

Federal State Autonomous Educational Institution for
Higher Education National Research University
Higher School of Economics
Tikhonov Moscow Institute of Electronics and Mathematics
BSc Information Security

BACHELOR'S THESIS
Research project
Security Hardening of Containerized Applications

Submitted by Shadrinov Aleksei
student of group BIB201, 4th year of study

Approved by Supervisor
Assistant Professor, V. V. Bashun

Moscow 2024

Contents

Annotation	4
1 Introduction	5
2 Theoretical background	6
2.1 Isolation features	6
2.1.1 Chroot	6
2.1.2 Cgroups	6
2.1.3 Namespaces	7
2.1.4 Capabilities	8
2.2 Runtimes	8
2.2.1 OCI specification	9
2.2.2 Traditional runtimes	9
2.2.3 Alternative runtimes	15
2.3 Vulnerabilities and attack surface	17
2.3.1 Insecure configurations	17
2.3.2 Kernel vulnerabilities	20
2.3.3 Runtime vulnerabilities	20
2.3.4 Application-level vulnerabilities	21
2.4 Container escape	22
2.5 Security hardening	22
2.5.1 Kernel security mechanisms	23
2.5.2 Security features of non-traditional runtimes	26
2.5.3 Scanning tools	27
3 Security analysis of hardening techniques	29
4 Vulnerability detection	30
4.1 Previous studies	30
4.2 Static analysers	30
4.3 Existing tools	31
4.3.1 Clair	31
4.3.2 Trivy	31

4.3.3	Docker Scout	31
4.4	Methodology	31
4.4.1	Images selection	32
4.4.2	Tags selection	32
4.4.3	Scan images	33
4.4.4	Analysis of obtained scan reports	34
5	Conclusion	35
	References	36
A	Clair docker-compose.yaml	47

Annotation

Container virtualization plays a significant role in modern software development practices. The combination of reduced overhead and faster startup time makes this technology advantageous for the industry. However, the growing adoption of containers raises concerns regarding the security of container solutions.

This study examines the issues of security hardening of containerized applications including an analysis of recent container architectures and their security aspects. Actual vulnerabilities and attacks such as container escapes will be discussed. Special attention will also be paid to hardening techniques that provide additional security as well as tools for automated detection and prevention of vulnerabilities.

Аннотация

Технология контейнерной виртуализации прочно заняла лидирующее место в современных практиках разработки и эксплуатации приложений. Сочетание экономии ресурсов и скорости работы делают эту технологию привлекательной для программной индустрии. Однако с распространением контейнеризации вопросы безопасности встают всё более остро.

В данной работе будут рассмотрены вопросы усиления безопасности контейнеризованных приложений. В частности, будет рассмотрена архитектура современных решений в области контейнерной виртуализации и их безопасность. Будут проанализированы актуальные уязвимости и атаки (такие как побег из контейнера) и причины появления данных уязвимостей. Особое внимание будет уделено исследованию механизмов защиты, которые позволяют усилить безопасность контейнеризованных приложений, а также способам автоматизированного обнаружения и предотвращения данных уязвимостей.

Keywords

Docker, Clair, Docker Vulnerability Scanner, Trivy, Container Escape, Container Security.

1 Introduction

The concept of containerization traces its origins back to 1979, when the `chroot` syscall was initially added to Version 7 Unix [1]. Containerization significantly increased in popularity since 2013, when Docker began to dominate the market, and now containers play an essential role in modern software development and distribution practices. Since the technology has become widespread, the security concerns became more evident. As a result, numerous vulnerabilities particularly related to containerized applications were discovered within the popular platforms, and various defensive mechanisms were proposed to address them.

The objective of this paper is to

The rest of the work is organised in the following way.

2 Theoretical background

A container is an isolated process that uses a shared kernel [1]. From the user's point of view, a container may appear similarly to a virtual machine, especially when the process inside the container is a shell. However, containers and virtual machines represent the opposite approaches to virtualization. While a virtual machine typically runs a guest kernel that is separate from the host kernel and resides on top of it, containerized applications usually share the host kernel with the host operating system, host processes and other containers. Nevertheless, containerized applications provide a several layers of isolation, including their own network stack, separate root directory and limited access to host resources. This isolation relies on several Linux kernel features including Linux namespaces, **chroot**, cgroups and capabilities [2].

