DroidOrchestrator: Container-Based Framework for Automation of Customized Android Security Testing Environment Generation

Dario Simões Fernandes Filho
Universidade Federal do Paraná
dariosfernandes@ufpr.br

Christian Debovi Paim de Oliveira
Universidade Federal do Paraná
christiandpo@gmail.com

Cláudio Torres Junior
Universidade Federal do Paraná
claudio.torres@ufpr.br

André Grégio
Universidade Federal do Paraná
gregio@inf.ufpr.br

ABSTRACT

Mobile systems are occupying an increasingly larger share of personal technology usage by the worldwide population, with Android being one of the most widely distributed operating systems in the market. With that in mind, testing security flaws in the Android environment has become a task of growing importance, considering that these tests should be conducted in a controlled environment. In this article, we propose a Container-Based solution for a system that creates an emulated Android device with customized kernels and Android versions. A working framework was implemented based on the proposed system, which was used to execute a case study using a vulnerability trigger as a security test example.

KEYWORDS

Android, Docker, Orchestration, Malware, Customization, Test

1 Introduction

The increasing use of mobile devices, particularly those with the Android operating system, has made them prime targets for malicious actors. Android's open-source nature and affordability contribute to its prevalence and vulnerability, as it allows access to third-party app stores with less rigorous app review processes. In addition to app-based attacks, rooting exploits system vulnerabilities to gain administrative access, inadvertently granting permissions to other apps and facilitating exploitation. Android vulnerabilities can be exploited through several methods (e.g., memory corruption, privilege escalation). Despite protective measures like antivirus software, code review processes, and OS mechanisms, new vulnerabilities are discovered annually [12].

Attacks leveraging those vulnerabilities on mobile devices have highlighted the need for proper environments to test cybersecurity vulnerabilities and flaws, with significantly fewer platforms available for such purposes [6]. Among mobile operating systems, Android stands out due to its open-source nature and accessibility to a variety of tools and repositories for development and testing. Security testing environments require specific variables and customization to ensure accurate reproduction of scenarios. Isolated and newly created environments are essential, as reused scenarios may not function as intended for security tests [7]. Therefore, generating a proper test environment is a crucial step in security test reproduction.

In this article, we propose a system to create heavily customized Android environments for executing security tests in a mostly automatic way, using a set of parameters. Leveraging the benefits of container virtualization, the proposed system aims to provide a robust solution for creating customized Android environments within a container for security testing. The framework implementation based on the proposed system is tested by generating an Android environment to test a security vulnerability. The contributions of this article include:

- Propose a system for creating a customized Android environment inside a container environment.
- Provide a working framework that allows the implementation of the functionalities of the proposed system.
- Present a case study with a security vulnerability to evaluate the proposed framework.

2 Background

Cybersecurity Testing. Cybersecurity testing involves evaluating possible vulnerabilities or potential threats that can affect computational systems. A vulnerability is a weakness or breach in a system that allows a possible attacker to exploit it to achieve a specific end goal [15], being caused by undetected flaws in a 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 [14]. The CVE project catalogues multiple vulnerabilities for multiple different computational systems, differentiating each vulnerability with a 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.

In the context of vulnerabilities, security testing can be done in many different ways, such as using Application Security Testing (AST). AST involves evaluating the security of a specific software or application by identifying potential vulnerabilities for patching them. Vulnerabilities are identified by techniques like code analysis, penetration testing (simulating real-world attacks to locate vulnerabilities in a software application) and security scanning (scanning for vulnerabilities in the software) [11].

For example, security testing for a vulnerability in the Linux kernel that allows the escalation of privileges for a user (allowing access to a resource that is normally only accessible with higher privilege rights) may involve analyzing the kernel source code and trying to simulate the vulnerability, i.e., reproducing the escalation of privileges for a given user in a controlled environment. Having

the ability to create and safely manage an environment of this type is very important, since the application or system being tested usually requires a very specific version and configuration, described in the CVE details, which is necessary for the reproduction of the vulnerability.

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 [16].

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, being named as Android Common Kernels (ACKs), having similar behavior and implementation to the Linux kernel [4].

