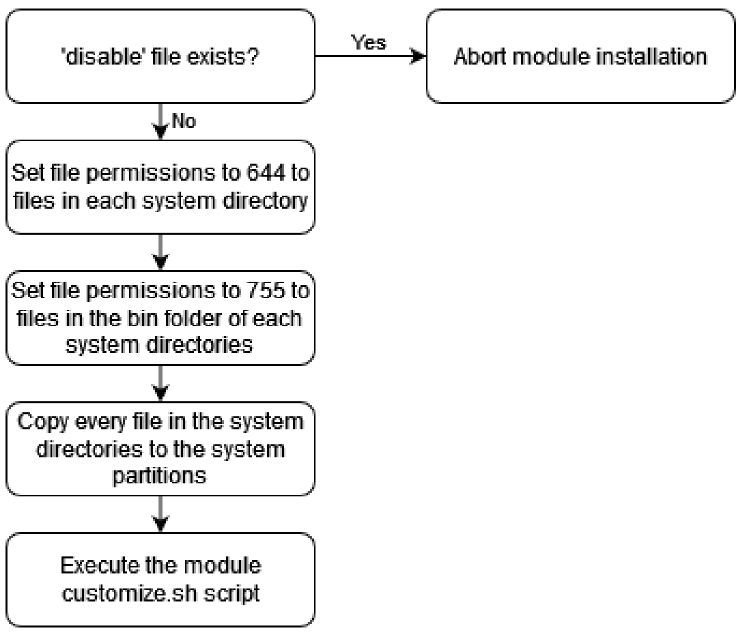


Figure 4.6: The InsecureOS module installation process

4.3 Update of InsecureOS

An InsecureOS update will be triggered every time a module is installed, enabled or disabled, or when InsecureOS itself is updated. It can be started in two different ways. The first is to apply the InsecureOS update package from the Android recovery. The second is to execute the `insecureos` binary with the `--update` argument and by providing the path to an InsecureOS update package (see Section 4.2.1).

In the first case, the Android recovery will start by executing the `update-binary` file, while in the second, the `insecureos` binary will start by executing the `utils/update.sh` file of the provided update package. However, in both cases, the `insecureos.sh` shell script will be executed after the system partitions are mounted in read-write mode.

The first step of the InsecureOS update process is to fully disable InsecureOS. It will start by restoring the original SELinux policies of the device, before disabling every installed InsecureOS module, and finally disable the InsecureOS core module. This step is required to ensure that SELinux policies added by disabled modules are correctly removed from the device.

When disabling an InsecureOS module, the files and folders it added to the system partitions are removed, while their original ones are restored. The `backups` folder in the module directory will then be removed. Finally, the `uninstall.sh` script of the module will be executed.

The InsecureOS updater will then verify if the core module needs to be updated. It will first verify if the `core` folder exists in the update package. If it does, and if its version code is greater than the currently installed core module, the content of the `core` folder of the update package will replace the current `core` folder in the InsecureOS directory.

Regardless if the core module was updated or not, the updater will then install the core module, following the flow discussed in Section 4.2.2.1.

The updater will then update every module that is included in the `modules` folder of the update package. If the module is already installed and if the new version code is greater, the content of the module folder in the update package will replace the existing module folder in the InsecureOS directory. However, if the module is not already installed, the module folder will simply be copied to the InsecureOS modules directory of InsecureOS.

Finally, the updater will iterate over all the modules in the InsecureOS `modules` directory and will install each of them following the flow discussed in Section 4.2.2.1.

Figure 4.7 shows the complete InsecureOS update process as explained above.

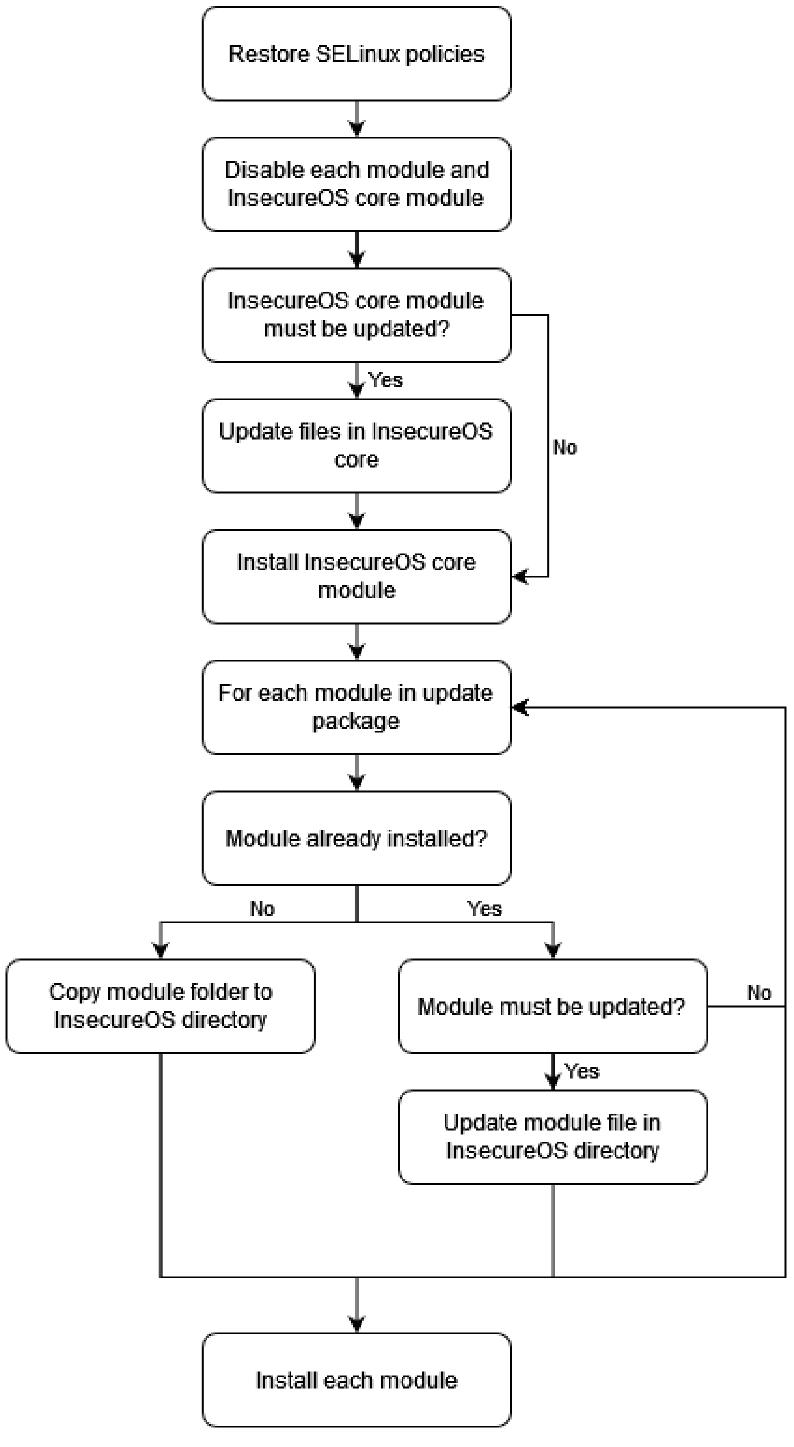


Figure 4.7: The InsecureOS update process

4.4 The InsecureOS boot process

InsecureOS will perform a number of actions during the startup of the device to ensure that all components and modules are properly applied to the system.