2.1 Isolation features

2.1.1 Chroot

The first attempts to create an environment similar to modern containers occurred when **chroot** system call was invented. This technology provides root directory isolation, as the process is unable to see or access files outside of the assigned part of file system.

More secure version of the same idea was implemented as **pivot_root** system call and it is primarily used by container runtimes instead of **chroot** [2].

2.1.2 Cgroups

Cgroups was the next feature added to the Linux kernel to achieve container isolation. Designed by Google in 2006, cgroups provide the segregation of computing resources. By assigning a control group to the process developers may limit available memory, CPU, disk and network bandwidth. Essentially, restricting a process inside certain limits prevents it from exhausting all available resources, which may lead to the denial of service attack.

Cgroups are organised in a hierarchy of controllers and could be interacted with by pseudo file system usually present at **/sys/fs/cgroup**. Files and subdirectories inside could be used to adjust limits, and writing process ID to **cgroup.procs** assigns the process to the group [2].

In 2006, version 2 of cgroups was merged to the kernel to address the inconsistency between various controllers. In version 2, process may no longer be assigned different cgroups for different types of resources (controllers), and all threads are grouped together [3].

2.1.3 Namespaces

Linux namespaces were added to the Linux kernel in 2002 in order to virtualize parts of the system as they appear to the groups of processes [4]. Parts of kernel resources can be abstracted by the namespace, and the processes within the namespace interact with their own isolated copy of the global resource. Current versions of the Linux kernel provide namespace isolation for eight types of resources, as described in Table 1 [5].

Table 1 – Linux namespaces

Namespace	Purpose	Version
Mount	Isolates filesystem mount points	2.4.19 (2002)
UTS (Unix Time-sharing System)	Isolates hostname and domain names independently of the hostname of the machine	2.6.19 (2006)
IPC (Inter-process Communication)	Isolate shared memory regions, message queues visible to processes	2.6.19 (2006)
PID (Process ID)	Isolate visible processes, allows PIDs duplication in separate namespaces (including PID 1)	2.6.24 (2008)
Network	Isolate network devices, addresses and routing tables	2.6.29 (2009)
User	Isolate User and Group IDs so that ID presented to process can be mapped to different ID on the host	3.8 (2013)
Cgroup	Isolate the subtree of cgroup hierarchy visible to the process	4.6 (2016)
Time	Isolate system time	5.6 (2020)

Each kind of namespaces may be used separately or in combination with others to provide necessary degree of isolation.

By using namespaces, developers can create environments that are isolated from the host and other processes, as the process cannot modify kernel resources and affect the processes outside the assigned namespace [6]. Furthermore, namespaces add very little overhead and use system resources more efficiently compared to virtual machines. For that reason namespaces are particularly useful for containerization [7].

2.1.4 Capabilities

Finally, capabilities have brought a fine-grained division of privileges than in traditional dichotomy of privileged (User ID 0) and unprivileged processes. Since they were introduced in kernel 2.2, it is possible to assign a thread with required groups of privileges so that it may perform certain sensitive actions in necessary parts of the system [8]. Modern kernel versions provide about 40 capabilities, including possibility to control system time, interact with kernel audit system, manipulate other processes and file permissions or bind to ports with numbers less than 1024.

Containers as well as regular processes can be assigned with various capabilities. Typically, the set of required capabilities for container to successfully run could be significantly reduced, as containers do not need to run administrative tasks. Docker daemon spawns containers with limited privileges, and necessary ones could be added by developers [6]. In addition, Docker provides the `--privileged` flag which grants access to an extended range of privileged activities [9]. It was developed to support Docker-in-Docker scenarios, however, it imposes additional security risks.