Each Android version has a compatible specific 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) [5], allowing users to experiment in the kernel's code and compilation process, including for security or vulnerability related tests.

Considering Android emulation, the two main emulators provided by Android are Android Emulator (also known as Goldfish or Ranchu) and Cuttlefish [1]. While Cuttlefish uses the Android AOSP Common Kernels, Android Emulator has its own kernel tree, the Android Goldfish kernels, that are designed to run using Android Emulator. The Goldfish kernel has one specific version for each Android version[1]. Being able to define highly specific kernel version for each Android device, physical or emulated, is another challenge while considering generating a proper environment for security testing related to the Android kernel.

Containerization. Containerization 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 [2].

Containers are administered by a container engine, that manages resources between the container and the operational system. One of the most well-known container engines is the Docker Engine [9], which allows the creation of containers as isolated processes that have a similar behavior of a Virtual Machine, encapsulating the system dependencies and application from the host machine. The key difference between Virtual Machines and containers is that VMs virtualize hardware, while containers virtualize the operational system (software) [10].

Containerization offers a lot of benefits to running applications. Considering VMs, containers are much lighter resource-wise, demanding less from the host machine [9]. Some of the benefits for applications include: ease of distribution of packages and applications using container images, scalability of the application through running multiple containers with the same service and agility while modifying isolated code and application portions in different containers [2]. Considering security tests, which usually require an isolated and reproducible environment [6], containers may prove to be a great tool for exploring the generation of security test related environment generation, given the low resource footprint required as well as the simplicity for creating new environments, with different configurations.

3 Related Work

Capone et al. [6] proposes *DockerizedAndroid*, a framework and platform that allows the creation of emulated, customized Android devices (using Android Emulator) running in a Docker container. Their proposal aims 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 a User Interface (UI) for interacting with the platform. The work was able to execute a security attack reproduction test in the platform using CVE-2018-7661 [13], 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, Gabriele et al. [7] 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 an 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 known, vulnerable configurations provided by a database of vulnerabilities (National Vulnerability Database), that are also specified for the infrastructure by the VSDL language. They show an example that uses the framework to generate the scripts for a cyber range scenario configuration.

Yamin et al. [20] compile papers published between 2002 and 2018 that address cyber ranges, providing an overview of the design and functionality of them and security testbeds. They describe the main components and features of a cyber range, including the types of scenarios, functions, tools, and architecture. The authors emphasize the importance of cyber ranges in providing a controlled and secure environment for conducting cybersecurity exercises and tests. They also discuss the challenges and limitations of current cyber range systems and suggest ways to improve their design and functionality.

Russo et al. [17] propose a new approach to creating cyber ranges that incorporate mobile security components, consisting of a design that integrates mobile devices, such as smartphones and tablets, into cyber ranges to provide a more realistic and comprehensive training environment. The authors evaluate their design through

a proof-of-concept implementation and demonstrate its ability to provide an effective mobile security training environment.

Andreolini et al. [3] present a framework for assessing trainee performance in cyber range exercises. In the author's context, a cyber range is a simulated network environment used for conducting security assessments and training exercises. The authors present a comprehensive framework that includes the following components: learning objectives, performance metrics, assessment methods, and feedback mechanisms.

Cruz et al. [8] conduct a study on the use of a SCADA (Supervisory Control and Data Acquisition) cyber range to promote active learning in the field of cybersecurity. In this context, a cyber range is a simulated network environment used for conducting security assessments and training exercises. The authors describe a study in which a group of students participated in a guided exploration of a SCADA cyber range, where they were able to actively interact with the environment and discover security vulnerabilities. The authors analyze the impact of this approach on the students' learning, including their understanding of SCADA systems and their ability to identify security vulnerabilities. They also discuss the benefits of using a SCADA cyber range for active learning, including the ability to provide hands-on experience with real-world security scenarios, the opportunity to practice critical thinking skills, and the potential to foster a deeper understanding of complex security concepts. The authors conclude that using a SCADA cyber range for active learning is a promising approach to fostering the development of critical skills in the field of cybersecurity.

