CHRISTIAN DEBOVI PAIM DE OLIVEIRA

CONTAINER-BASED FRAMEWORK FOR AUTOMATION OF CUSTOMIZED ANDROID SECURITY TESTING ENVIRONMENT GENERATION

(*pre-defense version, compiled at December 11, 2023*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Ricardo Abed Grégio.

CURITIBA PR

2023

RESUMO

Sistemas móveis estão ocupando uma parcela maior e em constante crescimento no uso de tecnologia pessoal pela população mundial, com o Android sendo um dos maiores sistemas operacionais distribuídos no *market share*. Com isso em mente, testar falhas de segurança no ambiente Android tornou-se uma tarefa de importância crescente, considerando que esses testes devem ser reproduzidos em um ambiente controlado. Este artigo propõe uma solução *container-based* para um sistema que cria um dispositivo Android emulado com versões de kernel e Android personalizados. Uma implementação funcional de um *framework* foi criada com base no sistema proposto, sendo usada para executar um estudo de caso usando um *trigger* de uma vulnerabilidade como exemplo de teste de segurança.

Palavras-chave: Android. Testes de Segurança. *Container-based*.

ABSTRACT

Mobile systems are occupying a greater, ever growing share in the usage of personal technology by the word-wide population, with Android being one of the biggest operating systems distributed in the market share. With that in mind, testing security flaws in the Android environment became a task with increasing importance, considering that these tests should be reproduced in a controlled environment. This article proposes a container-based solution for a system that creates an Android emulated device with a customized kernel and Android versions. A working framework implementation was created based on the proposed system, being used to execute a case study using a vulnerability trigger as a security test example.

Keywords: Android. Security Testing. Container-based.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

adb	Android Debug Bridge
AOSP	Android Open Source Project
AVD	Android Virtual Device
CVE	Common Vulnerabilities and Exposures
gcc	GNU Compiler Collection
KASAN	Kernel Address Sanitizer
SD	Secure Digital
SQL	Structured Query Language
SDK	Software Development Kit
IDE	Integrated Development Environment
UI	User Interface
OS	Operational System
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

1.1 PROPOSAL

This article has the goal of proposing a solution for creating customized Android environments in a container for the application of security tests. A orchestrated system will be proposed for tackling this issue. Then, a framework implementation based on the proposed system is created, being used later to test the generation of an Android environment for testing a security vulnerability.

1.2 CHALLENGES

The following changes were encountered in the making of this article:

- Proposing a system for creating a customized Android environment inside a container environment.
- Provide a framework that allows the implementation of the functionalities of the proposed system.
- Execute a case study with a security test for testing the framework implementation of the system.

1.3 MOTIVATION

For the increasing number of vulnerability attacks in mobile devices, the are significantly less platforms for generating proper environments for testing cybersecurity vulnerabilities and flaws in mobile devices (Capone et al., 2022). From the mobile operating systems, Android stands out in terms of open-source accessibility to a variety of tools and repositories for developing and testing the Android environment (Capone et al., 2022).

Considering the related works studied in this article and the tools available for Android development, emulation and testing, this article raises the possibility of proposing a system that can create a heavily customized Android environment for executing security tests in a mostly automatic way, creating a proper environment using a set of parameters, evaluating with the proposed system could be implemented and used for reproducing a real security test.

The usage of a container-based approach was to not only to provide an isolated environment for the running the Android emulated device, but also for the rising usage of container virtualization in current applications (Capone et al., 2022).

1.4 CONTRIBUTION

This article provided the following contributions

- The proposal of a system for creating customized Android environments in a Docker container using Android Emulator.
- A working framework implementation for creating the proposed system, with a case study of a security vulnerability as a proof of concept.

The vulnerability for the case study was not created by the studies of this article, but rather comes from part of a parallel project that included the proposed system in this article, being only used as a test for creating the environment using the implementation of the framework. More details of what specific features were not created in this article can be found in Section 5.2.

1.5 DOCUMENT STRUCTURE

This document has the following structure of chapters:

- **Chapter 2, Background:** specifies some related works to this article, as well as some vital concepts for understating the remaining sections.
- **Chapter 3, Methodology:** provides a detailed explanation of the chosen tools used for the contributions of this article.
- **Chapter 4, Proposed System:** chapter that explains the proposed system for creating customized Android emulated device inside a container.
- **Chapter 5, Evaluation:** Description of the implementation of a framework for the proposed system, as well as providing a case study for generating a proper Android environment for a security test.
- **Chapter 6, Conclusion:** states the conclusions reached in the end of the making of this article.

2 BACKGROUND

In this chapter, it will be detailed the required concepts and related works for a better understanding of this article.

2.1 CONCEPTUALIZATION

2.1.1 Cybersecurity Testing

Cybersecurity testing involves the process of evaluation possible vulnerabilities or potential threats that can affect computational systems. A vulnerability is a weakness or breach in a system that allows an possible attacker to exploit it to achieve a specific end goal (National Cyber Security Centre (NCSC), 2023), being caused by undetected flaws in an given system.

Attackers actively seek and exploit vulnerabilities across multiple different systems, resulting in multiple areas of interest regarding vulnerability exploitation and mitigation. One of the resulting projects is the Common Vulnerabilities and Exposures (CVE) project, that aims to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities (MITRE Corporation, 2023). The CVE project catalogues multiple vulnerabilities for multiple different computational systems, differentiating each vulnerability with an unique identifier in the format *"CVE-YYYY-NNNN"*, where *"YYYY"* is the year where the vulnerability was assigned, while *"NNNN"* is a unique number identifier for the vulnerability.

2.1.2 Linux Kernel and Android Kernel

The Linux Kernel is the core and main component of the Linux Operating System (OS), providing the main interface between the hardware and processes of a computer or any kind of device with a running OS. The kernel responsibilities include memory management, process management (CPU usage by each individual process), interacting with device drivers and dealing with system calls and with the system's security (Red Hat, 2023).

The Android Kernel is based on a mainline of the Linux Long Term Supported (LTS) kernel (a Linux Kernel that has been supported from 4 to 6 years) with Android-specific patches, with these kernels being named as Android Common Kernels (ACKs), having similar behaviour and implementation to the Linux kernel (Android, 2023a).

Each Android version has a compatible Android kernel version for running the system. Many different versions of the Android Kernel source codes are open source, being part of the Android Open Source Project (AOSP) (Android, 2023b), allowing users to experiment in the kernel's code and compilation process, including for security or vulnerability related tests.

2.1.3 Conteinerization

Conteinerization refers to the process of encapsulating the infrastructure, dependencies, environment and source code of a running application or program in a single software package called a container, for ease of distribution and deployment (Amazon, 2023).

Containers are administrated by a container engine, that manages resources between the container and the operational system. One of well-known container engines are the Docker Engine (Docker, 2023c), that allows the creation of containers as isolated processes that have a similar behaviour of a Virtual Machine, encapsulating the system dependencies and application from the host machine. The key difference between Virtual Machines and containers are that VMs virtualize hardware, while containers virtualize the operational system (software) (Docker, 2023d).

2.2 RELATED WORK

The article by Capone et al. (2022) proposes a system called *DockerizedAndroid*, a framework and platform that allows the creation of emulated, customized Android devices (using Android Emulator) running in a Docker container, aiming to provide a proper way to generate environments for exploring Cyber Ranges (simulated teaching environments that allows cyber-security professionals to test their skills without harming a real system), pointing out the scarcity of cyber range generation scenarios for mobile devices.

DockerizedAndroid provides a variety of features, including Android Virtual Device (AVD) execution in Android Emulator, Android applications management, connectivity with the device with Android Debug Bridge (*adb*), data collection and other management functionalities, also providing an User Interface (UI) for interacting with the platform. Later, Capone et al. (2022) was able to execute a security attack reproduction test in the platform using CVE-2018-7661, a vulnerability that allows remote attackers to obtain audio data via certain requests through a Baby Monitor app.

In the context of cyber ranges and security testing, Costa et al. (2022) proposed a framework for the automatic generation of cyber range scenarios through a proposed created language, the *virtual scenario description language (VSDL)*. VSDL is a complex customized language that allows the specifications of the high-level features of the desired cyber range infrastructure.

The framework uses this language to generate a set of scripts for deploying the customized cyber range scenario in a IaaS (Infrastructure as Service) provider, using tools like Terraform (language for infrastructure generation), Packer (defining and customizing OS images) and OpenStack(cloud infrastructure manager). For injecting vulnerabilities in the infrastructure, the framework uses a list of of knows a list of known, vulnerable configurations provided by a database of vulnerabilities (National Vulnerability Database), that are also specified for the

infrastructure by the VSDL language. Later in the article, Costa et al. (2022) shows an example usage that uses the framework to generate the scripts for a cyber range scenario configuration.

3 METHODOLOGY

3.1 TOOLS

This section will describe the tools and technologies that are relevant for understanding this article.

3.1.1 Docker

Docker is tool for allowing running applications or services in isolated environments, called containers. This allows the better isolation for multiple different services running as components in applications, as well a way for distribution of said services using Docker Images and Dockerfiles (Docker, 2023c). Docker also allows the mounting of volumes that share files between the host machine and the created containers. One of the tools offered by Docker is Docker Compose, that allows the definition and running of multiple different containers for applications through a single file (Docker, 2023a).

Another concept that is explored in this article is the *Docker From Docker* approach, used when needed to execute Docker commands inside Docker containers. This approach mounts the Docker Unix socket, so the executed Docker commands (like running a container) are transmitted to the host machine (Microsoft, 2023). This means that launching a second container, inside a first container, will make the second container run alongside the first in the host machine.

3.1.2 Android SDK

The Android SDK (Software Development Toolkit) is a series of open source tools and packages for developing applications for the Android platform. These tools include Android Emulator, a emulator system for high-fidelity emulation of Android devices without needing a physical device (Android for Developers, 2023c).

The Android SDK tools and packages, including Android Emulator, are encapsulated in an IDE called Android Studio (Android for Developers, 2023e), so they can be managed using a graphical interface. For using the SDK tools and packages from a command line approach, Android provides the Android SDK Command Line Tools (Android for Developers, 2023f), including *sdkmanager* and *avdmanager*, that will be better described in the remaining sections.

3.1.3 sdkmanager

The tool *sdkmanager*, part of the Android SDK Command Line Tools, is used to manage packages from the Android SDK repository from a terminal, being able to install, update, remove and list the available packages (Android for Developers, 2023d).

One of the important types of packages available are the AVD system directory package, a directory that contains all the necessary system image files for creating an Android Virtual Device (AVD) for running Android Emulator (Android for Developers, 2023f). For installing a system directory package, an unique string identifier is used, with the installation command being:

```
sdkmanager --install "system-images;apiLevel;variant;arch"
```

Listing 3.1: *sdkamanger* installation command for an AVD system directory package

This command retrieves the system directory package from the repository, saving it in a directory of structure *system-images/apiLevel/variant/arch* in the defined Android SDK home path in the machine. This directory contains important files like a boot partition images, a system image and a default kernel image for the device (Android for Developers, 2023f).

The values *apiLevel*, *variant* and *arch* are values for specifying details about the Android system images. These values are described as:

- ***apiLevel* (*android_api_level*)**: name that represents the Android API Level or identifier for the Android system images. The Android API level serves as a distinctive identifier for the framework API revision offered in a particular version of the Android platform (Android for Developers, 2023g). An example would be *android-23*, that implies API Level 23, witch correspond to Android 6.0.

- ***variant* (*android_tag*)**: name that corresponds to specific features implemented by the system images files. An example would be *google-apis* or *android-wear*.

- ***architecture* (*arch*)**: Represents the CPU architecture used for the system images.

More details about the possible values for these parameters and the correspondence between the Android API Level and the Android Version can be found in Appendix A.

3.1.4 avdmanager

The *avdmanager* tools is used for creating and interacting with Android Virtual Devices (AVDs). An AVD is a configurations that define the characteristics of an Android device (Android for Developers, 2023b).

An AVD is created with an unique name and a existing system directory package retrieved from *sdkmanager*, that specifies the Android version and specifications, with each AVD being stored in a separate directory in the system. This directory contains modifiable data for the device, including cache partition image, an SD card partition image and an image for the data partition for the device (Android for Developers, 2023f). The command for creating an AVD using a system directory package is shown in Listing 3.2.

```
avdmanager create avd --name MyAVD -k "system-images;apiLevel;variant;arch"
```

Listing 3.2: Creating an AVD with name *MyAVD* with the system directory package *"system-images;apiLevel;variant;arch"*.

3.1.5 emulator

Android Emulator creates an emulated device through Android Studio, using a graphic interface, so the creation of the AVD for the device is done by the UI. The *emulator* is the command line counterpart for Android Emulator, allowing the execution of Android Emulator through a specified, previously created AVD (Android for Developers, 2023f).

There are several flags for running *emulator* and customize the created device behavior. For example, like running Android Emulator through Android Studio, *emulator* starts an interactive UI for the emulated phone, so the user can interact with the device display, with features like sound and animations. To disable the graphical interface, there is the flag *-no-window*, that disable graphical elements for *emulator*.

One of the relevant flags for this article is the optional *–kernel* flag, that allows the emulated device to run with a customized Android kernel binary image. If this flag is not specified, *emulator* runs with the standard kernel for the device, that also comes with the AVD system directory package retrieved from *sdkmanager*. Listing 3.3 shows an example of running *emulator* with a custom compiled kernel image and with the name of an existing AVD.

```
emulator -no-window -kernel bzImage -avd MyAVD
```

Listing 3.3: Running *emulator* with a custom kernel binary image *bzImage* with the AVD named *MyAVD* an the flag *-no-window*.

3.1.6 Android Kernel Repository

The Android Open Source Project (AOSP) provides multiple open source tools and repositories for Android related functionalities (Android, 2023b). One of the results of the project is the Android kernel repository tree (*https://android.googlesource.com/kernel/*), that contains multiple Git repositories with the source codes for different Android kernels and related tools. Git is a version manage tool that allows versioning of code by using repositories, branches and commits, allowing different versions of the same code to be stored and used.