2.2 Runtimes

As was shown before, containers are processes with added isolation features. Naturally, it is possible to create an isolated process manually, using `chroot` to isolate root directory and `unshare` to isolate and assign new namespaces, as demonstrated in Listing 1 [2]:

```
mkdir alpine
cd alpine
curl -o alpine.tar.gz
      http://dl-cdn.alpinelinux.org/
      alpine/v3.10/releases/x86_64/alpine-minirootfs-3.10.0-x86_64.tar.gz
tar xvf alpine.tar.gz
cd ..
sudo unshare --user --map-user=0 --uts --mount \
      --net --ipc --pid --fork chroot alpine /bin/sh
/ # /bin/mount -t proc /proc /proc
/ # /bin/ps
PID USER  TIME COMMAND
1 root   0:00 /bin/sh
3 root   0:00 /bin/ps
/ #
```

Listing 1 – Isolated `/bin/sh` process

However, this approach is inconvenient for daily usage. Instead, containers are typically handled via container runtimes. As defined in [10], a container runtime is a software

that runs the containers and manages container images on a deployment node. While this definition is generally true for popular utilities like dockerd, Open Container Initiative (OCI) Runtime specification regulates only the lifecycle of a container [11].

2.2.1 OCI specification

The OCI runtime specification was established in 2015 by Docker, CoreOS and other leaders in the container industry, and it currently includes image, distribution and runtime specifications [12].

According to OCI runtime specification, the user of a compliant runtime must be able to use standard operations, including querying the container state, creating a container from a special set of files (OCI bundle), starting, killing or deleting container [13]. Both compliant and non-compliant implementations exist in the wild, and runc is the reference implementation of the OCI runtime specification [10].

In his blog post, Ian Lewis suggests to differentiate runtimes on a spectrum from low-level to high-level according to additional functionality they are packed with [11]. Indeed, while some of them (runc) have only essential methods to manipulate containers, other runtimes provide API, image management, and may, in fact, rely on runc internally. On this spectrum, we may explore such runtimes as runc, crun, youki, lxc, lsmctfy, containerd, docker, podman, rkt and cri-o.

In addition to traditional containerization (isolation of a process using the kernel mechanisms), researchers distinguish several technologies that bring the strength of virtual machine isolation to process isolation. X. Wang et al. proposed at their study to divide such related technologies into the Unikernel-like and MicroVM-based sandbox container technologies, namely gVisor, Kata containers, Firecracker and Unikernels (Nabla) [14].

2.2.2 Traditional runtimes

runc

runc was initially introduced in 2015 as a separated part of Docker and was presented as “just the container runtime and nothing else” [15]. In fact, runc is a low-level container executor with a very limited set of available features, as runc controls only container lifecycle management. runc is a reference implementation of OCI runtime specification which makes it default choice for many high-level runtime engines.

To run a container with runc, a container bundle must be prepared. Container bundle is a directory which includes config.json file with specification and container root filesystem [16]. The specification file allows users to customize the process environment adjusting the command and arguments, user and group IDs, environment variables, Linux capabilities,

mount points, namespaces and devices. `runc` has a `spec` command for generating this file. It is also possible to generate specifications for rootless containers, which creates a mapping between host and container User IDs [17].

The full list of commands supported by `runc` CLI tool is given below:

- `checkpoint` — checkpoint a running container;
- `create` — create a container;
- `delete` — delete any resources held by the container (often used with detached containers);
- `events` — display container events, such as OOM notifications, CPU, memory, I/O and network statistics;
- `exec` — execute a new process inside the container;
- `kill` — send a specified signal to the container's init process;
- `list` — list containers started by `runc` with the given `-root`;
- `pause` — suspend all processes inside the container;
- `ps` — show processes running inside the container;
- `restore` — restore a container from a previous checkpoint;
- `resume` — resume all processes that have been previously paused;
- `run` — create and start a container;
- `spec` — create a new specification file (`config.json`);
- `start` — start a container previously created by `runc create`;
- `state` — show the container state;
- `update` — update container resource constraints.

Although `runc` has more commands implemented than it is defined in the OCI runtime specification, the implementations of state query, create, start, kill and delete commands are OCI compliant. To deepen the understanding of how runtime works, let us unpack the internals of `runc create` command.

runc create

`runc create` command spawns a container instance from a OCI bundle. The bundle is a directory containing a specification file `config.json` and container root filesystem [18]. Next, we should break down the process of container creation, as implemented by `runc`. The source code of `runc` could be found in <https://github.com/opencontainers/runc>.