Most of these actions are defined in the `insecureos.rc` file which is included in InsecureOS core.

During the boot process of the Android device, `init` (see Section 2.1.2.1) will load the SELinux policies defined on the device. These policies are included in the `/vendor/etc/selinux/precompiled_sepolicy` file, in which InsecureOS injected the SELinux policies added by the different modules. All the SELinux policies added by the modules will therefore be loaded in the system by `init`.

When reaching the `post-fs-data` boot stage (see Section 2.1.2.1), `init` will trigger the `post-fs-data` action defined in `insecureos.rc`. This action will first execute the InsecureOS binary with the `--post-fs-data` argument, with root privileges and with the `u:r:su:s0` SELinux context. It will then start the `su_daemon`, which executes the `insecureos_su` binary with the `--daemon` argument, with root privileges and with the `u:r:su:s0` SELinux context.

When executing the InsecureOS binary with the `--post-fs-data` argument, the binary will iterate over all the installed InsecureOS modules, modify the system properties defined by the module in their `system.prop` file and execute their `post-fs-data.sh` script if it exists and the module is not disabled.

When reaching the `boot` stage of the boot process, `init` will trigger the `boot` action defined in `insecureos.rc`. This action will execute the InsecureOS binary with the `--boot` argument, with root privileges and with the `u:r:su:s0` SELinux context.

When executing the InsecureOS binary with the `--boot` argument, the binary will iterate over all the installed InsecureOS modules, and execute their `boot.sh` script if it exists and the module is not disabled.

4.5 Default InsecureOS modules

A number of InsecureOS modules were developed during this thesis and are installed by default when installing InsecureOS. They aim at making security assessments of Android applications easier.

4.5.1 Frida-server

The *Frida-server* InsecureOS module makes it possible to use Frida (see Section 3.1.2) during a security assessment.

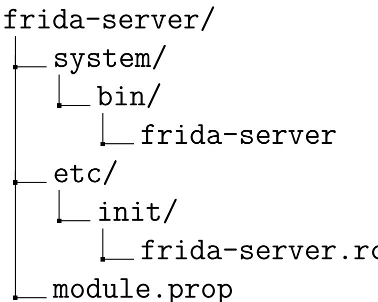


Figure 4.8: Frida-server module directory

The `frida-server.rc` file contains the definition of the `frida-server` system service. During the `early-boot` stage of the Android boot process (see Section 2.1.2.1), the service will be started and will run the `frida-server` binary, with root privileges and with the `u:r:su:s0` SELinux context.

The `frida-server` binary is a component of Frida that must be run on the Android device. When injecting a script with Frida, the Frida client will send the script to the `frida-server` running on the device, which will then inject the JavaScript runtime into the targeted application and run the script in its context.

4.5.2 Frida-inject

The *Frida-inject* InsecureOS module allows other InsecureOS modules to inject Frida scripts (see Section 3.1.2) into `system_server` (see Section 2.1.2.3) during the device startup.

The `customize.sh` shell script is only used to make the `frida-inject` binary executable. The `boot.sh` shell script will iterate over all the InsecureOS modules

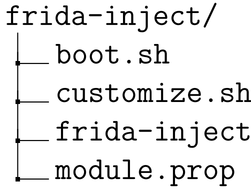


Figure 4.9: Frida-inject module directory

and attempt to inject all JavaScript files that are included in the `frida-inject` folder of the module.

The `frida-inject` binary is part of Frida, and is a standalone version of Frida that is meant to be executed on the device. `frida-inject` assumes the role of both the Frida client and `frida-server` to inject the JavaScript into the targeted application.

Other InsecureOS module can then leverage the Frida-inject module by creating Frida scripts and placing them in the `frida-inject` folder in their module directory.

4.5.3 Move Certificate

The *Move Certificate* InsecureOS module allows to use user installed certificates to validate the server certificate during a TLS connection, even if the application is not explicitly configured to do so (see Section 2.1.3.3).

It achieves this by moving any TLS certificate installed into the user Certificate Authority (CA) store into the system CA store.

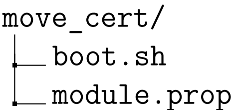


Figure 4.10: Move Certificate module directory

The `boot.sh` script will attempt to remount the `/system` partition in read-write mode, move the certificates from the user CA store (located in the `/data` partition) to the system CA store (located in the `/system` partition), and finally remount the `/system` partition in read-only mode.

4.5.4 MagiskLess-Riru

The `MagiskLess-Riru` InsecureOS module allows to inject arbitrary shared libraries into the process of any application by using MagiskLess-Riru [51], a port of the

Riru Magisk module (see Section 3.1.4).

MagiskLess-Riru was created during this thesis in order to make Riru compatible with InsecureOS and be able to inject libraries into the process of applications without Magisk (see Section 3.1.1) being installed on the device.

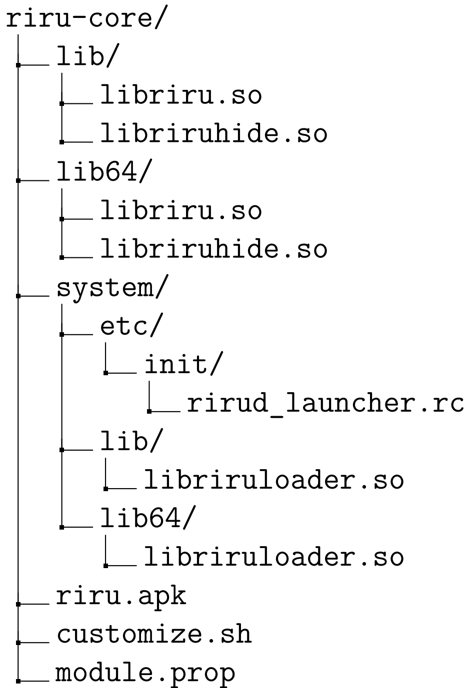


Figure 4.11: MagiskLess-Riru module directory

The `libriruloader.so` files are shared libraries which are part of MagiskLess-Riru and which will be loaded into the 32 and 64 bits versions of Zygote (see Section 2.1.2.3) during the startup of the device. Its goal is to load the more complete `libriru.so` and `libriruhide.so` libraries which will be responsible for communicating with the Riru daemon and load arbitrary shared libraries of other InsecureOS modules when a new application is forked from Zygote (see Section 2.1.2.3).

The `rirud_launcher.rc` file contains the definition of the `rirud_service` system service which will start the Riru daemon during the `post-fs-data` stage of the boot process (see Section 2.1.2.1). The file will also set the value of the `ro.dalvik.vm.native.bridge` system property to `libriruloader.so` in order for Zygote to load the library when it is started.

The Riru daemon is started with root privileges and the `u:r:su:s0` SELinux

context. It is started by executing the `riru.Daemon` class from the `rirud.apk` file, using the Android `app_process` binary, which allows to execute arbitrary Java classes from APK files.