It is interesting to note that Cyber Range can vary from one context to another, as can be seen on the different contexts used on the related works presented in this section but ultimately, the main goal of each work is to be able to produce test environments that can provide a highly customized scenario. Having such capability is essential to optmize the process for understanding and creating countermeasure actions to effectively mitigate newly discovered threats in the wild.

4 Proposed System Architecture

In this section, we introduce *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 in Docker containers, allowing them to be easily deployed on different environments.

DroidOrchestrator receives parameters that define specifications for the Android device and its kernel. It then generates necessary image files to create the emulated device, storing their contents in an 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, with this process being customized by the user through a set of automations that specify how the kernel should be compiled, so the resulting kernel image is also customized. This process will be detailed later in this article.

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 through the host machine, which is the location where the Docker containers used by *DroidOrchestrator* are executing. 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 of it, allowing the execution of security tests, such as vulnerability exploitation for example.

The container created with the Android emulated environment has a unique name, and is usable for a determined period of time. Once the specified time finishes, the container is removed and the environment used for the test is destroyed. The user may create a new clean environment, from the same test environment generated previously, after the first one is removed. This ensures that trying different testing scenarios are conducted in a clean generated environment.

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

4.1 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*. Each component and their respective functionalities are outlined next:

- ***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 and make sure their execution works as expected.
- ***KernelBuilder*:** This component is responsible for creating and storing the required image files for running an emulated

android system. It communicates with an 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 framework (detailed in Section 9), the components running as separate instances, 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 on the tests conducted for this paper, we opted to implement the components logic using automation. Each automation has the parameters specified directly in the implementation, with each part executing a step for creating the emulated device container: creating the android image, creating the kernel image and running the emulated device, simulating a real scenario interaction with a user.

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 1.

4.2 Database

When creating an image of a given type is requested, an entry with the received parameters (*android image parameters* or *kernel image parameters*, as shown in Figure 1) is added to the SQL database. *DroidOrchestrator* should know the host, port and database name for the running database service. This database diagram is represented in Figure 2.

There are two main tables: *kernel_image* and *android_image*, that are responsible for storing the received parameters for each image type (detailed in Section 6).

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 building through the *build_status_id* field. The status, in the context of the system, have the following meanings:

- ***COMPLETED*:** Describes that the following image build 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. The component is still creating the requested image and it is yet not present in the object storage.
- ***ERROR*:** The image creation process failed due to an error.

By setting and updating the correct values of *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 7.

4.3 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 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. An example would be if the values for *android_api_level*, *android_image_tag* and *architecture* are, respectively, *android-29*, *google_apis*, *x86_64*, then the key value would be *android-29_google_apis_x86_64*, ensuring a unique key value based on the database unique constraint (Figure 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).

4.4 Docker Containers

As explained before, some of the tasks executed by the system are performed in Docker containers. Each container has a specific behavior when executed. After the container finishes its job, it is completely removed, releasing the used resources of the host machine. Each Docker container and its behavior will be shown 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.
- ***kernel_builder*:** 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*