Prepare container object The code for the first step is mainly located in `runc/utils_linux.go`. Firstly, `runc` prepares the environment for fresh instance parsing container specification from

`config.json` file in the bundle (`spec, err := setupSpec(context)`). Then, container object is instantiated with `libcontainer` config and `libcontainer.Create` method. This method places the container root filesystem in a new directory with `0o711` permissions, applies cgroup manager (`v1` or `v2`) and returns `Container` object in stopped state.

Next, `startContainer` function starts the runner (`r.run`) with the process arguments defined in `config.json`. The runner is a struct which handles container object and other parameters like `detach` (true or false), `action` (`CT_ACT_CREATE` — create, `CT_ACT_RUN` — create and run) and `init` (false if process must be executed in the existing container).

`r.run` method creates a new `libcontainer.Process` object and initializes IO such as TTY (`stdin`, `stdout`, `stderr`). Finally, depending on the action parameter, the corresponding container method is called (`Start`, `Restore` or `Run`). `run create` calls the first of these methods.

Prepare binary and the parent process Code for the next step is located in the `runc/libcontainer/container_linux.go` file. `Start` function triggers process execution inside the container. Firstly, `c.createExecFifo` creates a FIFO in the root directory, the default path is `/run/runc/<container id>/exec.fifo` with `622` permissions. FIFO is a unidirectional interprocess communication channel that can be accessed as part of the filesystem [19]. Afterwards, the `process` object is passed to the internal container function `start`. This function basically creates a new parent process which helps to spawn a target child process defined in `config.json`. `newParentProcess` clones `/proc/self/exe` (runc binary) in order to protect the original binary from being overwritten. Then the path to runc binary and the first argument (`init`) are concatenated, so that the constructed command is `runc init`. After that, `newParentProcess` calls `newInitProcess` to create an `initProcess` object and pass the namespaces further from `config.json`.

After `newParentProcess` returns the parent process object, `parent.start()` method is called to trigger the parent execution (`p.cmd.Start()`).

Start parent and child processes `p.cmd.Start()` executes a new process (`runc init`). This command argument is handled in the `runc/init.go` file. Firstly, `runc/libcontainer/nsenter` package is imported to handle low-level namespace operations [20]. This import implicitly calls `nsexec()`. This lengthy C function initially accesses the `init` pipe to read bootstrap data (namespace paths, clone flags, uid and gid mappings, and the console path) from the parent process. Then it creates a child process **stage 1: STAGE_CHILD**. This first child process has to unshare all of the requested namespaces, pos-

sibly request parent for user mapping and finally spawn another child process **stage 2: STAGE_INIT** (as mentioned above, pid namespace of calling process must not be changed, so new process is forked to actually enter the new PID namespace) [21]. Latter child process is actually assigned all the required namespaces and it is the only process to return to the Go runtime after final cleanup steps [22].

After stage 2: **STAGE_INIT** child process is ready, `init.go` calls `libcontainer.Init()`, which consequently triggers `startInitialization()`. This function acquires init and log pipes and determines whether `runc exec (linuxSetnsInit)` or `runc create/run (linuxStandardInit)` was issued.

Next, `containerInit` and `linuxStandardInit.Init()` methods are executed. The last function actually does the initialization work such as networking and routing setup, SELinux labeling, console, hostname, AppArmor, sysctl properties, Seccomp. After that, `system.Exec` replaces the running binary image with the one from `config.json`.

Overall, `runc` is a lightweight, OCI compliant, high performance, and low-level container runtime which supports various security features including `seccomp`, `SELinux` and `AppArmor` [23]. `runc` relies on the Linux kernel features such as `cgroups` and namespaces to provide necessary degree of isolation. Although battle-tested, `runc` was discovered to be prone to vulnerabilities multiple times throughout its existence, for example, CVE-2019-5736, CVE-2016-9962 [24] [25].

crun

`crun` is another low-level container runtime that fully implements the OCI runtime specification. Unlike `runc`, `crun` was written in C [26]. According to developers, C suits better for lower level tools like container runtimes [27]. Although `runc` was written in Go, internally it still depends on a special C library, as was discussed previously, to perform low-level manipulations with system objects. On the contrary, JSON processing in `crun` is less elegant compared to the Go realisation.