Finally, the `customize.sh` script is used to inject the following SELinux policy during the installation of the module:

```
allow zygote adb_data_file:dir search
```

When Zygote is started and has correctly loaded MagiskLess-Riru, MagiskLess-Riru will iterate over all installed InsecureOS modules, and load any shared libraries located in the module directory under the folders `riru/lib/` and `riru/lib64/`. The loaded libraries will then be triggered when a new application is spawned from Zygote, allowing the libraries to be executed in the environment of the new application.

Other InsecureOS modules can therefore leverage the MagiskLess-Riru module to inject arbitrary shared libraries into all the applications on the device, simply by placing the shared libraries in the `riru/lib/` and `riru/lib64/` folder in their directory. Such InsecureOS modules are called `Riru modules`.

4.5.5 Universal SafetyNet Fix

The *Universal SafetyNet Fix* InsecureOS module is a Riru module (see Section 4.5.4) which is responsible for ensuring that Google SafetyNet (see Section 2.2.2) does not detect that the device has been modified, using the Universal SafetyNet Fix Magisk module [32].

It allows applications relying on Google SafetyNet to verify the integrity of the device to be run on modified devices.

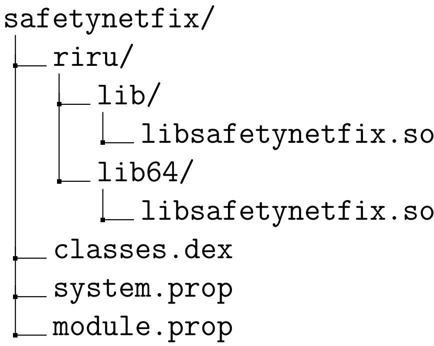


Figure 4.12: Universal SafetyNet Fix module directory

The `libsafetynetfix.so` file will be loaded by MagiskLess-Riru when Zygote is started. When a new application is spawned, the library will verify if the newly spawned application is the one responsible for Google SafetyNet, otherwise the library will stop its execution. If the application is the one responsible for Google SafetyNet, the library will load and execute the `classes.dex` file.

The `classes.dex` file contains Java classes which are responsible for disabling the hardware-backed key attestation enforced by Google SafetyNet on some devices (see Section 2.2.2). It is achieved by modifying the build model name of the device (`android.os.Build.MODEL`) returned to SafetyNet, which is used to determine if the device supports hardware-backed key attestation. In addition, a new empty Java security provider is created and replaces the default security provider in order to prevent the application from obtaining the key used in the attestation.

These two steps allow to completely circumvent the hardware-backed key attestation performed by SafetyNet by making it believe that the device does not support it.

In addition, a number of system properties are changed using the `system.prop` module file, such as the `ro.boot.flash.locked` property which indicates whether the bootloader of the device is locked. Each of these system properties are used by Google SafetyNet to verify the state of the device.

4.5.6 MagiskLess-LSPosed

The `MagiskLess-LSPosed` InsecureOS module is a Riru module (see Section 4.5.4) which allows to hook arbitrary methods of the Android framework and any application by using MagiskLess-LSposed [52], a port of the LSPosed Magisk module (see Section 3.1.5).

MagiskLess-LSPosed was created during this thesis in order to make LSPosed compatible with InsecureOS and be able to hook applications without Magisk (see Section 3.1.1) being installed on the device.

The `liblspd.so` files will be loaded by MagiskLess-Riru when Zygote is started. They are responsible for initializing the Xposed framework (see Section 3.1.5) by loading the `lspd.dex`, and to perform the actual hooking when an LSPosed module attempts to hook a method from another application. This is achieved by communicating with the LSPosed daemon from the module.

The `dex2oat32` and `dex2oat64` are binary files that will replace the Android Runtime `dex2oat` binaries, which are part of Android ART (see Section 2.1) and is responsible for compiling the APK files on the system [53].

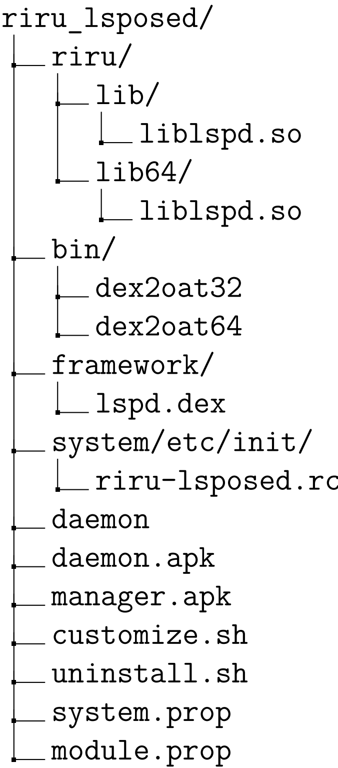


Figure 4.13: MagiskLess-LSPosed module directory

The LSPosed `dex2oat` binaries are used to monitor the parsed Java method from the compiled APKs and register the hooks during the runtime of the applications.

The `lspd.dex` file contains the Xposed framework which is accessible to LSPosed modules. It is loaded into Zygote by the `liblspd.so` library during the device startup in order to be made available to all applications.

The `rirud-lsposed.rc` file defines the `riru-lsposed` system service. It is started during the `post-fs-data` stage of the boot process with root privileges and with the `u:r:su:s0` SELinux context.

The service will execute the `daemon` shell script, which will in turn execute the `daemon.apk` file using the Android `app_process` binary. The APK file contains the LSPosed daemon, which is used to bridge all the components used by LSPosed. It is also responsible for notifying the different components when an LSPosed module is installed and making sure it is a valid module.

Finally, the `manager.apk` file is the LSPosed manager Android application which is installed after the installation of MagiskLess-LSPosed. It is used to manage the installed LSPosed modules and control which applications the modules are allowed to hook.

The MagiskLess-LSPosed module requires a number of SELinux policies to be added on the device to work properly. All the policies are therefore added during the installation of the module by the `customize.sh` script.

LSPosed allows installed Android applications to use the Xposed framework to hook into other applications. Such applications, called LSPosed modules, must follow the structure of regular LSPosed modules. In particular, they must declare the `xposedmodule` metadata in their Android manifest file and define the `xposed_init` file containing the entry point that should be executed by LSPosed.

LSPosed modules are therefore specialized Android applications with a specific structure which can use the Xposed framework to hook methods of other applications.

4.5.7 Logger

The *Logger* InsecureOS module is responsible for installing the `Logger` LSPosed module. The module was developed during this thesis and will log various information about the execution of the Android applications on the device to help understand the behavior of the applications.

The module will achieve this by using MagiskLess-LSPosed to hook several key methods in the Android framework and log information about their execution.

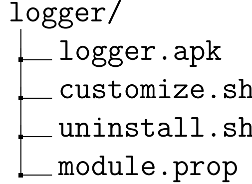


Figure 4.14: Logger module directory

The `customize.sh` and `uninstall.sh` scripts are responsible for installing and uninstalling the `logger.apk` application using the Android Activity Manager system service, which is responsible for managing installed applications.

The `loggger.apk` file contains the Logger LSPosed module which is responsible for hooking the various methods in the Android framework.

Currently, it writes all the logs into Android Logcat, the Android logging service. In addition, it currently only logs the execution of methods related to the Android Network Security Config SSL Pinning and the OkHttp3 SSL Pinning.