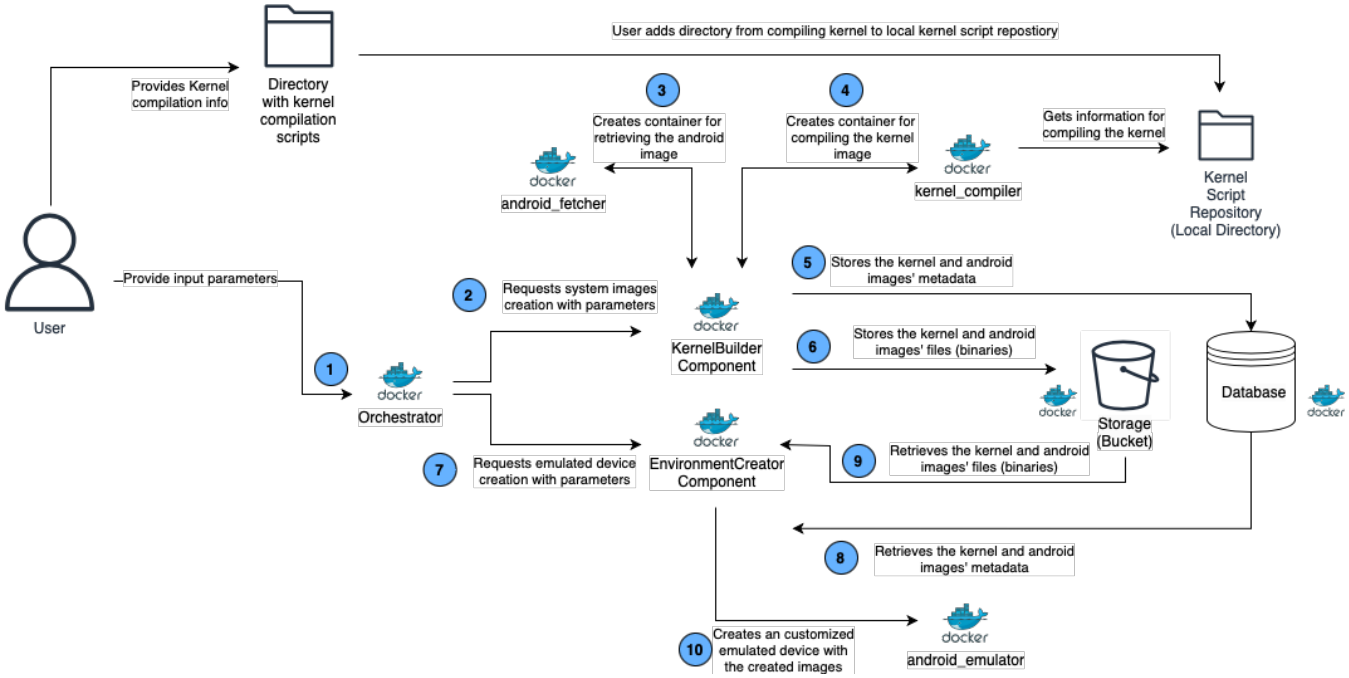

Figure 1: *DroidOrchestrator* system diagram.

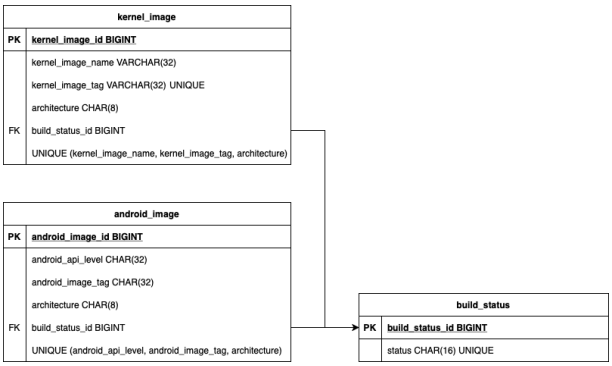

Figure 2: *DroidOrchestrator* database diagram.

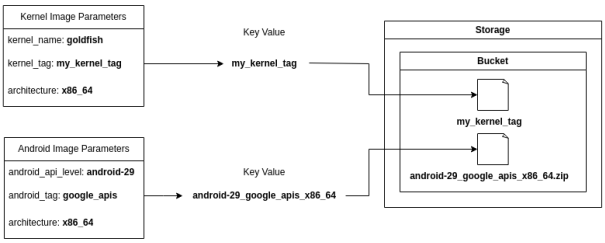

Figure 3: Relation between the received parameters, key value and filename in the *DroidOrchestrator* storage.

and *kernel image* from the storage and running a separate *android_emulator* Docker container.

5 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 other system components.
- ***Step 2, Images Creation*:** The component *KernelBuilder*, reads the parameters passed in step 1 then checks the database if the requested images (android and kernel) exist. If not, *KernelBuilder* launches two other containers, *android_fetcher* and *kernel_compiler*, that create the *android image* and *kernel image*, respectively. Once the images are created, *KernelBuilder* stores the binaries in the storage and their received parameters in the database, removing the used containers used for building the kernel.
- ***Step 3, Environment Creation*:** The component *EnvironmentCreator*, using the received parameters from step 1, checks if the *android image* and *kernel image* are present in the database. If so, *EnvironmentCreator* retrieves the images' binaries from the storage. Once the images are retrieved, an instance of the *android_emulator* container is launched using the image created on step 2. The *android_emulator* runs for a specified period of time, then is removed.