These repositories can be clonned and be used for compiling a kernel of desired type, as well as choose branches or commits so the compiled kernel source code can be in a specific version of implementation. The kernel's code and compilation process can be customized at will.

For example, its possible compiling the Android Goldfish kernel version 4.14 using the source code if the repository *https://android.googlesource.com/kernel/goldfish* is retrieved, changing to the branch named *android-goldfish-4.14-dev*.

3.1.7 adb

Android Debug Bridge (*adb*) is a command line program that allows communication with a device using the command line, being able to copy files, install applications or directly access an Android device's (real or emulated) environment through the terminal (Android for Developers, 2023a). Android Emulator also integrates with *adb*, so *adb* is able to recognize a running device.

The *adb* commands is a client-server program divided in three components: a client, responsible for sending adb commands, a daemon, that runs commands on devices, and a server, that manages the communication with the client and daemon (Android for Developers, 2023a). An *adb* client can send commands to a specific server hosted with an IP and port using flags. Listing 3.4 shows how to see the connected devices in an *adb* server that is running on a specific host and port.

```
adb -H 127.0.0.1 -P 5037 devices
```

Listing 3.4: Executing an *adb* command specifying the *adb* server host and port.

3.2 DESIGN

For the framework's main programming language, Python was the chosen, for the following reasons: the ease of use for implementation of complex logic and the availability of free libraries that allow interactions with most of the proposed system structure's, like the connection with databases and storage systems, as well of execution of *bash* commands from the *subprocess*[1] library (for running and inspecting Docker containers).

For the mobile system for emulation, Android was preferred for being an open source environment that runs on nearly every device, as long as having multiple external tools that provide a variety of features (Capone et al., 2022). These tools include the Android SDK Command Line Tools, for creating the required packages and files, Android Emulator, for emulating the device, and *adb*, for allowing interaction with the device by its commands. The AOSP kernel repository three was used for allowing open source access to the Android kernels source code.

Docker is used for isolating the implementations in containers, allowing a clean environment for creating the files for the system and for running Android Emulator. Docker also offers the distribution of customized images for running different types of services using Docker Hub (Docker, 2023b).

For storing the parameters for the kernel and Android, PostgreSQL was used for being an open source database provider with many interesting features (PostgreSQL, 2023), including having its own Docker image *postgres*[2]. For the storage of the kernel and Android files, MinIO was chosen, for being an efficient object storage, utilizing buckets for storing different types of files in a single way (MinIO, 2023), while also having an available official Docker image

[1] https://docs.python.org/3/library/subprocess.html
[2] https://hub.docker.com/_/postgres

minio/minio[3]. PostgreSQL and MinIO have also Python libraries for interacting with them, *psycopg2*[4] and *minio*[5].

[3]https://hub.docker.com/r/minio/minio
[4]https://pypi.org/project/psycopg2/
[5]https://pypi.org/project/minio/

4 PROPOSED SYSTEM

This chapter will propose *DroidOrchestrator*, an orchestrated system to automate the creation of customized Android emulated environments for security related testing. The system aims to provide a solution, in a semi-automatic way, for creating and running an emulated Android device using Android Emulator, using a specified Android version and a customized Android kernel. For this, *DroidOrchestrator* utilizes different components, a database and a storage system to create and manage the required files for creating the emulated environment. All the system components, including the database, storage and the emulated device, were designed to work inside Docker containers.

DroidOrchestrator receives parameters regrading the specification for the device's android and kernel. It then generates necessary image files to create the emulated device, storing their contents in a object storage and their parameters in a SQL database. Two types of images will be defined in the system:

- ***android image*:** a compacted (zip) directory containing all the necessary files (system images) for running Android Emulator with a specified Android version, tag and architecture. This directory is a package retrieved from a repository, using the program *sdkmanager*, from the *Android SDK Command Line tools*. This process is done in a dedicated Docker container.

- ***kernel image*:** A binary for a compiled Android kernel. The kernel source code is retrieved from the Android Kernel repository tree, getting it from a specific repository name. Once the base kernel code is retrieved, the system uses a set of scripts specified by the user to compile the Android kernel in a customized way, generating a custom compiled kernel image. The kernel compilation process is also done in a separate, dedicated Docker container.

Once the necessary image files for creating the environment are generated, *DroidOrchestrator* retrieves them from the storage, creating an Android Virtual Device (AVD) from the *android image*. Using the created AVD and the retrieved *kernel image*, the system runs Android Emulator in a Docker container, alongside an Android Debug Bridge (*adb*) server, making it connectable by an *adb* client by the host machine. The user can then execute *adb* commands in the emulated device running in the Docker container, being able to copy files, install android packages and execute shell commands inside the device, allowing the execution of security tests, like vulnerability exploitation.

The following sections will give a detailed explanation of the expected components and behaviour of *DroidOrchestrator*.

4.1 SYSTEM STRUCTURE

4.1.1 Components

DroidOrchestrator is divided in components, with each one with its own defined set of responsibilities that aim to create the customized environment. The components are code implementations, designed to be isolated in Docker containers, that receive fixed instruction steps from the main component, the *Orchestrator*. The components and their functionalities are described bellow:

- ***Orchestrator*:** The *Orchestrator* is responsible for receiving the user's parameters and sending them to the other components, calling their functionalities in a fixed order, aiming to create the customized emulated Android environment.
- ***KernelBuilder*:** This component is responsible for creating and storing the required image files for running an emulated android system. It communicates with a object storage and a SQL database to store, respectively, the content of the files and their parameter metadata. The creation of the images of each type (*kernel* and *android*) is done in different Docker containers, isolated from the main component.
- ***EnvironmentCreator*:** Has the function of creating the requested emulated device, using the images created by *KernelBuilder*, retrieved using the parameters received from the *Orchestrator*. The emulated device is also created in a dedicated Docker container.

When implementing the system's framework (detailed in Section 5.1), the components being separate instances running in separate Docker containers proved to be challenging, since *Orchestrator* would need to communicate with the user, as well as the other two components of the system. For simplifying the implementation, it was opted to implement the components logic using scripts, so the *Orchestrator* logic could be avoided. All scripts have the parameters specified directly in the implementation, with each script executing a step for creating the emulated device container: creating the android image, creating the kernel image and running the emulated device.

Besides the components, *DroidOrchestrator* also works with different storage mechanisms and Docker containers. The system's interactions between the components, database, storage and containers are represented in Figure 4.1. More information about the database (Section 4.1.2), storage (Section 4.1.3) and the used Docker containers (Section 4.1.4) can be found on their corresponding sections.

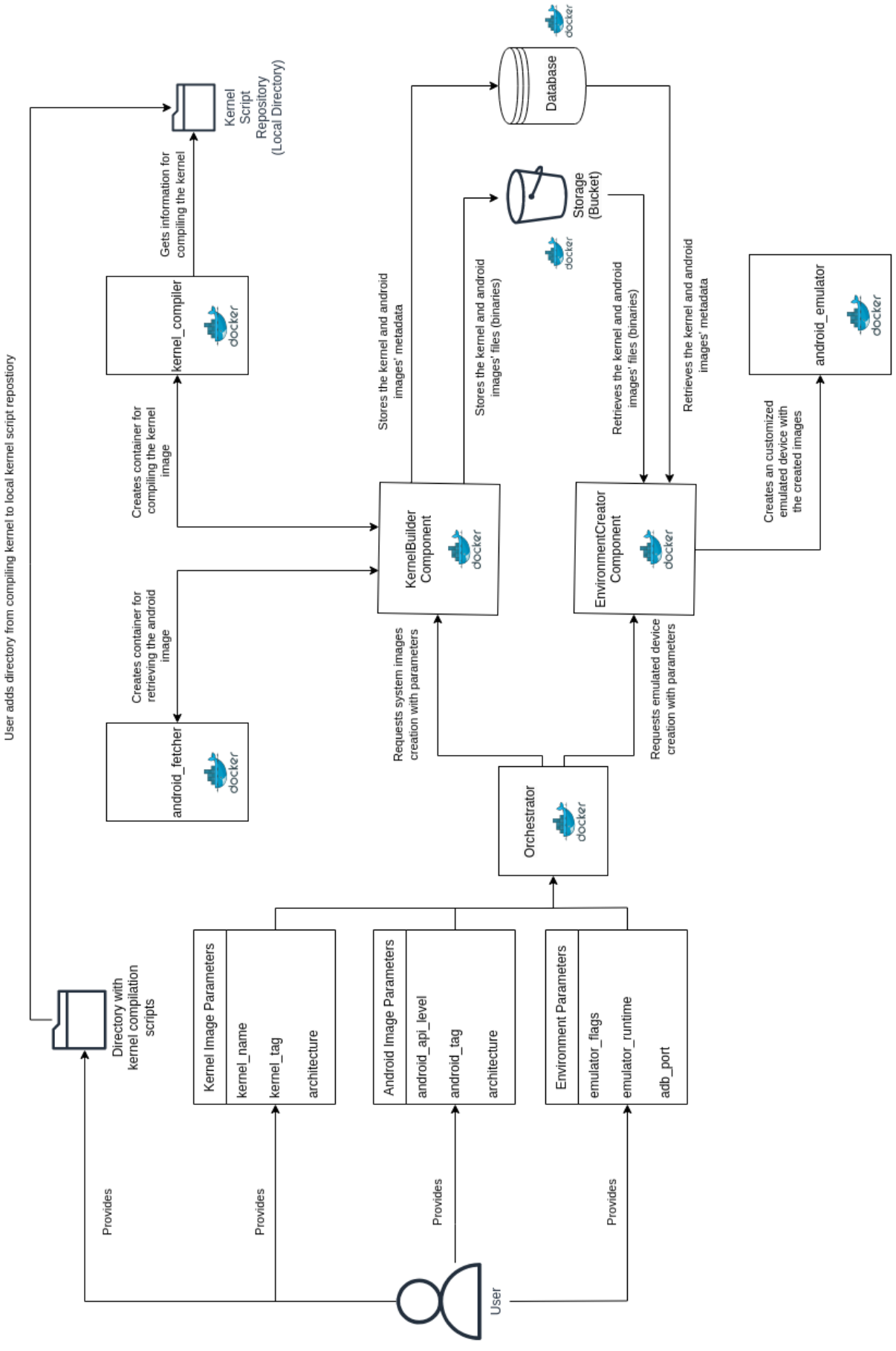

Figure 4.1: *DroidOrchestrator* system diagram.

4.1.2 Database

When the creation of a image of a given type is requested, an instance with the received parameters (*android image parameters* or *kernel image parameters*, as shows in Figure 4.1) is created in an SQL database. *DroidOrchestrator* should know the host, port and database name for the running database service. This database diagram is represented in Figure 4.2.

There are two main tables: *kernel_image* and *android_image*, that are responsible to storing the received parameters for each image type (detailed in Section 4.3), as well as a foreign key referencing the *build_status_id*. The *UNIQUE* constraint in the parameters avoids duplicated insertions of the same image information.

The table *build_status* has a field called *status*, that is pre-populated with three values: *BUILDING*, *COMPLETED* and *ERROR*. This field is used to indicate the status of an image through the *build_status_id* field. An image with the *build_status_id* referencing the id of the entry *COMPLETED* indicates that the image is completed. The status, in the context of the system, have the following meanings:

- ***COMPLETED*:** Describes that the following image is completed, meaning that the image creation process was finished successfully. The requested image exists and can be retrieved from the object storage.
- ***BUILDING*:** The image is being created and is not yet completed. Means the component is still creating the requested image and it is yet not present in the object storage.
- ***ERROR*:** Means the image creation process failed due to an error and could not be stored. When an image is with the status *ERROR*, its creation needs to be requested again with the same creation parameters, so the system can update the old entry with the status *BUILDING*.

By setting and updating the correct values of the *build_status_id* for an image entry in the database, *DroidOrchestrator* can better control the image creation flow. More details about the usage of these statuses can be found in Section 4.4.

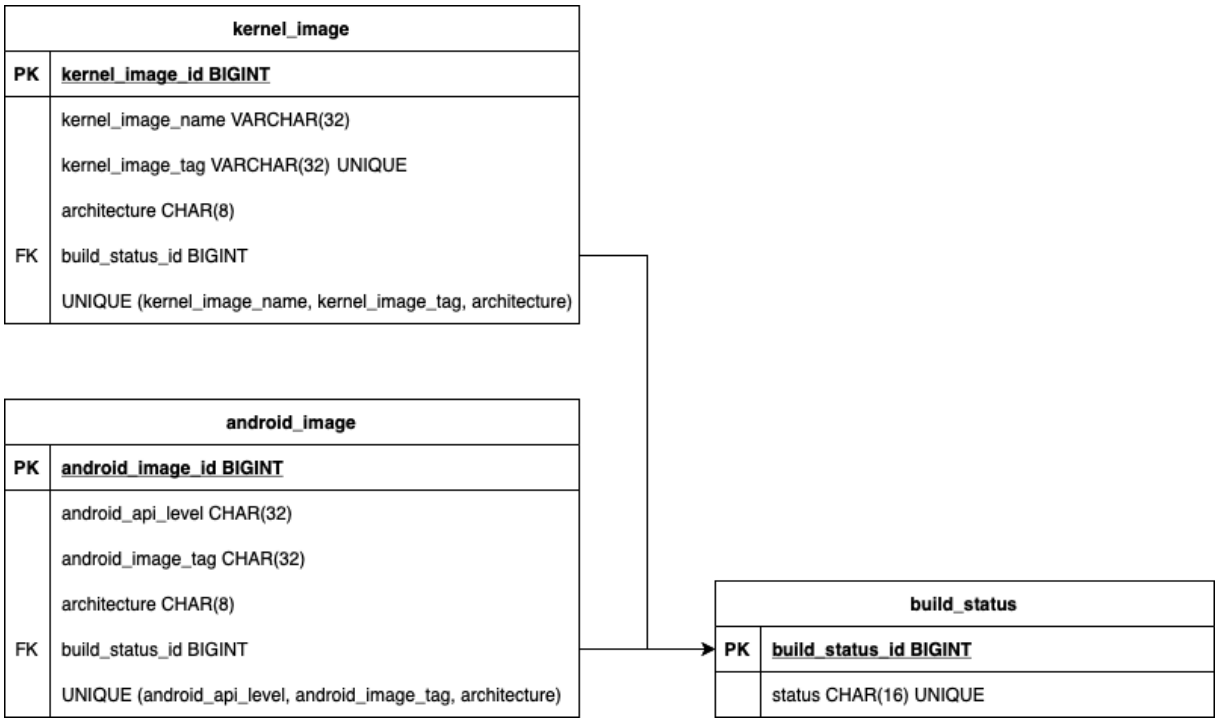

Figure 4.2: *DroidOrchestrator* database diagram.