In practice, it hooks the `android.security.net.config.NetworkSecurityConfig.getPins()` and `android.security.net.config.XmlConfigSource.getPerDomainConfigs()` methods to log the domains which are pinned using the Network Security Config. It also hooks the `okhttp3.CertificatePinner$Builder.add()` method to log the domains which are pinned using OkHttp3.

4.5.8 SSL Pinning Bypass

The *SSL Pinning Bypass* InsecureOS module is responsible for installing the `SSL Pinning Bypass` LSPosed module. The module was also developed during this thesis and will circumvent several implementations of SSL Pinning (see Section 2.2.1). It allows to intercept the communications of the application implementing SSL Pinning without the need to use tools such as Frida and Objection.

The module will achieve this by using MagiskLess-LSPosed to hook several key methods in the Android framework and third-party libraries and modify their behavior.

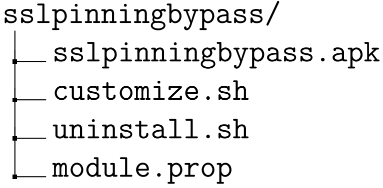


Figure 4.15: SSL Pinning Bypass module directory

The `customize.sh` and `uninstall.sh` scripts are responsible for installing and uninstalling the `sslpinningbypass.apk` application using the Android Activity Manager system service.

The `sslpinningbypass.apk` file contains the SSL Pinning Bypass LSPosed module, which is responsible to hook the methods in the Android framework and third-party libraries.

Currently, it will circumvent the Android Network Security Config SSL Pinning and the OkHttp3 SSL Pinning.

In practice, it hooks the `android.security.net.config.NetworkSecurityConfig.getPins()` and `android.security.net.config.XmlConfigSource.getPe-`

`rDomainConfigs()` methods and return an empty list of pins in order to circumvent the Network Security Config SSL Pinning.

It also hooks the `okhttp3.CertificatePinner.findMatchingPins()` method to circumvent OkHttp3 SSL pinning.

4.6 Improvements over existing solutions

The existing solutions to test applications and circumvent their runtime protections have several drawbacks, which can be summarized as follows:

- Magisk, which is used during all security assessments, is well-known and specifically targeted by advanced runtime protections. When using Magisk, many modifications are made to the device which can be used by the runtime protections to determine if it is installed on the device, even with Magisk deny list being enforced.
- Some of the tools, such as LSPosed, cannot be used together with Magisk deny list. It is therefore not possible to use these tools to test applications that require the use of Magisk deny list.
- Magisk can be used to circumvent the root detection of the applications but it cannot be used to circumvent other runtime protections such as SSL pinning or debugger detections. This requires the use of a debugger or runtime manipulation tool such as Frida or Objection, which will not be usable on applications implementing debugger detection without first circumventing it. LSPosed modules could be developed to circumvent the runtime protections, however it is not an ideal solution due to the previous issue.

InsecureOS provides improvements over the existing solutions with regards to all of the above issues.

InsecureOS being a rooting solution similar to Magisk, the use of Magisk will no longer be required during security assessments. Runtime protections targeting Magisk specifically will no longer be able to detect the rooted nature of the device, as InsecureOS was designed to evade such detection.

In particular, InsecureOS does not provide the default `su` binary and does not inject new permissive SELinux context on the device. In addition, no temporary file system are created, and no static directory used by InsecureOS can be accessed by installed applications.

As such, the different ways to detect if Magisk is installed on the device will not be able to detect the presence of InsecureOS. This effectively circumvents most root detection mechanisms by default.

In addition, the Universal SafetyNet Fix module modifies several system properties to circumvent Google SafetyNet, which further hides the rooted nature of the device, allowing all tested root detection methods to be circumvented.

Finally, it is possible to dynamically rename the few files which can be accessed by installed applications, such as `/system/bin/insecureos` or `/system/bin/resetprop`. By renaming the files, it would prevent applications from verifying if the file exists, as it has a different name. This will help further improve the root detection evasion, even if runtime protections are targeting InsecureOS specifically.

Section 4.6.1 further explains the differences between InsecureOS and Magisk, and how InsecureOS manages to remain undetected to applications that detect Magisk.

Since InsecureOS circumvents root detection by default, the use of a feature such as Magisk deny list is not required. As such, tools that were incompatible with it can be used in InsecureOS once they are properly ported to a system without Magisk installed.

In particular, Riru and LSPosed can be used in InsecureOS using their port to a system without Magisk, respectively MagiskLess-Riru and MagiskLess-LSPosed.

Finally, since InsecureOS allows the use of tools such as LSPosed while also hiding the rooted nature of the device to the application, it becomes feasible to use LSPosed modules to circumvent other runtime protections implemented by applications, such as SSL Pinning.

The SSL Pinning Bypass module was created to circumvent common SSL Pinning methods used in applications, which allows the interception of HTTPS communications of these applications without requiring the use of a debugger or runtime manipulation tool.

4.6.1 InsecureOS vs Magisk

InsecureOS was developed to provide the same features as Magisk and to replace it. However, InsecureOS differs from Magisk in the way it provides its features, which allows InsecureOS to remain invisible to installed applications.

In this section, we will discuss the differences between InsecureOS and Magisk, and in particular why InsecureOS manages to remain undetected when Magisk does not.

When installing either InsecureOS or Magisk on an Android device, a number of modifications are made to the system in order to provide root access to the user. Table 4.1 lists the modifications that are made to the device by both solutions. It should be noted that the table includes all the modifications made by InsecureOS

System modifications	InsecureOS	Magisk
Added files	/system/bin/insecureos /system/bin/insecureos_su /system/bin/resetprop /system/etc/init/insecureos.rc	/sbin/magisk /sbin/magisk32 /sbin/magisk64 /sbin/magiskpolicy /sbin/su /sbin/resetprop
Running processes	insecureos_su daemon	su daemon magiskd busybox
Added SELinux domains	/	magisk magisk_client magisk_file magisk_exec
Installed applications	InsecureOS manager*	Magisk manager

* Will be included in future version of InsecureOS

Table 4.1: Modifications made by InsecureOS vs Magisk to the system, which applications can detect

that can be detected by installed applications, excluding the ones made by the default modules, while it does not list the complete list of modifications made by Magisk.

InsecureOS will provide its features by modifying the system partitions of the Android device directly, and adding the necessary files to the system. For example, it will add the `insecureos.rc` file and the `insecureos_su` binary to the system to provide root access to the user.

InsecureOS will also add new SELinux policies by directly adding the policies to the `precompiled_sepolicy` file which will be loaded by the system.

Magisk however works in a very different way. Instead of modifying the system partitions directly, Magisk will modify the boot image and modify the `init` binary (`magiskinit`). `magiskinit` will then be used to mount the system partitions and overlay several Magisk directories on top of it using *bind mounts*. The Magisk files will therefore be accessible on the system partitions through these bind mounts, without ever modifying the system partitions directly. `magiskinit` will also load the SELinux policies differently. Instead of only loading the `precompiled_sepolicy` file, it will also load additional sepolicy files that are provided by Magisk.

All this makes it possible for Magisk to modify the device and provide its feature without the need of modifying the system partitions.