`crun` is praised to be faster than other runtimes by the developers and researchers, namely Velp et al. [28]. It was demonstrated that `crun` is about 2 times faster than `runc` according to the performance test. `crun` is currently used as the default runtime in Fedora, because `crun` provides full support of `cgroups v2`, default version in Fedora distribution [29].

`crun` command supports the following arguments [30]:

- **checkpoint** — checkpoint a running container;
- **create** — create a container;
- **delete** — delete container definition;

- **exec** — execute a command inside the running container;
- **kill** — send signal to the container init process, **SIGTERM** by default;
- **list** — list containers known to crun;
- **pause** — pause all the container processes;
- **ps** — show the processes running in a container;
- **restore** — restore a container from a checkpoint;
- **resume** — resume the processes in the container;
- **run** — create and immediately start a container;
- **spec** — generate a configuration file;
- **start** — start a container that was previously created;
- **state** — output the state of a container;
- **update** — update container resource constraints.

It can be noted that the list of offered commands is equivalent to the one of **runc**, but **crun** omits **events** command, which can provide runtime information about container.

Overall, **crun** runtime provides the same functionality as **runc**. One of advantages of **crun** is its improved performance thanks to the choice of the programming language. Some vulnerabilities were discovered in **crun**, for example CVE-2021-30465 [31].

LXC

Linux Containers (LXC) is another virtualization technology for running multiple isolated Linux containers sharing a single Linux kernel. It was introduced in 2008 after the appearance of isolation features in the Linux kernel. LXC precedes Docker, which also relies on the same kernel features and was initially built on top of LXC [32].

Unlike OCI-compliant container runtimes, LXC is suitable for running fully packed yet isolated Linux systems [33]. Although all of the LXC tenants share the same kernel and are only isolated by kernel features (cgroups, namespaces, root filesystem isolation, AppArmor and Seccomp), LXC containers resemble a traditional virtual machine. This advantage is beneficial in specific cases, for instance, running an Android emulator inside a container [34].

To spawn LXC containers, it is possible to use **lxc-create** command with the name of the selected Linux template. Other LXC utilities include **lxc-start** which executes the init process in the container and **lxc-attach** which provides a container shell [35].

As illustrated by Flauzac et al., there are a few differences between LXC and other runtimes [36]. LXC leverages User namespace by default to create unprivileged containers. Linux Containers also offer a broader selection of networking approaches, including allocation of physical interface to the container. However, it supports the same security features

as runc, including Capabilities limitation, AppArmor and Seccomp filter.

containerd

containerd is a high-level container runtime which manages both image and container lifecycle [37]. containerd became a separate runtime and replaced Docker Engine's built-in container execution module in Docker 1.11. Since then, containerd remains the default high-level runtime in Docker stack and it is also one of standard runtimes for Kubernetes [38].

containerd interacts with other software including Docker or with users via gRPC API and several command line tools such as `ctr`, `nerdctl` and `crictl` [39]. These tools support traditional operations like pulling or searching images, creating or deleting containers. However, the feature to build container is not included in containerd. To manipulate the containers, containerd relies on runc or other compatible low-level runtimes. For example, to run a container, containerd converts the image to OCI bundle and executes runc binary with necessary parameters [10].

Docker

Docker is one of the most popular and widely used container runtimes. In fact, Docker provides a convenient platform for building, shipping, and running applications in containers. It was developed in 2013 as a container managing platform with LXC as its backend, which was replaced with `libcontainer` in 2014. In 2015, starting with Docker 1.11, Docker Engine was split into several parts:

- `dockerd` — the daemon handling REST API, auth, networking and storage;
- `containerd` — high-level runtime component managing container and image lifecycle;
- `containerd-shim` — a helper per-container component for handling the container process decoupled from the containerd;
- `runc` — low-level runtime that manages OCI containers [32].

This division was performed in order to reduce the amount of platform-dependent code in Docker.

Alongside with the daemon, Docker usually ships with a docker client application that communicates with the daemon, typically through the socket `/var/run/docker.sock` [40]. Therefore, the ability to write to the socket is equivalent to having a full control over Docker daemon, which is a shortcut to attack at the host. For this reason, the access to the socket must be restricted.

As a high-level platform, docker is compatible with various high-level runtimes that act on the containerd-shim level (such as Wasmtime, gVisor and Kata Containers) and low-level runtimes that are runc replacements (`crun`, `youki`) [41].