4.1.3 Storage System

For storing the image files created by the *KernelBuilder* component, the system uses an object storage, that has a defined bucket. This bucket is responsible for storing the image files used by *DroidOrchestrator*, so the files can be retrieved later to create the emulated Android device. Each file or image stored in the bucket is defined by a given key, a unique string value that defines only a single file. For each image type, a pattern was given for the key value and the filename for the stored file. The pattern used for the key and filename for each image type is described in Figure 4.3 and will be described in details bellow.

For the *android image*, a zip file, the key value was defined as the concatenation of the string parameters for the android image (the same that are represented in the *android_image* database table), separated by the *'_'* character. And example, if the values for *android_api_level*, *android_image_tag* and *architecture* are, respectively, *android-29*, *google_apis*, *x86_64*, the key value would be *android-29_google_apis_x86_64*, ensuring an unique key value based on the database unique constraint (Figure 4.2), with the filename for this file is composed by "*android-29_google_apis_x86_64.zip*".

For the *kernel image*, the key value is the same value for the unique *kernel_image_tag* database column (*kernel_tag* parameter). In this case, the filename is defined as the *kernel_tag* itself, with no file extensions (since the kernel is a binary image).

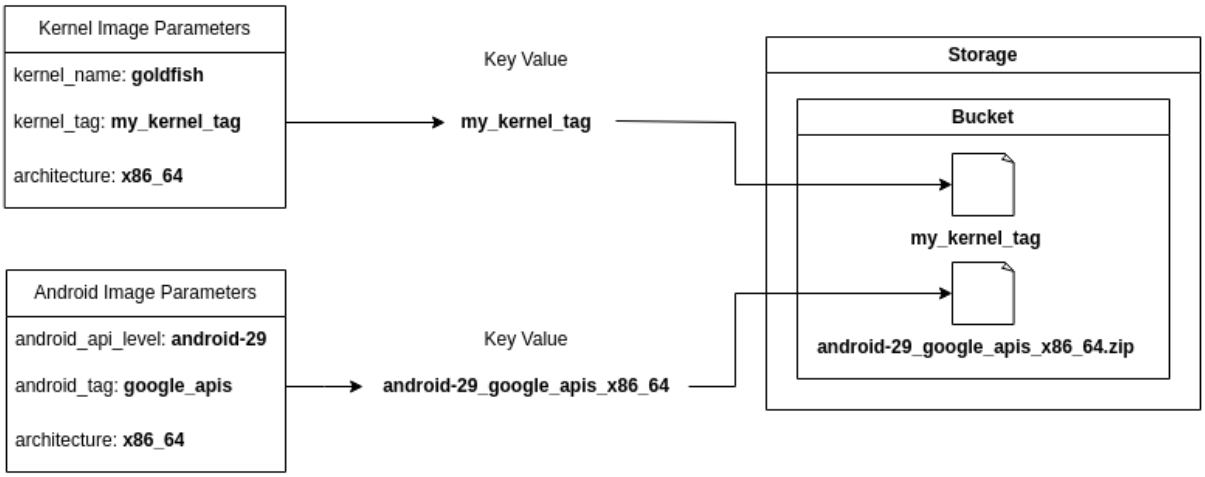

Figure 4.3: Relation between the received parameters, key value and filename in the *DroidOrchestrator* storage.

4.1.4 Docker Containers

The system executes some tasks inside Docker containers, for practicality and isolation. Each container has a defined behaviour when executed, while also containing the necessary dependencies installed for doing its defined purpose. After a container finishes its defined job, it is removed, releasing the used resources of the host machine. Each Docker container and its behaviour will be defined below.

- ***orchestrator*:** Contains the implementation of the *Orchestrator* component, being able to receive the user's parameters and coordinate the other components for creating the emulated device.
- ***android_fetcher*:** Has installed the Android SDK Command Line Tools. Is responsible for retrieving the Android system image files from the *sdkmanager* repository and compressing it into a zip file.
- ***kernel_compiler*:** Responsible for retrieving the kernel source code from the kernel repository and compiling it, generating the kernel binary image file. Has dependencies for compiling and modifying the kernel, including Git and *gcc*.
- ***kernel_builder*:** Responsible for having the implemented code for the *KernelBuilder* component. Is able to receive the parameters from the *Orchestrator*, create the required *kernel image* and *android image* and store it in the database and bucket storage. For creating the *android* and *kernel* images, this container must be able to launch other containers from itself, so it can execute *android_fetcher* and *kernel_compiler*, while also retrieving the images.
- ***android_emulator*:** Container that runs Android Emulator and the *adb* server. Has Android SDK Command Line tools installed for creating an Android Virtual Device (AVD), as well as *emulator* for running the device.

- ***environment_creator*:** Contains the implementation of *EnvironmentCreator*, being able to retrieve the *android image* and *kernel image* from the storage and running a separate *android_emulator* Docker container.

The database and storage also should have the necessary Docker images for creating its services in containers, being able to receive connections from the components. As pointed in Section 4.1.1, the component logic was implemented using a sequence of scripts. Each of these scripts was run inside a Docker container, to simulate the components' behaviour, without considering the *Orchestrator*. The details about these scripts can be found in Section 5.1.

4.2 SYSTEM WORKFLOW

A detailed workflow for *DroidOrchestrator* consists of the following steps:

- ***Step 1, Parameter Specification*:** The user provides the necessary parameters to the creation of the environment to the *Orchestrator*, including the Android Image Parameters and the Kernel Image Parameters. The user also adds a directory with the kernel compilation scripts to the Kernel Script Repository. The *Orchestrator* then sends the parameters to the other system components.

- ***Step 2, Images Creation*:** The component *KernelBuilder*, with the received parameters from *Orchestrator*, checks the database if the requested images (android and kernel) exist. If not, *KernelBuilder* launches two other Docker containers, *android_fetcher* and *kernel_compiler*, that create the *android image* and *kernel image*, respectively. Once the images are created, *KernelBuilder* stores their binaries in the storage (bucket) and their received parameters in the database, removing the used containers.

- ***Step 3, Environment Creation*:** The component *EnvironmentCreator*, using the received parameters from *Orchestrator*, checks if the *android image* and *kernel image* are present in the database. If so, *EnvironmentCreator* retrieves the images' binaries from the bucket storage. Once the images are retrieved, a instance of the *android_emulator* container is launched and uses the images to create and Android Virtual Device (AVD) and run *emulator*. The *android_emulator* runs for a specified period of time, then is removed.

4.3 PARAMETER SPECIFICATION

DroidOrchestrator receives three sets of parameters: *android image parameters*, *kernel image parameters* and *environment parameters* (illustrated in Figure 4.1). The *android image parameters* and *kernel image parameters* are used in the creation of the *android image* and the *kernel image* by *KernelBuilder*, being also stored in the database, so the images can be retrieved using the same

parameters by *EnvironmentCreator*. The *environment parameters* are used in *EnvironmentCreator*, for specifying some details for running emulated device.

These parameters should be specified directly to the *Orchestrator* and distributed through the other components. The way the *Orchestrator* receives this parameter is not strictly defined, either it being received from a terminal, a configuration file or or an through an User Interface (UI). As said before (Section 4.1.1), the implementation of the components logic was made using a series of scripts, so these parameters are directly specified in each script and are not received from the user (Section 5.1).

For the creation of the *kernel image*, a folder with scripts and files for compiling the specified kernel by the *kernel image parameters* must be provided by the user, by adding it directly to a directory, called *kernel script repository*. These folders have a defined structure expected by the system.

The remaining subsections will describe the details about each received parameter, as well as the structure of the *kernel script repository*.

4.3.1 Android Image Parameters

The *android image parameters* (*android_api_level*, *android_tag*, *architecture*) correspond to the ones defined in Section 3.1.3, being used to create an unique string identifier for retrieving a system directory package, from the Android SDK repository, with the necessary system images for a specific Android version, variant and architecture.

These parameters are also inserted in the *android_image* database table (defined in Section 4.1.2), so the images can identified and retrieve when necessary. Information regarding these parameters can be found in Appendix A.

4.3.2 Kernel Image Parameters

The *kernel image parameters* (*kernel_name*, *kernel_tag*, *architecture*) were designed to work with the AOSP Android kernel repository tree (detailed in Section 3.1.6) and the locally implemented *kernel script repository*, that will be detailed in Section 4.3.3.

The *kernel name* or *kernel_image_name* refers to the repository name in the Android kernel repository tree, for example, *goldfish*, that refers to the repository in *https://android.googlesource.com/kernel/goldfish*, being used as a identifier for retrieving an specific kernel source code for compilation.

The *kernel tag* or *kernel_image_tag* serves as an unique identifier, chosen by the user, for the kernel for *DroidOrchestrator*, including as a key for storing the kernel in the storage (Section 4.1.3) and as a directory name in the *kernel script repository*.

The *architecture* parameter values is particularly not used for retrieving or compiling the desired kernel, but serves to identify the target CPU for the kernel, being inserted in the database as well with the remaining parameters.

4.3.3 Kernel Script Repository

DroidOrchestrator allows a custom compilation and customization process for an Android kernel, that can be used for running Android Emulator. For allowing this process of customization for a kernel, the idea was having the user specify the scripts and required files for compiling a kernel of a given repository in the AOSP kernel repository tree. The user would specify a directory containing the structure represented in Figure 4.4, with this structure detailed bellow:

- ***kernel_name*:** The first folder in the directory structure, with the same name as the *kernel_name* parameter.
- ***kernel_tag*:** The second folder in the directory structure, named after the *kernel_tag* parameter, being an unique folder name. This folder contains three shell script files (*compile.sh*, *before_compile.sh*, *after_compile.sh*, with the last two files being optional) that specify the compilation process of the kernel, as well as a directory called *required_files*, containing all the required files for compiling the kernel.
- ***compile.sh*:** The main file that executes the kernel compilation process. After running this script, there must be a valid compiled kernel binary generated.
- ***before_compile.sh*:** An utility script that is executed before the compilation process (*compile.sh*). This script is optional and only executed if specified.
- ***after_compile.sh*:** An utility script that is executed after the compilation process (*compile.sh*). This script is optional and only executed if specified.
- ***required_files*:** A directory containing all the required files for modifying or customizing the kernel or its compilation process. If no files are required, this directory should be empty.

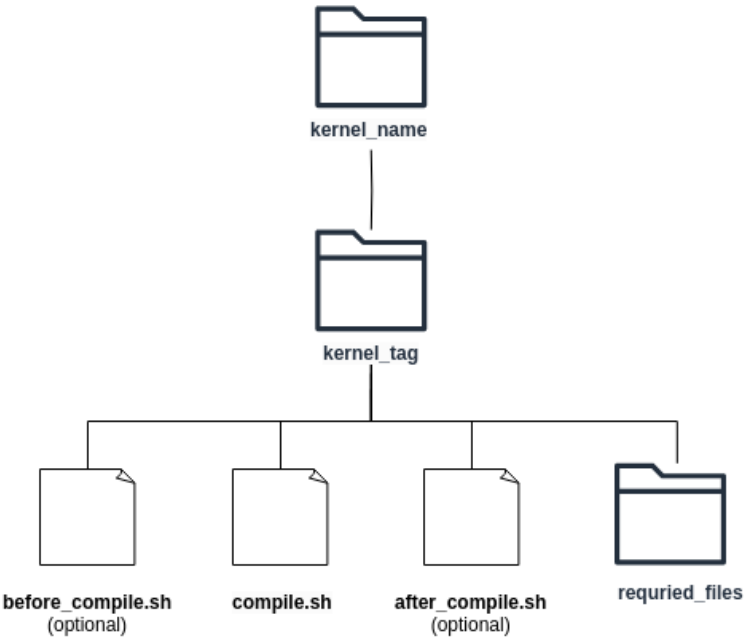

Figure 4.4: Visual representation of the directory structure that needs to be provided for compiling a kernel using *DroidOrchestrator*.

For making this directory usable by *DroidOrchestrator*, it would be placed inside a known directory in the system, that will be called *kernel script repository*. This folder would be composed of all the directories containing the scripts for compiling the kernels supported by the system. This structure is represented in Figure 4.5.