The modifications to the device made by InsecureOS and which are presented in Table 4.1 can all be detected by installed applications. However, InsecureOS provides features that will prevent the applications from detecting them. InsecureOS allows to easily rename and change the configuration of the files and daemons it uses, which prevents applications from detecting their presence.

For example, InsecureOS can hide the files added to the system by allowing them to be renamed easily, which would prevent applications from looking for the file using its name. For the running processes, InsecureOS will use the same idea. The `insecureos_su` daemon will be configurable to change its name and socket in order to prevent applications from using those information to detect the daemon.

As such, while installed applications can technically verify if the files and daemons used by InsecureOS exist on the device, it will not be a reliable method to detect InsecureOS as it will be possible to change their names and configuration easily.

Magisk, on the other hand, will do modifications to the system which are more challenging to hide to installed applications. For example, it adds several SELinux contexts (see Figure 4.1) on the system, relies on a number of daemons, etc.

To hide those modifications to installed applications, Magisk will inject itself into the process of the running applications and attempt to hide the modifications to the processes directly.

For example, when a new application is spawned, and if it is in the Magisk deny list, Magisk will unmount the files in the application process, which would effectively remove those files for the application and prevent it from detecting the files.

While Magisk is able to hide the files it added to the Android file system using this method, it also has to hide all the other modifications, such as the added SELinux contexts, the running daemons and the fact that it is injected into the application process.

In practice, Magisk is able to hide its presence to most applications that implement runtime protections. However, some applications implementing advanced runtime protections which target Magisk specifically are still able to detect that it is installed, despite the attempt of Magisk to hide itself.

5 InsecureOS in practice

As InsecureOS is meant to replace Magisk and be used during security research and security assessments, it is important to verify that the solutions does bring improvements over Magisk.

In this section, several tests will be discussed that aim at evaluating if InsecureOS could indeed be used for such security assessments, and if it does improve on existing solutions by remaining undetected by more applications.

To achieve this, the following two tests have been performed according to the test setup discussed in Section 5.1:

- **Root detection test**: Several applications that implement some form of root detection will be run and monitored to see if Magisk and InsecureOS are able to circumvent the root detection of the application by default.
- **Intercepting communications**: The communications of several applications will be intercepted to see if typical installations of Magisk and InsecureOS can be used to intercept communications by default.

The results of those tests are discussed in Section 5.2.

5.1 Setup of the tests

In order for the results to be meaningful, all the tests were performed on the same device, with the same version of Android, and using the same test applications.

The device used was a *Google Pixel 3a XL* device, running LineageOS 11, with the exact build number being `lineage_bonito-userdebug 11 RQ3A.211001.001 10037326`.

It should be noted that since the device was running LineageOS, the bootloader of the device was *unlocked* (see Section 2.1.3.4) for all the tests.

The applications that were tested are shown in Table 5.1. All the tested applications are accessible from the Google Play Store, with the exception of Itsme E2E which can be downloaded from the Microsoft App Center. These applications were chosen as they implement some form of runtime protections.

Application	Package	Version
SafetyNet Test	org.freeandroidtools.safetynettest	1.2.1
Root Checker	com.joeykrim.rootcheck	6.5.3
Advanced Root Checker	com.anu.developers3k.rootchecker	1.8.0
Itsme	be.bmid.itsme	3.11.0
Itsme E2E	be.bmid.itsme.e2e	3.11.0
Payconiq by Bancontact	mobi.inthepocket.bcmc.bancontact	7.15.0.416735
Auth-ES	com.DPMOBESA	4.28.7

Table 5.1: Tested applications

All the tests were performed on all the applications from Table 5.1 and on the following four different configurations of the device:

1. **LineageOS**: LineageOS 11 was installed on the device without any additional modifications. InsecureOS and Magisk were not installed.

2. **Magisk**: Magisk was installed on the device, but Zygisk and Magisk deny list were not enabled. No additional Magisk modules were installed.

3. **Magisk (Hidden)**: Magisk was installed on the device, Zygisk was enabled, and the deny list was enforced for the tested applications. In addition, the `Universal SafetyNet Fix` Magisk module was installed.

4. **InsecureOS**: InsecureOS was installed on the device. All of the default InsecureOS modules were enabled.

In all of the device configuration, a TLS certificate was also installed in the system CA store, with the exception of the default LienageOS installation where it was installed in the user CA store.

5.2 Evaluation of InsecureOS

InsecureOS and Magisk will both be evaluated based on the results on the following two tests:

- Circumventing root detection
- Intercepting communications

The results of the two solutions will then be compared in order to evaluate if InsecureOS offers more convenience than Magisk during security assessments.

5.2.1 Root detection

All the applications from Table 5.1 were run on the device in the various configurations mentioned in Section 5.1. The behavior of the applications was then monitored to see if the applications were able to detect that they were running on a modified or rooted device and prevented to be run on the device.

Table 5.2 indicates whether the applications were able to detect that the device was modified and/or rooted. For the SafetyNet Test, Root Checker and Advanced Root Checker applications, *Pass* indicates that the application did not detect that the device was modified and/or rooted, while *Fail* indicates that the application detected it was.

For the rest of the tested applications, *Pass* indicates that the application could be opened and used on the device as if it was a regular device with the stock firmware, while *Fail* indicates that the application detected that the device was modified in some way and prevented its use.

Application	LineageOS	Magisk	Magisk (Hidden)	InsecureOS
SafetyNet Test	**Fail**	**Fail**	**Fail***	**Pass**
Root Checker	**Pass**	**Fail**	**Pass**	**Pass**
Advanced Root Checker	**Fail**	**Fail**	**Pass**	**Pass**
Itsme	**Pass**	**Fail**	**Fail**	**Pass**
Itsme E2E	**Pass**	**Fail**	**Fail**	**Pass**
Payconiq	**Fail**	**Fail**	**Fail**	**Pass**
Auth-ES	**Pass**	**Pass**	**Pass**	**Pass****

* Magisk is able to hide itself to the application, but the application detects that the bootloader of the device is unlocked.

** The Auth-ES application is able to detect the MagiskLess-Riru InsecureOS module. When the module is disabled, Auth-ES was not able to detect InsecureOS.

Table 5.2: Root detection results

As is shown in Table 5.2, when using the default LineageOS installation, some applications were able to detect that the device was modified. The Payconiq application was not usable as it was showing an error page, likely due to the bootloader of the device being unlocked.

When using LineageOS with Magisk installed and without enabling Zygisk or the deny list, all the tested applications were able to detect Magisk, with the exception of Auth-ES. The applications were therefore not usable as they were showing an error message due to the presence of Magisk.

When the Magisk deny list was enforced however, some applications were no longer able to detect Magisk, while some still could. It should be noted that while SafetyNet Test is marked as *Fail*, it is due to the bootloader of the device being unlocked. SafetyNet Test was not able to detect Magisk.

When InsecureOS was installed on the device, no applications were able to detect that the device was modified or rooted. Applications that detected that the device was modified when using the default LineageOS installation were no longer able to detect those modifications.

It should be noted however that while Auth-ES did not detect InsecureOS in itself, it was able to detect the MagiskLess-Riru module. In order to run Auth-ES on the device, it was therefore necessary to disable the module and all the ones that rely on it.