6 Parameter Specification

DroidOrchestrator receives three sets of parameters: *android image parameters*, *kernel image parameters* and *environment parameters* (illustrated in Figure 1). The *android image parameters* and *kernel image parameters* are used in the creation of the *android image* and the *kernel image* by *KernelBuilder*. 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 for the system, so the parameters can be received through a terminal, a configuration file or through a User Interface (UI).

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*.

6.1 Android Image Parameters

The *android image parameters* (*android_api_level*, *android_tag*, *architecture*) correspond to the ones defined in Section 4.3, being used to create a 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 4.2), so the images can be identified and retrieved when necessary.

6.2 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 2) and the locally implemented *kernel script repository*, that will be detailed in Section 6.3.

The *kernel name* or *kernel_image_name* denotes the name of a repository in the Android kernel source tree. For example, *goldfish* refers to the *goldfish* kernel repository[1], which serves as an identifier for fetching the corresponding source code for compilation.

The *kernel tag* or *kernel_image_tag* is a unique string identifier, chosen by the user. This identifier is used as a key for the kernel in the storage (Section 4.3) and as a directory name in the *kernel script repository* (Section 6.3).

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.

6.3 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 build, the user needs to 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, 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 a unique folder name. This folder contains 3 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*). For example, this script can be used to change the branch and commit of the kernel repository, changing the source code version before the compilation process is started. 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 used if any needed command after the kernel finishes compiling, being 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.

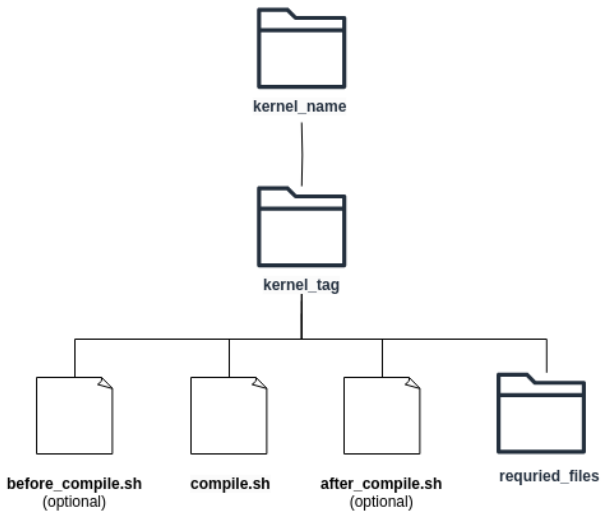

Figure 4: Visual representation of the directory structure that needs to be provided for compiling a kernel using *DroidOrchestrator*.

For making this directory usable by *DroidOrchestrator*, it should be placed inside a known directory in the system, that will be

[1]https://android.googlesource.com/kernel/goldfish

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 5.

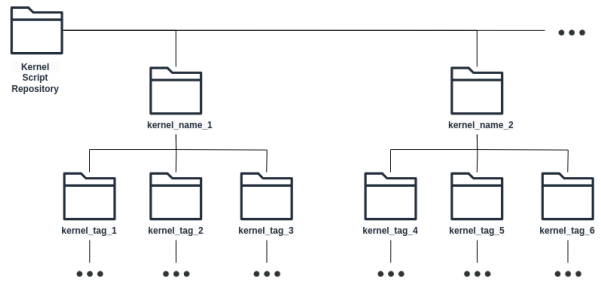

Figure 5: Visual representation of *kernel script repository* structure.

This directory structure will be used by *DroidOrchestrator* for compiling a specific kernel, since the folder path represented by *"kernel_name/kernel_tag"* will 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.

6.4 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.

The *emulator_runtime* is an integer value, in seconds, 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 a given amount of time.

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

7 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 6. In the following, it will be detailed how each type of image is created.

7.1 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 stored 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 6.1). 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 4.3). 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.

7.2 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 4.3). 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 will be updated to *ERROR*.

8 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.

8.1 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 with status *COMPLETED* in the database.