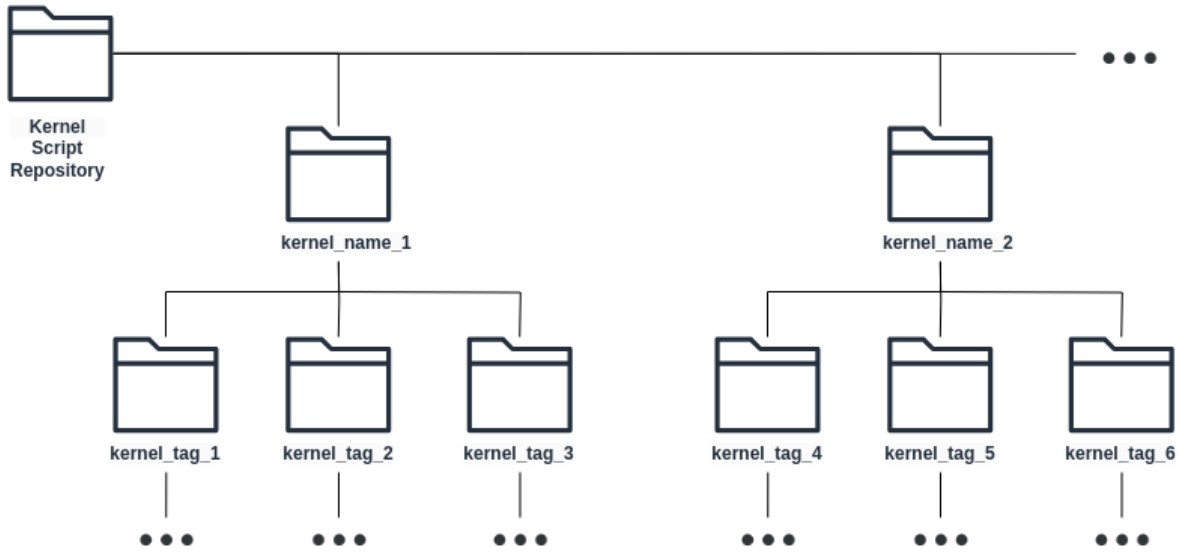

Figure 4.5: Visual representation of *kernel script repository* structure.

This directory structure would be used by *DroidOrchestrator* for compiling a specific kernel, since the folder path represented by *"kernel_name/kernel_tag"* would be unique for each kernel. The *kernel script repository* should be accessible by the system, so it is able to compile

the desired kernel, and by the user, so the structured directory with the kernel scripts can be specified.

4.3.4 Environment Parameters

The *environment parameters* (*emulator_flags*, *emulator_runtime*, *adb_port*) are specifications for the way the emulated device will run.

The *emulator_flags* parameter represents the flags that the *emulator* command should run, specifying the customized details for running Android Emulator. This parameters should be a list of flags, that are represented by strings.

The *emulator_runtime* is an integer value, in milliseconds, that represents the maximum amount of time that the emulated device container should run, being useful for automatically stopping and removing the emulator container after an given amount of time, avoiding manual intervention.

The last parameter, *adb_port* represents in witch port the device should be accessible for connecting to the emulated device through *adb*. This port is used to create an dedicated *adb* server in the *android_emulator* container.

4.4 IMAGES CREATION

The two types of images created by *DroidOrchestrator*, *android images* and *kernel images*, are used for running an emulated device in a Docker container using Android Emulator. These images are created by *KernelBuilder*, using the specified set of parameters in Section 4.3. In the following, it will be detailed how each type of image is created.

4.4.1 Android Images

The creation process of an *android image* is based on the retrieval of the system directory package containing the Android system images using *sdkmanager*. This directory is then compressed into zip file, being store in a bucket in the system's object storage.The process can be described with the following steps:

- ***Step 1, Insert image information in the database*:** *KernelBuilder* checks the database if there is an instance in the *android_image* table with the requested *android image parameters*. If there is no instance of the image in the database or there is an instance with status *ERROR*, The *android image parameters* are stored in the *android_image* table with the status *BUILDING*. If there is already an instance with the status *COMPLETED* or *BUILDING* in the database, it means that *KernelBuilder* already created or is creating the requested *android image*, so it skips the remaining steps and returns.
- ***Step 2, Launch the container and retrieve the android system directory package (android image)*:** A container instance of *android_fetcher* is launched by *KernelBuilder*.

Then, *android_fetcher* uses *sdkmanager* to fetch the system directory package from the repository, corresponding to the given *android image parameters*(Section 4.3.1). This directory is then compressed into a zip file and then returned to *KernelBuilder*.

- ***Step 3, Storing the image in the object storage*:** The zip file corresponding to the *android image* is stored in the local storage for later use (as described in Section 4.1.3). Then, *KernelBuilder* updates the instance for the image in the database with the status *COMPLETED*. If there was any error while retrieving the *android image* in the *android_fetcher* container, then the image status should be updated with the *ERROR* status.

4.4.2 Kernel Images

The process for creating the *kernel image* is quite similar to the creation of the *android image*, being described with the following steps:

- ***Step 1, Insert image information in the database*:** *KernelBuilder* checks the database if there is an instance in the *kernel_image* table with the requested *kernel image parameters*. If there is no instance of the image in the database or there is an instance with status *ERROR*, The *kernel image parameters* are stored in the *kernel_image* table with the status *BUILDING*. If there is already an instance with the status *COMPLETED* or *BUILDING* in the database, it means that *KernelBuilder* already created or is creating the requested *kernel image*, so it skips the remaining steps and returns.

- ***Step 2, Launch the container for compiling the kernel (kernel image)*:** A container instance of *kernel_compiler* is launched by *KernelBuilder*. Then, *kernel_compiler* clones the repository from the Android kernel repository tree using the *kernel_name* parameter. With the cloned repository, *kernel_compiler* searches for the path corresponding to *"kernel_name/kernel_tag"* in the *kernel script repository*, executing, in order, *before_compile.sh*, *compile.sh*, *after_compile.sh*. After the kernel finishes compiling, the binary image is returned to the *KernelBuilder* component.

- ***Step 3, Storing the image in the object storage*:** The binary file corresponding to the *kernel image* is stored in the local storage for later use (as described in Section 4.1.3). Then, *KernelBuilder* updates the instance for the image in the database with the status *COMPLETED*. If there was any error while compiling the kernel in the *kernel_compiler* container, then the image status should be updated with the *ERROR* status.

4.5 ENVIRONMENT CREATION

This section will explain how the emulated device is created by the *EnvironmentCreator* component, using the specified parameters and available *android* and *kernel* images.

4.5.1 Retrieving the Images

For retrieving the created *android image* and *kernel image* that were created by *KernelBuilder*, *EnvironmentCreator* searches the database with the same parameters used to create both images in the previous step, ensuring that both images are marked wit status *COMPLETED* in the database.

For accomplishing this, *EnvironmentCreator* checks if the image is with status *COMPLETED* from time to time, using a fixed time interval defined in the system. This verification occurs until a maximum timeout (also defined in the system) is reached or if the image reaches the status *COMPLETED*. If the timeout is reached and the image is not completed yet, the system should return an error. This verification process should be done for both the *android* and *kernel* images before retrieving then from the storage.

4.5.2 Starting the Emulated Environment Container

Once both the *android image* (zip file) and the *kernel image* (binary) are retrieved from the storage, *EnvironmentCreator* launches the a container *android_emulator*.

This container starts by uncompressing the *android image* directory inside the container, moving the directory to the same path where *sdkmanager* stores the system directory packages when installing them. With the Android system directory package available, *avdmanager* is used to create and Android Virtual Device (AVD) with said package. Then, the container launches *emulator* using the created AVD, the flags specified by the parameter *emulator_flags* and the custom *kernel image* through the command line. The container also starts and *adb* server alongside emulator, using the parameter *adb_port* as its main port for executing *adb* commands.

Once the *android_emulator* container starts, *EnvironmentCreator* checks periodically if the container is running. Any error while executing the *android_emulator* that causes the container to stop should make *EnvironmentCreator* to finish execution and report the error to the system. If no error happens while the *emulator* is running, *EnvironmentCreator* kills the container after the period of time specified by the *emulator_runtime* parameter.

5 EVALUATION

This chapter will describe an implementation for the *DroidOrchestrator* system, the limitations and a case study for creating an emulated environment.

5.1 IMPLEMENTATION

This section will describe the implementation of a framework for creating an Android emulated environment based on *DroidOrchestrator*. The chosen technologies and tools for the system can be found in Chapter 3. Some details about the implementation of the system workflow will be found in Details about the results of the case study for the implementation, in Section 5.3.

There are some key differences in the implementation of the system and from the system's proposal of Chapter 4. These differences will be pointed in 5.4, as well as the limitations they impose.

5.1.1 Components Implementation

Each one of the main components, *KernelBuilder* and *EnvironmentCreator* were implemented as Python modules. Each module contains classes that implements methods for allowing *KernelBuilder* and *EnvironmentCreator* to work as they were described in Section 4.1.1. The classes implemented are:

- ***ImageEnvironmentInterface*:** This class implements methods for creating and storing the *android* and *kernel* images, as described in Section 4.4. There also methods for retrieving an image of a given type (*android* or *kernel*) from the storage. This class is part of the *KernelBuilder* module.
- ***DockerAndroidEmulator*:** Class that implements a method for running Android Emulator in a container, as described in Section 4.5, using the *kernel* and *android* images present in the storage and database. This class is part of the *EnvironmentCreator* module.

These classes have attributes with clients for interacting with the database and its cached values(*PostgresDatabase* and *FixedCacheFromDb*), with the storage system (*MinIOSampleStorage*) and to launch and interact with Docker containers (*DockerClient*), acting as a framework for creating the necessary files and creating the emulated environment. All these clients are classes, that were implemented using the Python libraries described in Section 3.2. Figures 5.1 and 5.2 show the class diagrams for both classes.

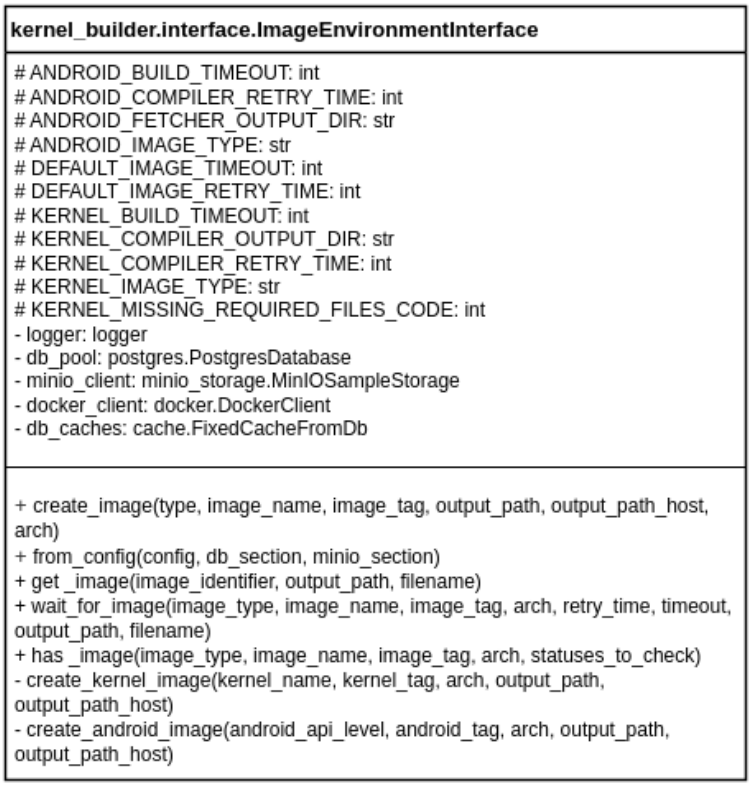

Figure 5.1: Class diagram for the *ImageEnvironmentInterface* class, part of the *KernelBuilder* module.

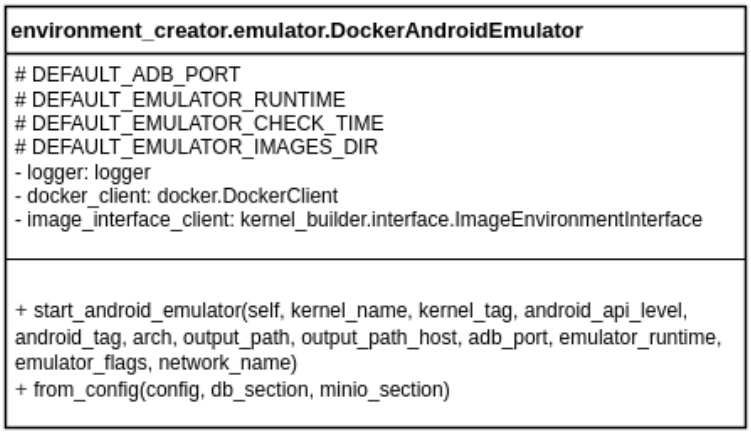

Figure 5.2: Class diagram for the *DockerAndroidEmulator* class, part of the *EnvironmentCreator* module.

Note that *DockerAndroidEmulator* has an attribute of *ImageEnvironmentInterface*, with this being due to *DockerAndroidEmulator* using the *ImageEnvironmentInterface* functions to retrieve the images from the storage for creating the Android Emulator container.

The *Orchestrator* component, as mentioned in Section 4.1.1, was not implemented, due to the complex logic of interacting with the remaining components and communicating with the user. For allowing a similar behaviour to the *DroidOrchestrator* system without the *Orchestrator*, a set of Python scripts were created for executing the system workflow and create the Android Emulator container as a proof of concept, utilizing the classes *ImageEnvironmentInterface* and *DockerAndroidEmulator*.

These scripts do not receive parameters from the user, but instead have the *android image parameters*,*kernel image parameters* and *environment parameters* specified directly in the scripts' code. These parameters were specifically set for creating the emulated environment for an example case study, that will be described in Section 5.2. Each script will be described bellow.

- ***First script: test_interface_android.py*:** This script executes the creation of an *android image*, following the workflow described in Section 4.4.1. It launches a container that retrieves the system directory package from the *sdkmanager* repository, compresses it in a zip file and stores it in the storage, while also updating the database with the *android image parameters*.
- ***Second script: test_interface_kernel.py*:** Compiles a kernel image, executing the process described in Section 4.4.2. It launches a container that clones the repository from the Android kernel repository tree, compiling it with scripts specified in the *kernel scripts repository* and storing it afterwards.
- ***Third script: test_emulator_container.py*:** This scripts, using the images created by the previous scripts, starts Android Emulator and an *adb* server inside a Docker container, as described in Section 4.5.

By executing these scripts in the order, the system workflow for *DroidOrchestrator* is maintained, not considering the user interaction. All the scripts run in Docker containers, for simulating their behaviour inside a Docker environment (more info in Section 5.1.3). Figure 5.3 shows a diagram of the implementation for the case study, considering the described scripts.