5.2.2 Intercepting the application communications

For the applications from Table 5.1 which are performing network communications, an additional test was performed. During this test, the communications of the application were attempted to be intercepted.

To achieve this, as was mentioned in Section 5.1, a TLS certificate was installed on the system CA store of the device, except for the default LineageOS installation where it was installed in the user CA store.

Table 5.3 indicates whether it was possible to intercept the communications of the tested applications. *Pass* indicates that it was possible to intercept the communications of the application, while *Fail* indicates that it was not possible. *NA* indicates that the application was not running in that configuration and that the communications could therefore not be intercepted.

Application	LineageOS	Magisk	Magisk (Hidden)	InsecureOS
Itsme	**Fail**	NA	NA	**Pass**
Itsme E2E	**Fail**	NA	NA	**Pass**
Payconiq	NA	NA	NA	**Fail**
Auth-ES	**Fail**	**Fail**	**Fail**	**Fail**

Table 5.3: SSL pinning bypass results

When using the default LineageOS installation, the communications of none of the applications could be intercepted. This was expected as the TLS certificate used to intercept the communications was installed in the user CA store instead of the system CA store.

When Magisk was installed on the device, it was not possible to intercept the communications of the applications that could be run. This is due to the applications implementing SSL Pinning.

When using Magisk with Zygisk and enforcing its deny list, it was also not possible to intercept the communications of any application. All the tested applications are implementing SSL Pinning, which would require additional Magisk modules or tools such as Frida to circumvent.

It should be noted that Magisk modules could be installed in order to attempt to bypass the SSL pinning implementation of the application and intercept its communications. However, with the deny list enabled, such modules would not be loaded into the application's process and would therefore not be able to circumvent the SSL pinning implementation. The deny list could be disabled for that

purpose, but this would allow more applications to detect Magisk, as was shown in Section 5.2.1.

For the applications used in this test, even with the installation of a module capable of circumventing SSL pinning, Magisk would not be able to intercept communications of all the application that prevent to be run on the device. For the Auth-ES application, such module would likely be detected, as was the MagiskLess-Riru InsecureOS module detected.

When InsecureOS was installed on the device however, the communications of some applications could be intercepted. This is due to the application using either the Network Security Config or OkHttp3 SSL Pinning implementation, which are circumvented by default in InsecureOS by the SSL Pinning Bypass module (see Section 4.5.8).

The communications of the other applications could not be intercepted by default as those applications used different implementations of SSL Pinning.

5.2.3 Conclusion

Based on the results of the root detection tests shown in Table 5.2, it appears that InsecureOS is able to remain undetected by more applications than Magisk. Using InsecureOS during security assessments would therefore allow to test more applications without the need of first circumventing the root detection of the application.

The results of the communications interceptions tests shown in Table 5.3 show that InsecureOS manages to circumvent the SSL Pinning implementations of some applications while Magisk does not provide such feature by default. The use of InsecureOS during security assessments would therefore allow to intercept the communications of more applications, without the need of circumventing the SSL Pinning implementation of the application.

Overall, based on those results, InsecureOS allows to more easily perform security assessments on applications implementing advanced runtime protections than using the current existing solutions such as Magisk.

6 Going further...

While InsecureOS can be used to circumvent the runtime protections of some applications, it is currently in a proof of concept state that does not provide many configuration options. Using InsecureOS in its current state may be a hassle to the security researcher as enabling and disabling modules will require a few steps from the security researcher.

This section will discuss a number of improvements that should be implemented in order to make InsecureOS a more efficient tool to use during security assessments.

6.1 Core improvements

A number of improvements should be made to the core of InsecureOS and to its different flows.

The first key improvement that can be made to InsecureOS is the development of a manager application. The manager application will allow the user to enable, disable or install modules with a few button presses, making it much easier to further develop InsecureOS and evaluate applications.

The manager application can also include features such as setting a system proxy on the device to more easily configure the device for intercepting the communications of applications.

The second key improvement that can be made is to incorporate all the binaries used by the core module into a single binary. More concretely, the `insecureos_su` and `resetprop` binaries should be incorporated into the `insecureos` binary. This would allow to have only one binary that could potentially be detected by installed applications.

Another improvement that can be made to InsecureOS is to support more device architectures. Currently, InsecureOS only supports the `arm64-v8a` CPU architecture. Android devices using `armeabi-v7a` or `x86` CPU architecture are therefore not supported by InsecureOS.

Finally, it is currently required to disable `dm-verity` (see Section 2.1.3.1) to be able to install InsecureOS. In the future, InsecureOS should be capable of disabling `dm-verity` on the device during its installation when needed.

6.2 Improvements to existing modules

In addition to the improvements that should be made to InsecureOS as a whole, a number of improvements should be made to the modules that were developed during this thesis and that are included by default in InsecureOS.

Currently, the MagiskLess-LSPosed InsecureOS module is derived from the LSPosed Magisk module, which includes an LSPosed manager Android application. MagiskLess-LSPosed therefore also relies on this manager application, which is a different application than the InsecureOS manager application. In future versions of InsecureOS, the MagiskLess-LSPosed module should be modified in order to be able to manage the LSPosed modules from the InsecureOS manager application, without requiring an additional application on the device.

Improvements can also be made to the Logger InsecureOS module, as the module currently only logs information related to SSL Pinning. In future versions of InsecureOS, the module should log more information, such as file system interactions, all network access, cryptographic operations, etc.

Similarly, the SSL Pinning Bypass InsecureOS module currently circumvents only two implementations of SSL Pinning. In the future, the module should be extended to be able to circumvent more implementations of SSL Pinning, even for applications that are heavily obfuscated.

6.3 Modules to implement

Finally, in addition to the improvements to the existing components of InsecureOS, the solution should also be improved through the addition of several modules, that would provide some benefits during security assessments.

An example of such a module would be a module that will make all installed applications use the user CA store instead of the system CA store (see Section 2.1.3.3). This would allow to intercept communications of most applications that do not implement SSL Pinning without requiring to move the user certificates into the system certificates, therefore making it more convenient to add new certificates to the system.

Another example would be a module that would hook applications to prevent them from verifying if the device developer options are enabled as some applications will refuse to run on devices with the developer options enabled. A module preventing the applications to access the state of the developer options would therefore allow more applications to be run on the device without requiring investigation.

A The InsecureOS sources

The InsecureOS sources are located in a private Git repository on GitHub, to which the jury members of this thesis received access.

InsecureOS is a Gradle project that includes several Gradle modules that can all be build using Gradle. The InsecureOS repository is structured as shown in Figure A.1.

The root of the repository contains the general Gradle and Git configuration files such as `gradlew`, `settings.gradle` or `.gitignore` files. The `gradle` folder also contains additional gradle files.

Each folder in the repository, with the exception of the `gradle` folder, contains the sources and build files of the different InsecureOS components.

The `core` folder contains the sources and build files of the InsecureOS core module. The folder contains two sub-folders, `files` and `src`. The `src` contains the sources and build files of the `insecureos` binary, while the `files` folder contains all the other module files needed by InsecureOS to install the module, such as the `module.prop` file.