If the image is being build (status is marked as *BUILDING*, *EnvironmentCreator* checks the status in the database from time to time, using a fixed time interval. 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 will return an error. This verification process should be done for both the *android* and *kernel* images before retrieving then from the storage.

8.2 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 container *android_emulator*.

This container starts by unzipping 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 an 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 an *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 will 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.

9 Case Study

In this section, we present a case study of our framework: we automatically generate an emulated Android environment tailored to a specific security test, then deploy and trigger the test within it.

9.1 Security Test Specification

For this case study, we target CVE-2019-2215[2], a local privilege escalation via a use-after-free in Android's Binder IPC driver [18]. The bug was discovered by *syzkaller* in November 2017 and patched upstream in the Goldfish 4.14 development branch in early January 2018.

The root cause lies in `drivers/android/binder.c`: when a process issues the `BINDER_THREAD_EXIT` `ioctl()`, its `binder_thread` structure (which embeds a `wait_queue_head_t`) is freed, but its wait-queue remains registered with any active `epoll`. During cleanup, `epoll` dereferences the freed queue head, allowing an attacker to craft an unlink primitive to overwrite `addr_limit` and gain arbitrary kernel read/write [18].

Listing 1 shows our minimal trigger program, which allocates a `binder_thread`, frees it via `ioctl()` (line 26), and then exits, producing a use-after-free.

```
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 1: Trigger program (`trigger.cpp`) for CVE-2019-2215.

We build the Goldfish 4.14-dev kernel with KASAN enabled. KASAN (Kernel Address Sanitizer) instruments all memory accesses and immediately reports out-of-bounds or use-after-free errors with stack traces, greatly simplifying root-cause analysis [19]. We pair this kernel with Android 10 (Q) + Google APIs. Table 1 summarizes the exact parameters our framework consumes to generate the emulated device.

Our framework's KernelBuilder and EnvironmentCreator components consume these fields to:

(1) Check out the Goldfish 4.14-dev git tree and (un)apply the patch at `drivers/android/binder.c`.
(2) Build a custom kernel image with KASAN.
(3) Fetch and package the Android 10 system image (Google APIs).
(4) Launch an Android Emulator container wired to `adb`.

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

Table 1: Emulated device specifications for reproducing CVE-2019-2215.

Parameter	Value
Android Version	Android 10 (Q) with Google APIs
CPU Architecture	x86_64
Kernel Source	Goldfish 4.14-dev (binder.c unpatched)
Compiled with	Kernel Address Sanitizer (KASAN) enabled
Patch Commit	`7a3cee43e935b9d526ad07f20bf005ba7e74d05b`

This end-to-end automation yields an environment perfectly matched to CVE-2019-2215 testing.

9.2 Triggering the Vulnerability

Once the emulated device is running, we connect via `adb`, push the compiled trigger binary to `/data/local/tmp`, and execute it via `adb shell`. We then collect kernel-level logs with:

```
adb shell dmesg
```

An excerpt is shown in Listing 2, where KASAN reports the use-after-free. This confirms that our generated environment faithfully reproduces CVE-2019-2215.

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

Listing 2: KASAN log showing the use-after-free

10 Limitations

User Interaction. Since the workflow of the components was implemented using automation, parameters are specified directly in each automation rather than being provided directly through a user interface. The directory containing kernel compilation scripts must be placed in the kernel script repository, with the user needing to edit script parameters manually to generate different emulated environments. This lack of user interaction can lead to errors in script editing and manual error resolution in the database and storage. Additionally, kernel scripts must be placed in a known directory, making them prone to human error, such as accidental overrides. The scripts utilize environment variables for the compilation process, requiring substantial adaptation to compile the desired kernel for security tests.

Component Communication. The limitations of the implementation, compared to the proposed system, stem from the absence of the *Orchestrator* component and the *KernelBuilder* and *EnvironmentCreator* components being script-based rather than message-receiving. This restricts parallel functionality management and limits error handling, despite the framework incorporating some error management features, tackling these limitations will be left as *future work*. Note that this doesn't reduce or change the effectiveness of the proposed system, as it will be just an improvement that will make it less error prone.

Emulation Challenges. Running Android Emulator in a Docker container can limit emulated device behavior, particularly with features like Bluetooth and network tools. Further testing with different vulnerabilities is needed to enhance the framework's adaptability for various security tests.

11 Conclusion

The article details the *DroidOrchestrator* system, which creates customized Android emulated environments using a containerized approach. A working framework was implemented to allow customization of Android versions and kernel image compilation in Docker containers, interacting with database and storage systems. The feasibility and functionality of the system was demonstrated through the execution of a case study, along with its limitations. The case study considered successfully generated a vulnerable Android OS in a containerized Android Emulator and reproduced the CVE-2019-2215 vulnerability without issues. We plan on expanding this tool so it can work with other Android OS versions.

REFERENCES

[1] 2net. 2023. Cuttlefish: A Dive into Android 12. https://2net.co.uk/blog/cuttlefish-android12.html. Accessed on: 20/06/2025.

[2] Amazon. 2023. What is Containerization? https://aws.amazon.com/what-is/containerization/. Accessed on: 20/06/2025.

[3] Mauro Andreolini, Vincenzo Giuseppe Colacino, Michele Colajanni, and Mirco Marchetti. 2020. A Framework for the Evaluation of Trainee Performance in Cyber Range Exercises. *Mobile Networks and Applications* 25 (2020), 236–247.

[4] Android. 2023. Android Kernel Architecture Documentation. https://source.android.com/docs/core/architecture/kernel. Accessed on: 20/06/2025.

[5] Android. 2023. Android Open Source Project. https://source.android.com/. Accessed on: 20/06/2025.

[6] Daniele Capone, Francesco Caturano, Angelo Delicato, Gaetano Perrone, and Simon Pietro Romano. 2022. Dockerized Android: a container-based platform to build mobile Android scenarios for Cyber Ranges. *arXiv:2205.09493* (5 2022). http://arxiv.org/abs/2205.09493

[7] Gabriele Costa, Enrico Russo, and Alessandro Armando. 2022. Automating the Generation of Cyber Range Virtual Scenarios with VSDL. *arXiv:2001.06681* (12 2022). doi:10.22667/JOWUA.2022.03.31.0033

[8] Tiago Cruz and Paulo Simões. 2021. Down the Rabbit Hole: Fostering Active Learning through Guided Exploration of a SCADA Cyber Range. *Applied Sciences* (2021).

[9] Docker. 2023. Docker Overview. https://docs.docker.com/get-started/overview/. Accessed on: 20/06/2025.

[10] Docker. 2023. What is a Container? https://www.docker.com/resources/what-container/. Accessed on: 20/06/2025.

[11] HackOne. 2023. What is Security Testing? https://www.hackerone.com/knowledge-center/what-security-testing. Accessed on: 20/06/2025.

[12] Mitre. 2023. Android CVEs. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=android. Accessed on: 20/06/2025.

[13] MITRE Corporation. 2018. CVE-2018-7661. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7661. Accessed on: 20/05/2025.

[14] MITRE Corporation. 2023. The CVE Project. https://cve.mitre.org/. Accessed on: 20/05/2025.

[15] National Cyber Security Centre (NCSC). 2023. Understanding Vulnerabilities. https://www.ncsc.gov.uk/information/understanding-vulnerabilities. Accessed on: 20/06/2025.

[16] Red Hat. 2023. What is the Linux Kernel? https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel. Accessed on: 20/06/2025.

[17] Enrico Russo, Luca Verderame, and Alessio Merlo. 2020. Enabling Next-Generation Cyber Ranges with Mobile Security Components. In *Testing Software and Systems*, Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak (Eds.). Springer International Publishing, Cham, 150–165.

[18] Maddie Stone. 2019. Bad Binder: Android In-the-Wild Exploit. Google Project Zero Blog. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html This post explains the bug (CVE-2019-2215), the discovery methodology, proof-of-concept and exploitation context on Android devices..

[19] The Linux Kernel Documentation Project. [n. d.]. Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html. Accessed: 2025-06-23.

[20] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. 2020. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security* 88 (2020), 101636. doi:10.1016/j.cose.2019.101636