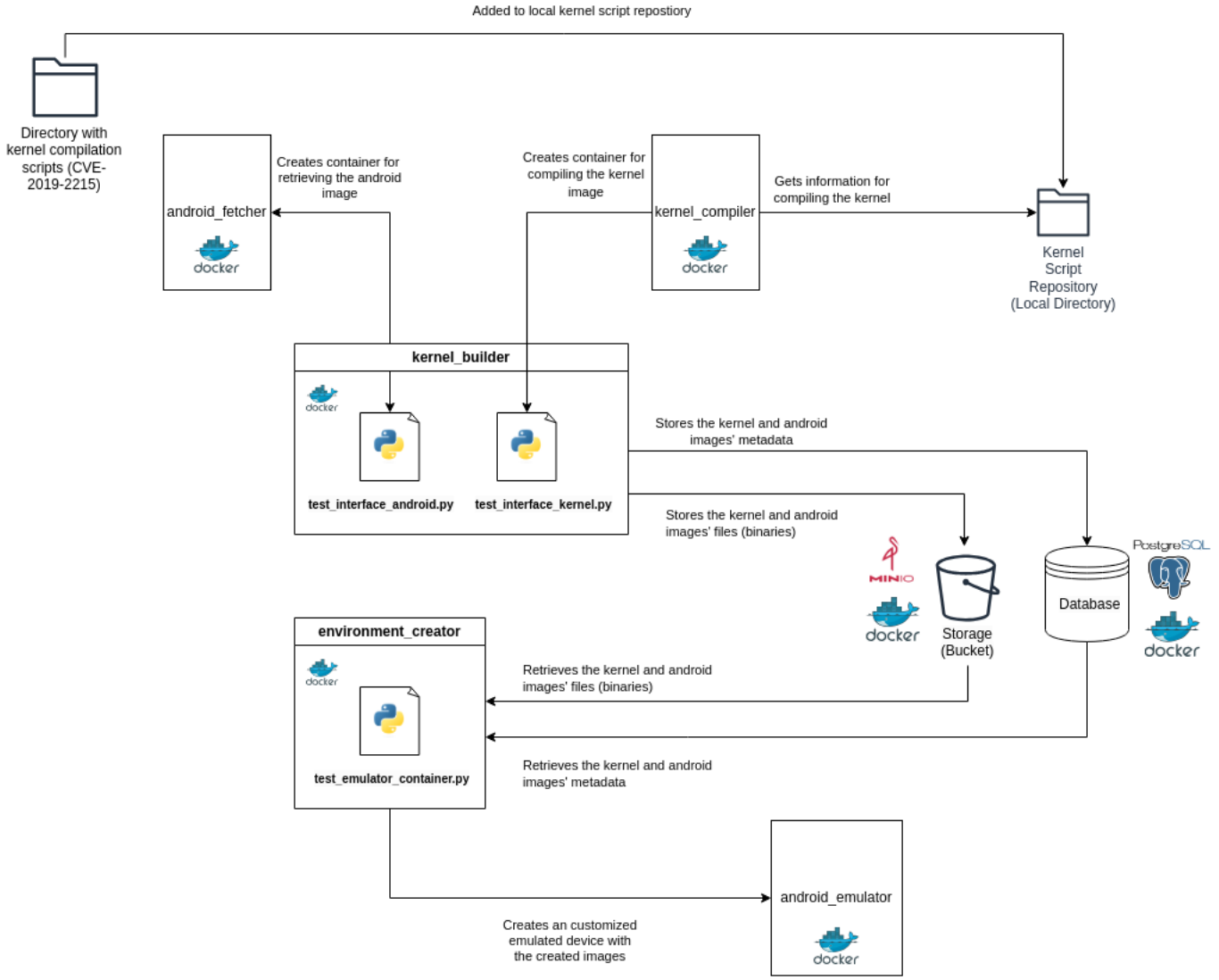

Figure 5.3: Visual representation of the implementation using Python scripts.

5.1.2 Database and Storage Implementation

Both the PostgresSQL database and storage were implemented in Docker containers, using their official Docker images from Docker Hub and are implemented as they were described in Chapter 4.

The database Docker image *kernel_builder_postgres* is a modified version of the official *postgres* image from Docker Hub, with the addition of creating the database with the schema from Figure 4.2 and populating the *build_status* database table with the statuses from Section 4.1.2. Listing 5.1 shows the code of the Docker Compose file that runs the database and storage containers.

```
services:
   test-builder-postgres:
      image: kernel_builder_postgres:latest
      networks:
         - test_builder_network
      environment:
         - POSTGRES_USER=postgres
         - POSTGRES_PASSWORD=123
      healthcheck:
         test: ["CMD-SHELL", "pg_isready -U postgres"]
         interval: 5s
         timeout: 30s
         retries: 3
      volumes:
      - test_builder_postgres:/var/lib/postgresql/data:rw
   test-builder-minio:
      image: minio/minio
      networks:
         - test_builder_network
      ports:
         - 9000:9000
      volumes:
         - test_builder_storage:/data:rw
      environment:
         - MINIO_ACCESS_KEY=minio_access_key
         - MINIO_SECRET_KEY=minio_secret_key
      command: server /data
```

Listing 5.1: The *docker-compose.yml* file lines that creates the database and storage containers.

For connecting and interacting with the database and storage, the *ImageEnvironmentInterface* and *DockerAndroidEmulator* classes receive a configuration file when they are created. This file is represented in Listing 5.2.

```
[minio]
host=test-builder-minio
port=9000
bucket=testbucket
access_key=minio_access_key
secret_key=minio_secret_key
https_enabled=false

[postgres]
database=kernel_builder
username=postgres
password=123
host=test-builder-postgres
port=5432
```

Listing 5.2: The configuration file used for specifying the database and storage connection details.

5.1.3 Docker Containers Implementation

The Docker containers implemented are the same from the described in Chapter 4, Section 4.1.4, with the exception of the *orchestrator* container, since the *Orchestrator* component was not implemented. Each Docker container has a base Docker image of Linux Ubuntu 20.04, with each container installing the required packages and tools for doing its specific job.

The *kernel_builder* container has the *KernelBuilder* Python module installed, being used for executing the scripts *test_interface_android.py* and *test_interface_kernel.py* scripts, simulating the behaviour of a component running in a Docker container. The same can be said by the *environment_creator* container, having the *EnvironmentCreator* Python module installed and being used to execute the *test_emulator_container.py* script, as illustrated in Figure 5.3.

5.2 CASE STUDY

In this section, we will show a case study for the implementation, where an emulated environment is generate for a specific security test.

5.2.1 Security Test Specification

For the case study, the CVE-2019-2215[1] was chosen. This CVE exploits a user-after-free (usage of a structure that has already been freed from memory) in *binder.c* that aims to allow a privilege escalation (obtaining high access privileges to software resource through security flaws) in the Linux Kernel. Binders are interprocess communication mechanisms for Android (eLinux Community, 2023). This vulnerability was patched in the Linux Kernel versions above 4.14, so the target kernel version is 4.14 or bellow. The Android version used is Android 10 with access

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2215

to the Google APIs. The versions for the system for the given vulnerability can be found in Table 5.1.

This CVE's trigger is the reproduction of the user-after-free using the binder structures, while the exploit is the escalation of privileges through the manipulation of the memory addresses after the trigger. Listing 5.3 shows the program used to trigger the vulnerability. This program allocates a *binder_thread* structure, then frees it using the *ioctl()* function (line 26). This structure is then freed again after the program ends, permitting a double free of the same structure.

```
1  #include <fcntl.h>
2  #include <sys/epoll.h>
3  #include <sys/ioctl.h>
4  #include <unistd.h>
5
6  #define BINDER_THREAD_EXIT 0x40046208ul
7
8  int main(int argc, char const *argv[]) {
9     int fd, epfd;
10    //create an epoll event
11    struct epoll_event event = { .events = EPOLLIN };
12
13    //for using binder, we should open the kernel binder module
14    //now, fd is the file descriptor for the binder IPC
15    fd = open("/dev/binder", O_RDONLY);
16
17    //create an epoll instance
18    //'epoll' API is used when we want to monitor multiple file descriptors
19    epfd = epoll_create(1000);
20
21    //we add (EPOLL_CTL_ADD) an event (&event) associated with a file
          descriptor (fd) to our created 'epoll' (epfd)
22    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
23
24    //all interactions with the driver is made with the 'ioctl' function
25    //here, we communicate with the binder we created and exit it
26    ioctl(fd, BINDER_THREAD_EXIT, NULL);
27
28    close(fd);
29    close(epfd);
30
31    return 0;
32 }
```

Listing 5.3: The *trigger.cpp* file that generates a use-after-free scenario.

CVE-2019-2215 Device Specifications	
Android Version	Android 10 with Google APIs access
Kernel Version	Android Goldfish Kernel 4.14 (with unpatch in binder.c and custom compilation)
CPU Architecture	x86-64

Table 5.1: System specifications for the CVE-2019-2215 vulnerability.

The case study will aim to generate a proper an emulated environment for testing the execution of the trigger for CVE-2019-2215, from the creation of the required *kernel image* and *android image* for the environment, to running Android Emulator and triggering the vulnerability using *adb*.

The study involving the CVE-2019-2215 was part of a parallel project done by other group members that integrated to the proposal of this article. With this in consideration and to avoid diverging from the topic of this article, the CVE will not be described in great detail, and will be used exclusively to test the emulated environment generation for this case study. The following information was received from external sources and not produced by this article:

- The study of the trigger and exploitation of the vulnerability, including *trigger.cpp* file and the *Makefile* that compiles it.
- The parameters for the environment generation, including the Android version (Android 10 with Google APIs) and the Linux Kernel version and type (Goldfish 4.14), including the scripts and required files for compiling the kernel, that were adapted to be used by the implementation.
- The commands used to trigger the vulnerability in the emulated device using *adb*.

5.3 PRELIMINARY RESULTS

This section will show how the environment for the CVE-2019-2215 was generated using the implementation of the framework.

5.3.1 Parameter Specification

The chosen *android image parameters*, *kernel image parameters* and *environment parameters* are illustrated in Figure 5.4.

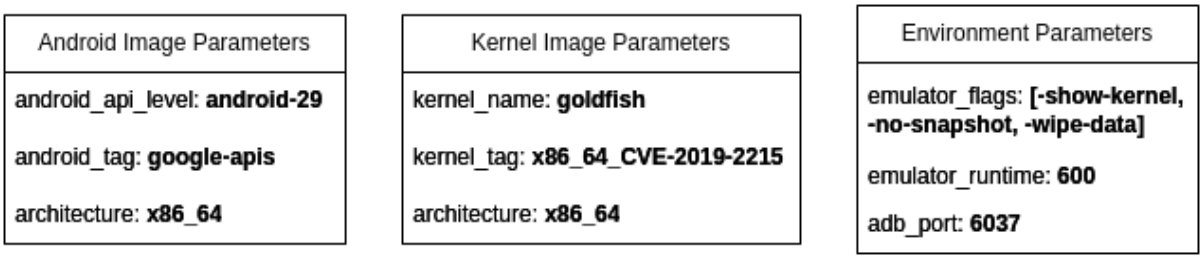

Figure 5.4: Visual representation of the parameters for CVE-2019-2215.

The *android image parameters* reflect the android api level, variant and architecture for the corresponding *sdkmanager* package, being Android 10 (API level 29) with Google APIs access. The *kernel image parameters* specify the repository for the kernel Goldfish Linux Kernel, a kernel for running in emulated platforms. The *kernel_tag* was chosen to be an unique way to identify the kernel in the system, associating it with the CVE number. Both the *architecture* parameters are *x86_64*, so it can run Android Emulator in a machine with a CPU with x86 64 bits architecture that was used in this experiment.

The *environment parameters* are used to specify the running device specifications. The *emulator_runtime* identifies that Android Emulator will run for 600 milliseconds (10 minutes) before its killed, and *adb_port* tells that the *adb* server that allows connection to the device will run in port 6037. The *-show-kernel* flag specified that Android Emulator should display the kernel messages, so the triggered vulnerability can be observed through the kernel logs. The *-no-snapshot* and *-wipe-data* specify that the device should not autosave on boot and should wipe and reset the user image (partition) on boot, respectively. These last two flags were mainly put for testing the system behavior with the *emulator_flags* parameter and for avoiding consuming more memory while booting.

5.3.2 Kernel Scripts

Figure 5.5 shows the structure for the kernel scripts for the CVE-2019-2215 inside the *kernel script repository*, named *kernel_repo*, that is copied in the Docker container *kernel_compiler* when it starts. This script structure follows the described structure in Section 4.3.3, being executed after the kernel repository is clonned.

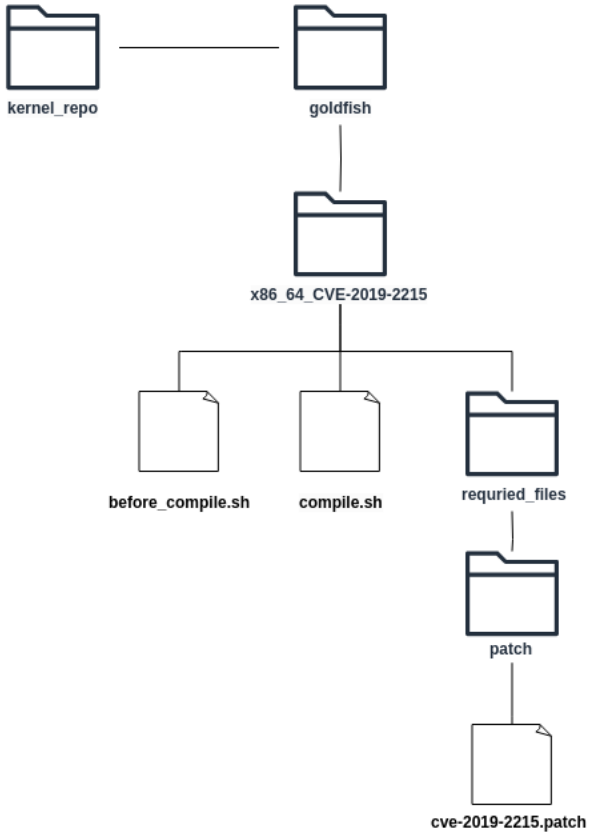

Figure 5.5: Representation of the kernel scripts directory provided for CVE-2019-2215.