Podman

Podman is a daemonless container engine for developing, managing, and running OCI Containers on Linux. Podman avoids the client-server architecture implemented by Docker and relies on the classic fork-exec model. The absence of root daemon means that Podman has better support for rootless containers, which allows the execution of containers with user namespace functionality [42]. As a high-level runtime, Podman delegates container execution to runc or other OCI-compatible runtimes. For image building, Podman stack includes a separate tool called Buildah. This tool has ability to create OCI-compatible images defined by Dockerfile without root privileges or daemon [43].

CRI-O

CRI-O is a Go implementation of the Kubernetes CRI (Container Runtime Interface) specification. It is a middle-level runtime similar to containerd and it acts as a middleware between Kubernetes and other OCI-compliant low-level runtimes that directly execute pod containers [10]. CRI-O fully supports runc and Kata Containers and potentially any other OCI runtime.

rkt

rkt (Rocket) is a single-binary Docker alternative, initially released in December 2014 by CoreOS [44]. Designed with security in mind, rkt has several strong features available such as pulling images as a non-root user and image signature verification [45]. In addition, rkt can provision a special lightweight virtual machine for each pod and place the container inside. This approach provides stronger isolation than traditional kernel virtualisation used by other runtimes.

While rkt has several advantages compared to other runtimes, it has been discontinued since 2020 and receives no more development or maintenance [46].

2.2.3 Alternative runtimes

We have already observed various runtimes that provide containerization based on kernel security and isolation features such as cgroups, namespaces and root filesystem isolation. The common flaw of these runtimes is the shared kernel, which poses additional security risks. The following runtimes overcome this disadvantage by moving from host kernel to guest kernel in one way or another.

gVisor

gVisor is an open-source container runtime designed to add an extra layer of isolation between containerized process and the host system. gVisor consists of several components.

One of them, Sentry, is the user space kernel. Sentry listens for incoming system calls from the application, intercepts and implements them by itself, relying on just a limited subset of system calls. Those calls are actually passed to the host. This way, the user application does not interact with the host directly [47].

The other components of gVisor are Gofer, which allows Sentry to access the file system resources, and Netstack, which handles the networking. Additionally, gVisor includes an OCI-compatible runtime called runsc. This runtime can be used by Docker and Kubernetes, making it simple to embed sandboxed gVisor containers into existing flows [48].

Although gVisor does not virtualize system hardware, it still creates an additional layer of protection. However, this approach decreases the performance of applications that frequently issue system calls. Another limitation is that applications which require any system calls not supported by Sentry cannot be ported to gVisor [10].

Kata Containers

Kata Containers is an open source project focused on lightweight virtual machines. Kata Containers feel and perform like containers, but provide the workload isolation and security advantages of VMs. They are designed to be compatible with the OCI specifications and are fully compatible with Docker [49].

Kata Containers run each application within its own lightweight, isolated virtual machine. Kata Containers are compatible with various hypervisors as backends for VMs and rely on several technologies to improve boot time and reduce memory footprint, such as a minimal set of supported devices and pool of pre-configured VMs.

Firecracker

Amazon Firecracker is an extremely lightweight hypervisor for sandboxed applications [50]. On top of this technology cloud services such as AWS Lambda and AWS Fargate were developed. Firecracker uses KVM hypervisor to launch microVMs instances on a Linux host. These microVMs run a special Linux guest operating system with lightweight input/output services and a minimalistic set of software which results in the reduced overhead [47].

Nabla

The Nabla container, developed by IBM, leverages Unikernel technology to minimize the number of system calls accessible. Unikernels are specifically compiled images that combine an application with only the necessary components of an operating system library. This enables the application to run directly on virtual hardware. Currently, Unikernels support a range of programming languages and are compatible with multiple common platforms [51].

To manipulate Nabla containers, a special OCI-compliant command line tool known

as `runnc` could be used [52].

While Unikernels approach provides the desirable degree of isolation comparable to virtual machines, some researchers highlight several issues, such as incompatibility with current VM managing tools, difficulty to debug applications and excessive overhead of nested virtualization when running Nabla containers with hypervisors. A possible mitigation of these challenges was proposed by Williams et al [51]. They suggest a way of running unikernels as processes by leveraging existing kernel system call whitelisting mechanisms (seccomp).