The `manager` folder contains the sources and build files of the InsecureOS manager application. Currently, it only contains the default files when creating an Android application with Android Studio, but in future versions of InsecureOS, all the sources of the manager application will be contained in the folder.

The `modules` folder contains the sources and build files of all the default InsecureOS modules. Each sub-folder contains the files of a particular module. The folder of each module can contain the module files directly without providing a `build.gradle` file, in which case the module folder will be copied as is in the

InsecureOS update package. Alternatively, if a `build.gradle` file exists in the module folder, it will be responsible for building the module and placing its files in the InsecureOS update package.

For example, the `frida-server` module does not include any `build.gradle` file. The files in the folder will therefore be copied as is in the InsecureOS update package. The `MagiskLess-Riru` module (contained in the `riru-core` folder) on the other hand contains a `build.gradle` file. This file is responsible to build the `MagiskLess-Riru` project, extract the relevant files and binary from it, and place them in the InsecureOS update package along with the needed InsecureOS files located in the `files` folder of the module.

Finally, the `update-package` folder contains all the files required to install InsecureOS in its `zip` folder, such as `insecureos.sh` or `utils/utils.sh` (see Section 4.2.1). The `flashable.sh` file is a shell script that is used to compress the `zip` folder into a ZIP archive and push it on the device to be installed.

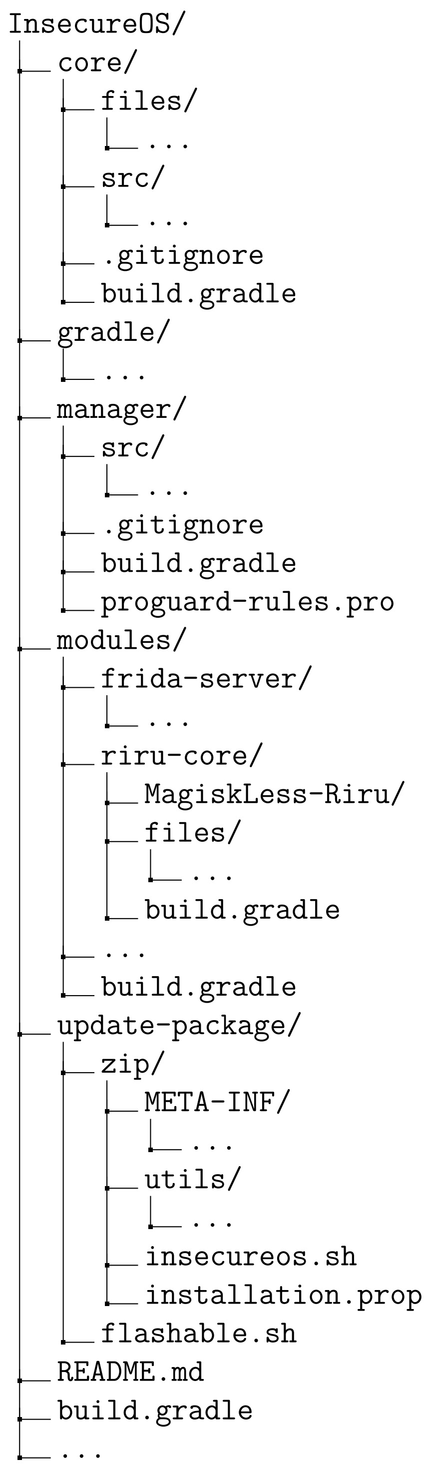


Figure A.1: InsecureOS sources structure

Bibliography

[1] "Android Sources - Android Open Source Project." https://source.android.com/, Accessed: December 2022.

[2] "Samsung - One UI." https://www.samsung.com/us/apps/one-ui/, Accessed: December 2022.

[3] "Android Sources - The Android Kernel." https://source.android.com/docs/core/architecture/kernel, Accessed: December 2022.

[4] "Linux - Android Binder." https://elinux.org/Android_Binder, Accessed: December 2022.

[5] J. Levin, "Binder," in *Android Internals A Confectioner's Cookbook*, vol. II The Developper's View, pp. 275–335, 2022.

[6] "Android Sources - Hardware Abstraction Layer (HAL)." https://source.android.com/docs/core/architecture/hal, Accessed: December 2022.

[7] "Android Sources - Bionic." https://android.googlesource.com/platform/bionic/, Accessed: December 2022.

[8] J. Levin, "The Android Runtime (ART)," in *Android Internals A Confectioner's Cookbook*, vol. II The Developper's View, pp. 223–273, 2022.

[9] "Android Sources - Android Runtime (ART) and Dalvik." https://source.android.com/docs/core/runtime, Accessed: December 2022.

[10] J. Levin, "Dalvik Virtual Machine Internals," in *Android Internals A Confectioner's Cookbook*, vol. II The Developper's View, pp. 199–221, 2022.

[11] J. Levin, "Partitions & Filesystems," in *Android Internals A Confectioner's Cookbook*, vol. I The Power User's View, pp. 61–93, 2021.

[12] "Android Sources - Android partitions." https://source.android.com/docs/core/architecture/bootloader/partitions, Accessed: December 2022.

[13] "Android Developers - Here comes Treble: A modular base for Android." https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html, Accessed: December 2022.

[14] "Android Sources - A/B (Seamless) System Updates." https://source.android.com/docs/core/ota/ab, Accessed: December 2022.

[15] "Android Sources - OTA Updates." https://source.android.com/docs/core/ota, Accessed: December 2022.

[16] J. Levin, "The Android Boot Process," in *Android Internals A Confectioner's Cookbook*, vol. I The Power User's View, pp. 191–212, 2021.

[17] "Android Sources - Android Boot Loader." https://source.android.com/docs/core/architecture/bootloader, Accessed: December 2022.

[18] J. Levin, "User mode startup - init & Zygote," in *Android Internals A Confectioner's Cookbook*, vol. I The Power User's View, pp. 213–214, 2021.

[19] "Android Sources - Android init language." https://android.googlesource.com/platform/system/core/+/master/init/README.md, Accessed: December 2022.

[20] "Android Sources - Application Sandbox." https://source.android.com/docs/security/app-sandbox, Accessed: December 2022.

[21] "Android Developers - Permissions on Android." https://developer.android.com/guide/topics/permissions/overview, Accessed: December 2022.

[22] "Android Sources - Verified Boot." https://source.android.com/docs/security/features/verifiedboot, Accessed: December 2022.

[23] "Android Sources - Android Verified Boot 2.0." https://android.googlesource.com/platform/external/avb/+/master/README.md, Accessed: December 2022.

[24] "The Linux kernel user's and administrator's guide - dm-verity." https://docs.kernel.org/admin-guide/device-mapper/verity.html, Accessed: December 2022.

[25] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, 2001.

[26] S. Smalley and R. Craig, "Security enhanced (se) android: bringing flexible mac to android.," in *Ndss*, vol. 310, pp. 20–38, 2013.

[27] "Android Developers - Network security configuration." `https://developer.android.com/training/articles/security-config`, Accessed: December 2022.

[28] "OkHttp3 - CertificatePinner." `https://square.github.io/okhttp/4.x/okhttp/okhttp3/-certificate-pinner/`, Accessed: December 2022.