The *before_compile.sh* script (Listing 5.4) changes the kernel repository to a specific commit (line 11) so it's in the correct 4.14 version for the vulnerability (Goldfish 4.14), while also doing an unpatch (reverting a portion of code to an older version) in the *binder.c* file (line 15) with the *cve-2019-2215.patch* patch file.

```
1  #!/bin/bash
2  set -e
3
4  # The following env variables contains important directory information
5  # $ROOT_DIR is the directory where the kernel will be cloned
6  # $REQUIRED_FILES_DIR is the directory where your kernel's required_files
       should be present.
7  #     Please copy these files to their correct location in the $ROOT_DIR
8  # $OUTPUT_IMAGE_PATH is where the image should be copied after compiling
9
10 #Set config env varibles
11 COMMIT='182a76ba7053af521e4c0d5fd62134f1e323191d'
12
13 # Change to correct version
14 git checkout $COMMIT
15 git apply $REQUIRED_FILES_DIR/patch/cve-2019-2215.patch
```

Listing 5.4: The *before_compile.sh* script for CVE-2019-2215

Then, the *compile.sh* scripts (Listing 5.5) configures and compiles the kernel using different options and flags. The compilation enables KASAN for the kernel, a dynamic memory error detector tool (Kernel Development Community, 2023), that will detect if the use-after-free was possible through showing a message in the device logs.

Both scripts use environment variables to work better inside the Docker container, so the person writing the scripts can use the environment variables inside the container in a generic way. These variables are automatically populated when the container for compiling the kernel is generated.

```
#Set this var as the location where the image will be after compiling
COMPILED_IMAGE_PATH=$ROOT_DIR/arch/x86/boot/bzImage

#Set kernel config variables
export ARCH=x86_64
export CLANG_PREBUILT_BIN=prebuilts-master/clang/host/linux-x86/clang-
    r377782b/bin
export BUILDTOOLS_PREBUILT_BIN=build/build-tools/path/linux-x86
export CLANG_TRIPLE=x86_64-linux-gnu-
export CROSS_COMPILE=x86_64-linux-gnu-
export LINUX_GCC_CROSS_COMPILE_PREBUILTS_BIN=prebuilts/gcc/linux-x86/x86/
    x86_64-linux-android-4.9/bin

cp $ROOT_DIR/arch/x86/configs/x86_64_ranchu_defconfig $ROOT_DIR/arch/x86/
    configs/new_defconfig

$ROOT_DIR/scripts/config --file $ROOT_DIR/arch/x86/configs/new_defconfig \
      -e CONFIG_KASAN \
      -e CONFIG_KASAN_INLINE \
      -e CONFIG_TEST_KASAN \
      -e CONFIG_KCOV \
      -e CONFIG_SLUB \
      -e CONFIG_SLUB_DEBUG \
      -e CONFIG_SLUB_DEBUG_ON \
      -d CONFIG_SLUB_DEBUG_PANIC_ON \
      -d CONFIG_KASAN_OUTLINE \
      -d CONFIG_KERNEL_LZ4 \
      -d CONFIG_RANDOMIZE_BASE \
      -d CONFIG_SECURITY_DROIDSTATS

make new_defconfig

echo "#Compiling the kernel"
make

#Copy image to correct location
if [ -f $COMPILED_IMAGE_PATH ]; then
   cp $COMPILED_IMAGE_PATH $OUTPUT_IMAGE_PATH
fi
```

Listing 5.5: The code for the *compile.sh* script, for compiling the kernel for CVE-2019-2215.

5.3.3 Kernel Image Creation

The *kernel image* is created through the *test_interface_kernel.py* script. This script runs in a Docker container (illustrated in 5.3), using the command described in Listing 5.6, with the execution of the command starting the entire process for the *kernel image creation*.

In line 3, the Docker socket is mounted, allowing the execution of other Docker commands inside the container (Docker From Docker). The configuration file for the database and storage (represented in Listing 5.2) is copied to the container in line 5. There also volumes for accessing the *test_interface_kernel.py* script itself inside the container (line 6) and for sharing a common output path between the container and host machine (line 4).

```
1 docker run --rm --name test_interface_kernel_cont \
2    --network test_builder_network \
3    -v /var/run/docker.sock:/var/run/docker.sock \
4    -v /tmp/output_path:/usr/src/code/output_path:rw \
5    -v $(pwd)/config.conf:/usr/src/code/config.conf:ro \
6    -v $(pwd)/test_interface_kernel.py:/usr/src/code/test_interface_kernel.
        py \
7    kernel_builder:latest \
8    python3 test_interface_kernel.py
```

Listing 5.6: Command for launching the container that runs *test_interface_kernel.py*.

First, *test_interface_kernel.py* creates an instance of a class *ImageEnvironmentInterface* with the configurations for connecting with the database and storage (Listing 5.7).

```
32    client = interface.ImageEnvironmentInterface.from_config(
33       config=config, db_section="postgres", minio_section="minio"
34    )
```

Listing 5.7: Creating *ImageEnvironmentInterface* class in *test_interface_kernel.py* script in lines 32 to 34.

Then, the *kernel image* is created using the function *create_image()* with the *kernel image parameters* (Listing 5.8).

```
48    client.create_image(
49       type="kernel",
50       name="goldfish",
51       tag="x86_64_CVE-2019-2215",
52       arch="x86_64",
53       output_path=OUTPUT_PATH,
54       output_path_host=OUTPUT_PATH_HOST,
55    )
```

Listing 5.8: Function for creating a *kernel image* executed in *test_interface_kernel.py* in lines 48 to 55.

This function, for the *kernel image* type, implements the steps for creating the *kernel image* described in Chapter 4, in Section 4.4.2. Listing 5.9 shows the code for inserting the image

information in the database, following ***Step 1, Insert image information in the database*** of the *kernel image* creation in Section 4.4.2.

```
338         kernel_image_id = None
339         kernel_result = self._db_pool.select(
340             "kernel_image JOIN build_status USING (build_status_id)",
341             where_values={
342                 "kernel_image_name": kernel_name,
343                 "kernel_image_tag": kernel_tag,
344                 "architecture": arch,
345             },
346             columns=["kernel_image_id", "status"],
347         )
348
349         if kernel_result:
350             if kernel_result[0]["status"] == BuildStatusType.ERROR.value:
351                 # Building an image that runned with an error before
352                 kernel_image_id = kernel_result[0]["kernel_image_id"]
353                 self._logger.warning(
354                     f"Entry for {kernel_tag} exists with status ERROR in
                            database. It will be overriden"
355                 )
356             else:
357                 # Image is already building/completed
358                 raise BuildAlreadyExists(
359                     f'Entry for {kernel_tag} already exists with status {
                            kernel_result[0]["status"]}.'
360                 )
361         else:
362             kernel_image_id = self._db_pool.insert(
363                 "kernel_image",
364                 {
365                     "kernel_image_name": kernel_name,
366                     "kernel_image_tag": kernel_tag,
367                     "architecture": arch,
368                     "build_status_id": building_id,
369                 },
370                 returning=["kernel_image_id"],
371             )
```

Listing 5.9: Code that inserts the *kernel image* information in the database for the *create_image()* function.

Listing 5.10 shows the command for launching the container for compiling the kernel, starting ***Step 2, Launch the container for compiling the kernel (kernel image)*** of Section 4.4.2. The used *kernel image parameters* are passed through environment variables to the container. The output path volume (line 378) is used for saving the image, so its is able to be retrieved from the container compiling the kernel.

```
374     self._docker_client.run_container(
375       "kernel_compiler",
376       container_name=container_name,
377       volumes=[
378           f"{output_path_host}:{ImageEnvironmentInterface.
                KERNEL_COMPILER_OUTPUT_DIR}:rw",
379       ],
380       env_variables=[
381           f"KERNEL_NAME={kernel_name}",
382           f"KERNEL_TAG={kernel_tag}",
383       ],
384       remove=False,
385       detached=True,
386     )
```

Listing 5.10: Code that launches the Docker container for compiling the kernel in the *create_image()* function.

The container for compiling the kernel *kernel_compiler*, when started, executes a script *start.sh* that clones the repository from the Android kernel repository using the *kernel_name*. It then executes the scripts in the *kernel script repository* corresponding the desired kernel (described in Section 5.3.2). After the compiling is done, the kernel should be in the output path directory (Listing 5.11).

```
34 echo "Cloning kernel repository for $KERNEL_NAME"
35 git clone https://android.googlesource.com/kernel/$KERNEL_NAME $ROOT_DIR
36
37 echo "Copying required files to $REQUIRED_FILES_DIR"
38 cp -r /root/kernel_repo/$KERNEL_NAME/$KERNEL_TAG/required_files
       $REQUIRED_FILES_DIR
39
40 echo "Copying scripts to $ROOT_DIR"
41 cp -r /root/kernel_repo/$KERNEL_NAME/$KERNEL_TAG/*.sh $ROOT_DIR
42
43 if [ -f /root/kernel_repo/$KERNEL_NAME/$KERNEL_TAG/before_compile.sh ];
       then
44    echo "Executing before_compile.sh"
45    bash $ROOT_DIR/before_compile.sh
46 fi
47
48 echo "Executing compile.sh"
49 bash $ROOT_DIR/compile.sh
50
51 if [ -f /root/kernel_repo/$KERNEL_NAME/$KERNEL_TAG/after_compile.sh ]; then
52    echo "Executing after_compile.sh"
53    bash $ROOT_DIR/after_compile.sh
54 fi
```

Listing 5.11: The *start.sh* script that is executed when the *kernel_compiler* container is launched

After launching the container, the function inspects the container to see if it finished running, while also verifying if the compilation process did not reach the timeout (Listing 5.12). After the container stops, the exit code is checked to see if there was any errors in the kernel compilation.

```
395     status = self._docker_client.container_status(container_name=
            container_name)
396     time_spent = 0
397     while status["Running"]:
398         if time_spent >= ImageEnvironmentInterface.KERNEL_BUILD_TIMEOUT:
399             status["ExitCode"] = 124
400             status["Error"] = (
401                 f"Image build timeout of {ImageEnvironmentInterface.
                        KERNEL_BUILD_TIMEOUT}(s) reached for "
402                 f"{kernel_name}_{kernel_tag}"
403             )
404             self._logger.error(status["Error"])
405             break
406
407         self._logger.debug(
408             f"Image for compiling {kernel_name}_{kernel_tag} is still running.
                     "
409             f"Retrying in {ImageEnvironmentInterface.
                    KERNEL_COMPILER_RETRY_TIME} seconds"
410         )
411
412         time.sleep(ImageEnvironmentInterface.KERNEL_COMPILER_RETRY_TIME)
413         status = self._docker_client.container_status(container_name=
                container_name)
414         time_spent += ImageEnvironmentInterface.KERNEL_COMPILER_RETRY_TIME
415
416     build_status_id = None
417     exit_code = status["ExitCode"]
```

Listing 5.12: Code that periodicly checks the container *kernel_compiler* in the *create_image()* function.

After compiling the kernel, it is saved in the output path directory with the filename *kernel_tag* (***Step 3, Storing the image in the object storage*** of Section 4.4.2), then is retrieved and saved in MinIO storage (Listing 5.13). The database is updated with the status *COMPLETED* for the image and then the container is removed.

```
425     with open(os.path.join(output_path, kernel_tag), "rb") as f:
426         kernel_bytes = io.BytesIO(f.read())
427         self._minio_client.store_file(
428             file_id=kernel_tag, file_bytes=kernel_bytes
429         )
```

Listing 5.13: Code that stores the compiled kernel image in the storage in the *create_image()* function.

After executing this process, the script ends and the *kernel image* for the CVE-2019-2215 is successfully created in the database and storage, being able to be retrieved later for using it with Android Emulator container. The containers used for running the script and for compiling the kernel are removed after finishing this process.

5.3.4 Android Image Creation

The creation for the *android image* is very similar to the creation of the kernel image, so some details already described in the previous section will be omitted. The creation starts with the execution of the *test_interface_android.py* being executed in a Docker container (Listing 5.14).

```
1 docker run --rm --name test_interface_android_cont \
2    --network test_builder_network \
3    -v /var/run/docker.sock:/var/run/docker.sock \
4    -v /tmp/output_path:/usr/src/code/output_path:rw \
5    -v $(pwd)/test_interface_android.py:/usr/src/code/test_interface_android.
        py \
6    -v $(pwd)/config.conf:/usr/src/code/config.conf:ro \
7    kernel_builder:latest \
8    python3 test_interface_android.py
```

Listing 5.14: Command for launching the container that runs *test_interface_android.py*.

Similar to the *kernel image* creation, an instance of a class *ImageEnvironmentInterface* is created in the *test_interface_android.py*, with the connections for the database and storage. Then, the *android image* is created through the *create_image()* function using the *android image parameters*, with the *name* parameter corresponding to the *android_api_level* (Listing 5.15).

```
53    client.create_image(
54       type="android",
55       name="android-29",
56       tag="google_apis",
57       arch="x86_64",
58       output_path=OUTPUT_PATH,
59       output_path_host=OUTPUT_PATH_HOST,
60    )
```

Listing 5.15: Function for creating a *android image* executed in *test_interface_android.py* in lines 53 to 60.

Starting with ***Step 1, Insert image information in the database*** for the *android image* creation process described in Section 4.4.1, the function inserts the image information in the database (Listing 5.16).

```
android_image_id = None
   android_result = self._db_pool.select(
      "android_image JOIN build_status USING (build_status_id)",
      where_values={
         "android_api_level": android_api_level,
         "android_image_tag": android_tag,
         "architecture": arch,
      },
      columns=["android_image_id", "status"],
   )

   if android_result:
      if android_result[0]["status"] == BuildStatusType.ERROR.value:
         # fetching an image that runned with an error before
         android_image_id = android_result[0]["android_image_id"]
         self._logger.warning(
            f"Entry for {image_identifier} exists with status ERROR in
               database. It will be overriden"
         )
      else:
         # image is already building/completed
         raise BuildAlreadyExists(
            f"Entry for the image {image_identifier} already exists with
               status"
            f'{android_result[0]["status"]}'
         )
   else:
      android_image_id = self._db_pool.insert(
         "android_image",
         {
            "android_api_level": android_api_level,
            "android_image_tag": android_tag,
            "architecture": arch,
            "build_status_id": building_id,
         },
         returning=["android_image_id"],
      )
```

Listing 5.16: Code that inserts the *android image* information in the database for the *create_image()* function.

Then, the *create_image()* function launches the *android_fetcher* container, as in ***Step 2, Launch the container and retrieve the android system directory package (android image)*** of Section 4.4.1, for fetching the package with the android system images from the Android SDK repository (Listing 5.17).

```
554     self._docker_client.run_container(
555         "android_fetcher",
556         container_name=container_name,
557         volumes=[
558             f"{output_path_host}:{ImageEnvironmentInterface.
                    ANDROID_FETCHER_OUTPUT_DIR}:rw",
559         ],
560         env_variables=[
561             f"ANDROID_API_LEVEL={android_api_level}",
562             f"ANDROID_IMAGE_TAG={android_tag}",
563             f"ARCH={arch}",
564         ],
565         remove=False,
566         detached=True,
567     )
```

Listing 5.17: Code that launches the Docker container for retrieving the android package in the *create_image()* function.

The *android_fetcher* container, when created, executes the *start_android_fetcher.sh* script that retrieves the package for the android system images with *sdkmanager*, then compresses it in a zip file that is moved to the output path (Listing 5.18).

```
10 IMAGE_DIRNAME=$ANDROID_API_LEVEL'_'$ANDROID_IMAGE_TAG'_'$ARCH
11 OUTPUT_PATH=/root/output/
12
13 echo "Fetching image from android sdk repository"
14 sdkmanager --install "system-images;$ANDROID_API_LEVEL;$ANDROID_IMAGE_TAG;
       $ARCH"
15
16 echo "Ziping image to $OUTPUT_PATH/$IMAGE_DIRNAME.zip"
17 mv system-images/$ANDROID_API_LEVEL $OUTPUT_PATH
18 cd $OUTPUT_PATH
19 zip -r $IMAGE_DIRNAME.zip $ANDROID_API_LEVEL
```

Listing 5.18: The *start_android_fetcher.sh* script that is executed when the *kernel_compiler* container is launched.

The container is inspected from time to time to see if it is still running (Listing 5.19). When the container stops, the exit code is inspected to see if the image was able to be created.

```
574     status = self._docker_client.container_status(container_name=
            container_name)
575     time_spent = 0
576     while status["Running"]:
577         if time_spent >= ImageEnvironmentInterface.ANDROID_BUILD_TIMEOUT:
578             status["ExitCode"] = 124
579             status["Error"] = (
580                 f"Image build timeout of {ImageEnvironmentInterface.
                        ANDROID_BUILD_TIMEOUT}(s) reached "
581                 f"for {image_identifier}"
582             )
583             self._logger.error(status["Error"])
584             break
585 
586         self._logger.debug(
587             f"Image for fetching {image_identifier} is still running. "
588             f"Retrying in {ImageEnvironmentInterface.
                    ANDROID_COMPILER_RETRY_TIME} seconds"
589         )
590 
591         time.sleep(ImageEnvironmentInterface.ANDROID_COMPILER_RETRY_TIME)
592         status = self._docker_client.container_status(container_name=
                container_name)
593         time_spent += ImageEnvironmentInterface.ANDROID_COMPILER_RETRY_TIME
594 
595     build_status_id = None
596     exit_code = status["ExitCode"]
```

Listing 5.19: Code that periodicly checks the container *android_fetcher* in the *create_image()* function.

Finally, after the image is finishes being retrieved, it is stored in the storage (***Step 3, Storing the image in the object storage***), using the image identifier (key) and filename specified in Chapter 4.1.3, composed by the *android image parameters*, also removing the Docker containers used in the process. This process completes the creation of the *android image* for the CVE-2019-2215.

```
606     with open(
607         os.path.join(output_path, f"{image_identifier}.zip"), "rb"
608     ) as f:
609         kernel_bytes = io.BytesIO(f.read())
610         self._minio_client.store_file(
611             file_id=image_identifier, file_bytes=kernel_bytes
612         )
```

Listing 5.20: Code that stores the compressed *android image* in the storage in the *create_image()* function.

```
501 image_identifier = f"{android_api_level}_{android_tag}_{arch}"
```

Listing 5.21: the *image_identifier* initialization for the *android image* in the *create_image()* function.

5.3.5 Environment Creation

With both the *android image* and *kernel image* for CVE-2019-2215 created as described in the previous sections, they can be used to create the emulated environment, following the process described in Section 4.5.

This process is done by the script *test_emulator_container.py* (Figure 5.3), that is executed inside a Docker container. The command that initiates the script execution is described in Listing 5.22. The volumes used are similar to the ones described for the *kernel image* creation (Listing 5.6), with the directory *images_path* acting as the volume between container and host machine.

```
1 docker run --rm --name test_emulator_container \
2    --network test_builder_network \
3    -v /var/run/docker.sock:/var/run/docker.sock \
4    -v $(pwd)/config.conf:/usr/src/code/config.conf:ro \
5    -v $(pwd)/test_emulator_container.py:/usr/src/code/
        test_emulator_container.py \
6    -v /tmp/images_path:/tmp/images_path:rw \
7    environment_creator:latest \
8    python3 test_emulator_container.py
```

Listing 5.22: Command for launching the container that runs *test_emulator_container.py*.

After launching, this script creates an instance of *DockerAndroidEmulator* class from the same configuration file for the database and storage (Listing 5.23).

```
31    client = emulator.DockerAndroidEmulator.from_config(
32       config=config, db_section="postgres", minio_section="minio"
33    )
```

Listing 5.23: Creating *DockerAndroidEmulator* class in *test_emulator_container.py* script in lines 31 to 33.

As mentioned earlier, an *ImageEnvironmentInterface* is created as an attribute (named *image_interface_client*) inside *DockerAndroidEmulator*, so the functions for interacting with the *android* and *kernel* images can be used (Listing 5.24).

```
80      image_interface_client = interface.ImageEnvironmentInterface.from_config
            (
81          config=config, db_section=db_section, minio_section=minio_section
82      )
83
84      docker_client = docker.DockerClient()
85
86      return DockerAndroidEmulator(
87          image_interface_client=image_interface_client, docker_client=
                docker_client
88      )
```

Listing 5.24: Creating an *ImageEnvironmentInterface* instance inside the *DockerAndroidEmulator* function *from_config()*, later being set as an class attribute fro *DockerAndroidEmulator*.

The creation of the environment starts with the call of the function *start_android_emulator()* from *DockerAndroidEmulator*. This function executes the process described in Section 4.5, receiving all the specified parameters for the environment (Figure 5.4), with the particularity of receiving a single architecture for both image types for compatibility. Listing 5.25 shows the execution of the function.

```
37      client.start_android_emulator(
38          kernel_name="goldfish",
39          kernel_tag="x86_64_CVE-2019-2215",
40          android_api_level="android-29",
41          android_tag="google_apis",
42          arch="x86_64",
43          output_path=OUTPUT_PATH,
44          output_path_host=OUTPUT_PATH,
45          emulator_flags=["-show-kernel", "-no-snapshot", "-wipe-data"],
46          emulator_runtime=600,
47          adb_port=6037,
48      )
```

Listing 5.25: Execution of the function *start_android_emulator()* in *test_emulator_container.py*.

The function starts by waiting for both the *android image* and *kernel image* to be ready for being retrieved, that is, having the status *COMPLETED* in the database. This is done by the *wait_for_image()* function (Listing 5.26 and Listing 5.27).

```
156     self._image_interface_client.wait_for_image(
157         image_type=interface.ImageEnvironmentInterface.KERNEL_IMAGE_TYPE,
158         image_name=kernel_name,
159         image_tag=kernel_tag,
160         arch=arch,
161         output_path=output_path,
162         filename=kernel_image_id,
163     )
```

Listing 5.26: Execution of the function *wait_for_image()* for the *kernel image*, inside the *start_android_emulator()* function

```
180     self._image_interface_client.wait_for_image(
181         image_type=interface.ImageEnvironmentInterface.ANDROID_IMAGE_TYPE,
182         image_name=android_api_level,
183         image_tag=android_tag,
184         arch=arch,
185         output_path=output_path,
186         filename=f"{android_image_id}.zip",
187     )
```

Listing 5.27: Execution of the function *wait_for_image()* for the *android image*, inside the *start_android_emulator()* function

The *wait_for_image()* from the *ImageEnvironmentInterface* function accomplishes that by checking the database periodically for the images, until a given timeout, for each image type. After the images finished *BUILDING*, the *wait_for_image()* retrieves them from the storage, saving the files in the a output path directory. The images are retrieved from the storage using a key (identifier), being the same as the described in Section 4.1.3.

```
147     android_image_id = f"{android_api_level}_{android_tag}_{arch}"
148     kernel_image_id = kernel_tag
```

Listing 5.28: The image indentifier (key) value for both the image types

```
758     time_spent = 0
759     while time_spent <= timeout:
760         if self.has_image(
761             image_type=image_type,
762             image_name=image_name,
763             image_tag=image_tag,
764             arch=arch,
765             statuses_to_check=[BuildStatusType.BUILDING],
766         ):
767             self._logger.debug(
768                 f"Waiting for image {image_identifier} of type '{image_type}'
                        to finish building"
769                 f"Retrying in {retry_time} seconds"
770             )
771             time.sleep(retry_time)
772             time_spent += retry_time
773         else:
774             break
775     if time_spent >= timeout:
776         raise ImageNotReady(
777             f"The requested image {image_identifier} of type '{image_type}'
                    did not finish building"
778             f"after the given timeout of {timeout} seconds"
779         )
780     if self.has_image(
781         image_type=image_type,
782         image_name=image_name,
783         image_tag=image_tag,
784         arch=arch,
785         statuses_to_check=[BuildStatusType.COMPLETED],
786     ):
787         return self.get_image(
788             image_identifier=image_identifier,
789             output_path=output_path,
790             filename=filename,
791         )
```

Listing 5.29: Logic in the *wait_for_image()* function that checks if the images are ready, then retrieves them from the storage to an output path directory.

Once the images are ready and retrieved in the output path, they are used to launch a Docker container to run Android Emulator, with the command shown in Listing 5.30. This container receives the parameters using environment variables, receiving the identifier for each image type as in Listing 5.28 (lines 765 and 766) and also the path were the images are saved, so they are retrievable by the Android Emulator container (lines 762 and 770).

The container also receives additional flags enabling the devices of the host machine in the container that are used by Android Emulator. The *"–device /dev/kvm"* flag (lines 776 and 777)

enables the Kernel-based Virtual Machine (KVM) device, allowing the needed virtualization. The flag *"–privileged"* (line 778) enables other host machines devices that Android Emulator might access. In the end, the port for *adb* is set by an environment variable (line 769) and exposed by the *"-p"* flag (lines 774 and 775), so the container can be accessed by *adb* from the host machine by the specified *adb_port* parameter.

```
758       self._docker_client.run_container(
759           "android_emulator",
760           container_name=container_name,
761           volumes=[
762               f"{output_path_host}:{DockerAndroidEmulator.
                      DEFAULT_EMULATOR_IMAGES_DIR}:rw",
763           ],
764           env_variables=[
765               f"ANDROID_IMAGE_NAME={android_image_id}",
766               f"KERNEL_IMAGE_NAME={kernel_image_id}",
767               f"ARCH={arch}",
768               f"EMULATOR_FLAGS={emulator_flags_string}",
769               f"ANDROID_ADB_SERVER_PORT={adb_port}",
770               f"IMAGES_PATH={DockerAndroidEmulator.
                      DEFAULT_EMULATOR_IMAGES_DIR}",
771           ],
772           network_name=network_name,
773           additional_flags=[
774               "-p",
775               f"{adb_port}:{adb_port}",
776               "--device",
777               "/dev/kvm",
778               "--privileged"
779           ],
780           remove=False,
781           detached=True,
782       )
```

Listing 5.30: Command that launches the container running Android Emulator in the *start_android_emulator()* function.

When the container starts, it executes the script *start_android_emulator.sh* (Listing 5.31). It starts by uncompressing the *android image* zip file in the *system-images/* directory used by *avdmanager* (line 18). Then, it creates an AVD for the system image (line 27), using the *"$SKD_ID"* variable, that corresponds to the identifier for the *sdkmanager* package for the *android image* (*"android-29;google-apis;x86_64"*). The *adb* server is started (line 30) and Android emulator is executed with the created AVD, kernel image and with the specified emulator flags (lines 35 to 38).

```
17 echo "Unziping android image zip to android system-images path"
18 unzip $IMAGES_PATH/$ANDROID_IMAGE_NAME.zip -d system-images/
19
20 AVD_NAME=$ANDROID_IMAGE_NAME'_'$KERNEL_IMAGE_NAME'_AVD'
21
22 #Gets system image dir path based on the ARCH (gets the dir structure to
       the android image and replaces '/' with ';')
23 SDK_ID=$(find system-images/ -name $ARCH -print | tr '/' ';')
24
25 #echo is to avoid Do you wish to create a custom hardware profile? [no]
26 echo "Creating AVD for the android image $SDK_ID"
27 echo no | avdmanager create avd --name $AVD_NAME -k "$SDK_ID"
28
29 echo -e "\nRunning adb server (no daemon) in background"
30 adb -a nodaemon server start &
31
32 echo "Running emulator command:"
33 echo "emulator $NO_INTERFACE_FLAGS $EMULATOR_FLAGS -kernel $IMAGES_PATH/
       $KERNEL_IMAGE_NAME -avd $AVD_NAME"
34
35 emulator $NO_INTERFACE_FLAGS \
36    $EMULATOR_FLAGS \
37    -kernel $IMAGES_PATH/$KERNEL_IMAGE_NAME \
38    -avd $AVD_NAME
```

Listing 5.31: Command that launches the container running Android Emulator in the *start_android_emulator()* function.