[29] "Android Developer - Play Integrity API." `https://developer.android.com/google/play/integrity`, Accessed: December 2022.

[30] "Android Developer - SafetyNet Attestation API." `https://developer.android.com/training/safetynet/attestation`, Accessed: December 2022.

[31] "Android Developers - Verifying hardware-backed key pairs with Key Attestation." `https://developer.android.com/training/articles/security-key-attestation`, Accessed: December 2022.

[32] "GitHub: kdrag0n - Universal SafetyNet Fix." `https://github.com/kdrag0n/safetynet-fix`, Accessed: December 2022.

[33] "GuardSquare - DexGuard." `https://www.guardsquare.com/dexguard`, Accessed: December 2022.

[34] "OneSpan - Mobile Security Suite." `https://www.onespan.com/products/mobile-security-suite`, Accessed: December 2022.

[35] "Linux manual page - ptrace(2)." `https://man7.org/linux/man-pages/man2/ptrace.2.html`, Accessed: December 2022.

[36] "GitHub: topjohnwu - Magisk." `https://github.com/topjohnwu/Magisk`, Accessed: December 2022.

[37] "Magisk - Modules." `https://topjohnwu.github.io/Magisk/guides.html`, Accessed: December 2022.

[38] "Frida." `https://frida.re/`, Accessed: December 2022.

[39] "GitHub: freehuntx - frida-mono-api." `https://github.com/freehuntx/frida-mono-api`, Accessed: December 2022.

[40] “Frida: The engineering behind the reverse-engineering.” https://frida.re/slides/osdc-2015-the-engineering-behind-the-reverse-engineering.pdf, 2015, Accessed: December 2022.

[41] “GitHub: sensepost - Objection.” https://github.com/sensepost/objection, Accessed: December 2022.

[42] “GitHub: RikkaApps - Riru.” https://github.com/RikkaApps/Riru, Accessed: December 2022.

[43] “GitHub: LSPosed - LSPosed Framework.” https://github.com/LSPosed/LSPosed, Accessed: December 2022.

[44] “GitHub: LSPosed - LSPlant.” https://github.com/LSPosed/LSPlant, Accessed: December 2022.

[45] “LineageOS Downloads - Extras.” https://download.lineageos.org/extras, Accessed: December 2022.

[46] “Magisk Tools - resetprop.” https://github.com/topjohnwu/Magisk/blob/master/docs/tools.md#resetprop, Accessed: December 2022.

[47] “TeamWin - TWRP.” https://twrp.me/, Accessed: December 2022.

[48] “LineageOS Android Distribution.” https://lineageos.org/, Accessed: December 2022.

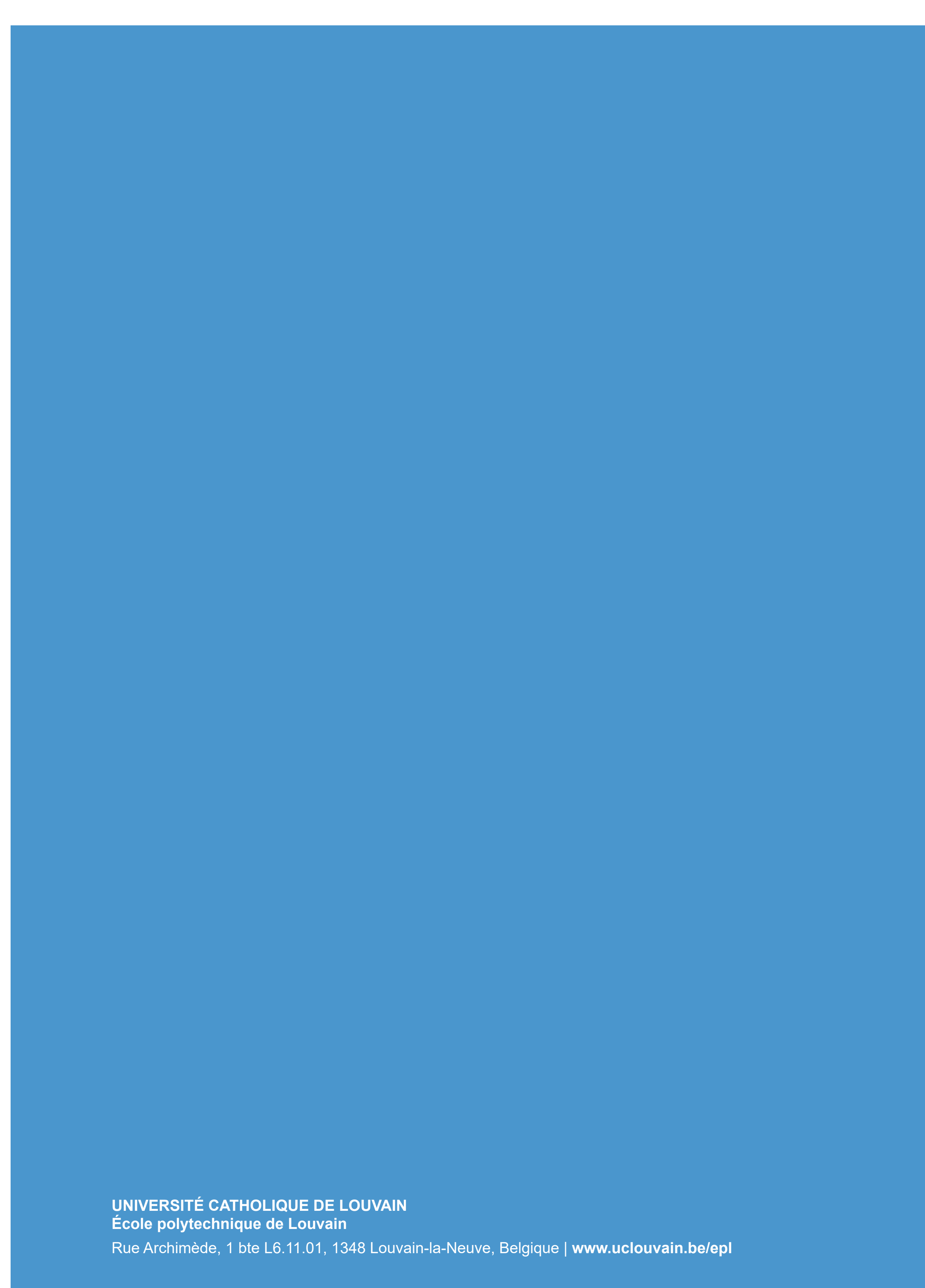
[49] “Android Sources - Inside OTA Packages.” https://source.android.com/docs/core/ota/nonab/inside_packages, Accessed: December 2022.

[50] “Android Sources - Edify syntax.” https://source.android.com/docs/core/ota/nonab/inside_packages#edify-syntax, Accessed: December 2022.

[51] “GitHub: Alhyoss - MagiskLess-Riru.” https://github.com/Alhyoss/MagiskLess-Riru, Accessed: December 2022.

[52] “GitHub: Alhyoss - MagiskLess-LSPosed.” https://github.com/Alhyoss/MagiskLess-LSPosed, Accessed: December 2022.

[53] “Android Sources - How ART works.” https://source.android.com/docs/core/runtime/configure#how_art_works, Accessed: December 2022.


UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**