After the container for the device starts, the *start_android_emulator()* function periodically checks if the container is still running without errors. Then, it kills the container after 600 milliseconds, determined by the *emulator_runtime* parameter (Listing 5.32). For this period of time, the device is accessible through *adb*.

```
239     status = self._docker_client.container_status(container_name=
            container_name)
240     time_spent = 0
241     while status["Running"] and time_spent <= emulator_runtime:
242         self._logger.debug(
243             f"Emulator container {container_name} is still running. "
244             f"Checking status in {DockerAndroidEmulator.
                    DEFAULT_EMULATOR_CHECK_TIME} seconds"
245         )
246         time.sleep(DockerAndroidEmulator.DEFAULT_EMULATOR_CHECK_TIME)
247         time_spent += DockerAndroidEmulator.DEFAULT_EMULATOR_CHECK_TIME
248         status = self._docker_client.container_status(container_name=
                container_name)
249
250     if status["ExitCode"] != 0:
251         logs = self._docker_client.container_logs(container_name=
                container_name)
252         error_message = f"Emulator for {container_name} had an error while
                runing. Printing logs:\n{logs}"
253         self._logger.error(error_message)
254         self._docker_client.remove_container(container_name=container_name)
255         raise EmulatorExecutionError(error_message)
256     self._logger.debug(
257         f"Finishing execution and removing container for {container_name}."
258         f"Maximun runtime of {emulator_runtime} reached"
259     )
260     self._docker_client.stop_container(container_name=container_name)
261     self._docker_client.remove_container(container_name=container_name)
```

Listing 5.32: Loop in the *start_android_emulator()* that checks if the container running Android Emulator is running, killing and removing if after the specified time.

5.3.6 Triggering the Vulnerability

Through the process described in the previous sections, the framework implementation was able to generate an container running an vulnerable emulated device that is accessible through *adb*. With the running environment, now its possible to try triggering the vulnerability for CVE-2019-2215. This process involves the following steps: connecting with the device using *adb*, sending the compile trigger program to inside the device and executing it, then retrieve the devices logs to see if the user-after-free situation was able reproduced.

For reproducing this, we use the script *trigger.sh*, represented in Listing 5.33. First, we use the command *adb devices* (line 4), that connects to the *adb* server using port 6037 (specified using the *adb_port* parameter), showing the available devices. Since there is only a single available device running, *adb* connects to it, so the next commands will target the device that is running the vulnerable environment.

The file *cve-2019-2215-trigger*, the compiled executable of the program *trigger.cpp* of Listing 5.3, is moved to the path *"data/local/tmp"* inside the device with the *adb push* command (line 7) and is executed in that path using the *adb shell* command (line 10). After executing the trigger, the device kernel logs are retrieved through the *adb logcat* command, saving them in the local hots machine file *syslog.txt*.

```
1  #!/bin/bash
2
3  # Show devices running, connects to the device
4  ./adb -P 6037 devices
5
6  # Pushes trigger executable in /data/local/tmp inside emulated device
7  ./adb -P 6037 push ./cve-2019-2215-trigger /data/local/tmp
8
9  # Executes trigger inside emulated device
10 ./adb -P 6037 shell "./data/local/tmp/cve-2019-2215-trigger"
11
12 # Fetches kernel logs from device
13 ./adb -P 6037 logcat -d -b kernel > syslog.txt
```

Listing 5.33: The script *trigger.sh* that triggers the vulnerability inside the Android Emulator device running in the Docker container.

After running the script, the message of Listing 5.34 were present, indicating a user-after-free situation was identified by KASAN from the process of the executing CVE-2019-2215 trigger program. This message asserts that the vulnerability of CVE-2019-2215 is indeed present in the generated Android Emulator container environment from the implementation.

```
1  11-24 00:15:54.854 0  0 E      :
       ================================================================
2  11-24 00:15:54.855 0  0 E BUG  : KASAN: use-after-free in
       _raw_spin_lock_irqsave+0x33/0x57
3  11-24 00:15:54.857 0  0 E      : Write of size 4 at addr ffff888043ca80a8 by
        task cve-2019-2215-t/6967
```

Listing 5.34: The portion of *syslog.txt* that indicates an use-after-free message from KASAN.

5.4 LIMITATIONS

This section will describe some of the limitations of the implemented framework.

5.4.1 User Interaction

Since the workflow of the components was implemented using scripts, the parameters are not received by the user, but are directly specified in each script that uses them. The directory containing the scripts for compiling the kernel can still be placed in the *kernel script repository*.

The system is unable to receive parameters from the user in a interactive way, so the user needs to edit the parameters inside the code of each script for generating a different emulated environment. This limits the user interaction, allowing possible errors while editing the direct source code of the scripts, with the addition of needing the to execute the scripts in order and manually resolving errors caused by failed executions in the database and storage.

The specified kernel scripts for compiling the kernel also needs to be directly put inside the kernel repository folder, witch needs to be a known directory location and also being susceptible for human error, for example, the accidental override of an folder container the kernel scripts for another security test. As mentioned in Section 5.3.2, the scripts use environment variables for executing the compilation process correctly inside the container in a transparent manner, implying that the code for compiling the desired kernel for a security test needs to be substantially adapted, while also following the scripts structure specified for the system in Section 4.3.3.

5.4.2 Component Communication

As mentioned in the previous sections, some limitations of the implementation, compared the proposed system, are caused by the missing *Orchestrator* component, as well as the *KernelBuilder* and *EnvironmentCreator* components being implemented as scripts instead of actual, message-receiving components. The implementation of the components using scripts limits the parallel management of the components functionalities for creating the Android Emulator container. While error handling was implemented in the frameworks functions, the error treatment is also limited by the implementation using scripts, that were implemented with complex error handling features for the system as a whole.

5.4.3 Emulation Challenges

Running Android Emulator inside a Docker container may limit the behaviour of the emulated device testing. This behaviour was not fully explored while implementing the component, more specifically in the testing the behaviour of some features in the Android device, like Bluetooth and other network related tools present in an Android device. More testing with different type of vulnerabilities may be needed to improve the framework so its able to properly adapt the environment for different types of security tests.

6 CONCLUSION

In this article, it was provided the specification of the orchestrated system *DroidOrchestrator*, designed to generate customized Android emulated environments using Android Emulator, with a containerized approach. With the system specification, it was provided a working implementation of a system framework that allows the creation of a customized Android Emulator environment in a Docker container.

The implementation allows the customization of both the android versions and the kernel image compilation process for generating a vulnerable, customized device, coordinating Docker container executions for creating the required files while also interacting with a database and storage systems. This concludes that the proposed system implementation is indeed possible and can work as intended, providing valuable information on how to run the emulated device using Android Emulator inside a Docker container.

The limitations of the current implementation were also pointed. Although the solutions for said limitations were not described in this article, their evaluation may serve as interesting subjects for future works involving customized Android environments generation.

This document also provided a case study that generates a vulnerable in Android Emulator inside a container, being able to reproduce the CVE-2019-2215 vulnerability trigger (use-after-free in a binder structure) inside the generated device though interacting with it using *adb*.

REFERENCES

Amazon (2023). What is Containerization? https://aws.amazon.com/what-is/containerization/. Accessed on: 28/11/2023.

Android (2023a). Android Kernel Architecture Documentation. https://source.android.com/docs/core/architecture/kernel. Accessed on: 21/11/2023.

Android (2023b). Android Open Source Project. https://source.android.com/. Accessed on: 15/11/2023.

Android for Developers (2023a). Android ADB. https://developer.android.com/tools/adb. Accessed on: 21/11/2023.

Android for Developers (2023b). Android avdmanager. https://developer.android.com/tools/avdmanager. Accessed on: 21/11/2023.

Android for Developers (2023c). Android Emulator Documentation. https://developer.android.com/studio/run/emulator. Accessed on: 21/11/2023.

Android for Developers (2023d). Android sdkmanager. https://developer.android.com/tools/sdkmanager. Accessed on: 19/11/2023.

Android for Developers (2023e). Android Studio Documentation. https://developer.android.com/studio. Accessed on: 21/11/2023.

Android for Developers (2023f). Start the emulator from the command line. https://developer.android.com/studio/run/emulator-commandline. Accessed on: 15/11/2023.

Android for Developers (2023g). What's API Levels? https://developer.android.com/guide/topics/manifest/uses-sdk-element/. Accessed on: 19/11/2023.

Capone, D., Caturano, F., Delicato, A., Perrone, G., and Romano, S. P. (2022). Dockerized Android: a container-based platform to build mobile Android scenarios for Cyber Ranges. *arXiv:2205.09493*.

Costa, G., Russo, E., and Armando, A. (2022). Automating the Generation of Cyber Range Virtual Scenarios with VSDL. *arXiv:2001.06681*.

Docker (2023a). Docker compose documentation. https://docs.docker.com/compose/. Accessed on: 15/11/2023.

Docker (2023b). Docker Hub. https://hub.docker.com/. Accessed on: 21/11/2023.

Docker (2023c). Docker Overview. https://docs.docker.com/get-started/overview/. Accessed on: 15/11/2023.

Docker (2023d). What is a Container? https://www.docker.com/resources/what-container/. Accessed on: 28/11/2023.

eLinux Community (2023). Android Binder Documentation. https://elinux.org/Android_Binder. Accessed on: 28/11/2023.

Kernel Development Community (2023). Kernel Address Sanitizer (KASAN) Documentation. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html. Accessed on: 28/11/2023.

Microsoft (2023). Use Docker or Kubernetes from a container. https://code.visualstudio.com/remote/advancedcontainers/use-docker-kubernetes. Accessed on: 15/11/2023.

MinIO (2023). MinIO, High Performance Object Storage for Modern Data Lakes. https://min.io/. Accessed on: 22/11/2023.

MITRE Corporation (2023). The CVE Project. https://cve.mitre.org/. Accessed on: 28/11/2023.

National Cyber Security Centre (NCSC) (2023). Understanding Vulnerabilities. https://www.ncsc.gov.uk/information/understanding-vulnerabilities. Accessed on: 28/11/2023.

PostgreSQL (2023). What is PostgreSQL? https://www.postgresql.org/about. Accessed on: 22/11/2023.

Red Hat (2023). What is the Linux Kernel? https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel. Accessed on: 21/11/2023.

APPENDIX A – ANDROID VERSIONS FOR SDKMANAGER

A.1 POSSIBLE PARAMETER VALUES FOR SDKMANAGER SYSTEM PACKAGES

In Table A.1 are listed the possible values for the parameters (*android_api_level, android_tag, architecture*) composing the system directory package name format for *sdkmanager*, that is: *"system-images;android_api_level;android_tag;architecture"*.

All the data was extracted using the command *"sdkmanager –list"* in *sdkmanager* with *Android SDK command-line tools* version 8.0. Not all combinations of this parameters are a valid system directory packages, for example, using the *mips* value for architecture, the only packages available are *"system-images;android-16;default;mips"* and *"system-images;android-17;default;mips"*, so a valid combination of parameters must be selected based on the list given by *sdkmanager*, with 200 available system directory packages in total.

Android API Levels	Android Tags		Architectures	
android-10		android-automotive		arm64-v8a
android-14		android-desktop		armeabi-v7a
android-15		android-tv		mips
android-16		android-wear		x86
android-17		android-wear-cn		x86_64
android-18		aosp_atd		
android-19		default		
android-21		google_apis		
android-22		google_apis_playstore		
android-23		google_atd		
android-24		google-tv		
android-25				
android-26				
android-27				
android-28				
android-29				
android-30				
android-31				
android-32				
android-33				
android-33-ext4				
android-33-ext5				
android-34				
android-34-ext8				
android-TiramisuPrivacySandbox				
android-UpsideDownCakePrivacySandbox				

Table A.1: Possible values for Android API Level, Android Tag and Architecture for the system directory packages for *skdmanager* in *Android SDK command-line tools* version 8.0

A.2 ANDROID API LEVEL AND ANDROID VERSION CORRELATION

In Table A.2, there will be the corresponding Android Versions for each Android API Level (Android for Developers, 2023g).

Android API Level	Android Version	Code Name
android-34	Android 14	UPSIDE_DOWN_CAKE
android-33	Android 13	TIRAMISU
android-32	Android 12	S_V2
android-31	Android 12	S
android-30	Android 11	R
android-29	Android 10	Q
android-28	Android 9	P
android-27	Android 8.1	O_MR1
android-26	Android 8.0	O
android-25	Android 7.1 to 7.1.1	N_MR1
android-24	Android 7.0	N
android-23	Android 6.0	M
android-22	Android 5.1	LOLLIPOP_MR1
android-21	Android 5.0	LOLLIPOP
android-20	Android 4.4W	KITKAT_WATCH
android-19	Android 4.4	KITKAT
android-18	Android 4.3	JELLY_BEAN_MR2
android-17	Android 4.2 to 4.2.2	JELLY_BEAN_MR1
android-16	Android 4.1 to 4.1.1	JELLY_BEAN
android-15	Android 4.0.3 to 4.0.4	ICE_CREAM_SANDWICH_MR1
android-14	Android 4.0 to 4.0.2	ICE_CREAM_SANDWICH
android-10	Android 2.3.3 to 2.3.7	GINGERBREAD_MR1

Table A.2: Android API Levels and their corresponding Android Version ranges