2.3 Vulnerabilities and attack surface

As discussed previously, container virtualization is not a recently developed technology. The origins of containerization could be traced back to the previous century. Despite being battle-tested, containers are often targeted by various cyber attacks, as shown by several studies. Jian and Chen describe two possible attacks that exploit Linux namespace vulnerabilities and memory handling to perform what is known as container escape attack [53]. Another vulnerability mentioned by Bélair et al. allows privilege escalation to the host after rewriting `runc` binary from a maliciously crafted container [54].

In the context of container virtualization, possible attack vectors might be classified into different categories depending on the targeted components. Liz Rice classifies attack vectors based on the life cycle of containers [2]. Through this lens, the attack surface includes vulnerabilities in the application code, insecure configurations during the build and run stages, the security of the hosts where the build and execution stages take place, supply chain integrity, secret handling, container networking, and runtime escaping from their boundaries. Sultan et al. establish a threat model focused on host and container levels of the container life cycle [55]. This taxonomy is based on four directions: protecting container from malicious applications, container from other malicious containers, host and its software from compromised containers and, finally, containers from the host.

As researchers propose multiple taxonomies to understand the security landscape in the context of container virtualization, we will later consider the main sources of threats in more detail.

2.3.1 Insecure configurations

Even though it may seem counterintuitive, security misconfigurations are a major cause of system compromise. According to OWASP, security misconfiguration is among the top 10 security risks for web application [56]. The same conclusion applies to containers.

Root and privileged containers

The majority of container runtimes require root privileges to manipulate containers. Moreover, they instantiate containers with root user inside. For instance, runc and Docker are usually executed with elevated privileges either directly or via Docker daemon, because creating new namespaces in Linux requires the `CAP_SYS_ADMIN` capability [4]. This remains true for the regular configuration of LXC containers [57].

The primary security concern of this approach is that the root user inside the container is the same root user on the host. Because of that, an intruder who manages to move beyond the container isolation will be instantly granted elevated privileges on the host. Another point to consider, in shared environments such as computing clusters in universities regular users should be deprived of elevated privileges and still be able to manage their containers [58].

To mitigate the risk of privilege escalation and establish an additional layer of protection in case of container escape, several approaches are possible.

Set User ID

Firstly, it is possible to set a specific user inside the container with `process.user.uid` parameter (runc; other runtimes have similar options). This way, the containerized process is executed under the specified user and therefore has limited permissions.

Rootless containers

However, in some cases, it is not possible to use a user other than the root user. For example, software designed to run directly on the host cannot be switched to another user ID. For such cases, a container can be deployed in rootless mode. When rootless mode is enabled, the user inside the container is assigned the UID 0 (i. e. considered to be the root) and, at the same time, it is a regular user on the host. This approach was developed on top of user namespaces — a special kind of Linux namespaces providing isolation for UIDs and GID, process root directory and capabilities. According to the documentation, creating a user namespace on recent Linux distributions is an unprivileged operation, and it effectively enables regular users to manage their containers without obtaining excessive privileges [59]. Currently, several runtimes could be installed and used by a regular root user, including Docker, runc, LXC, Podman and other runtimes [57, 60, 61]. However, the configuration for spawning rootless containers is less straightforward and has some limitations such as limited file systems support, Linux security features and networking [60]. Also, user IDs remapping

implies that file permissions on the host must be configured more carefully, as the user inside the container is different from the one on the host [2].

It is worth mentioning that rootless mode in container deployment can successfully prevent the exploitation of certain vulnerabilities. To illustrate, vulnerability CVE-2019-5736 (“Runcescape”) discovered in the reference runtime implementation, runc, offered an intruder the possibility to override **runc** binary and execute arbitrary commands on the host. However, this vulnerability is mitigated by correct enforcement of user namespaces [62]. Other vulnerabilities that could be successfully addressed with rootless mode are mentioned in [58] and [63].

Capabilities

In addition to the non-root/root dichotomy, there is also a way to control container permissions on a more granular level, which is based on Linux capabilities. For example, Docker flag **--privileged** assigns extended privileges to the container including an access to all host devices [9]. Although it may be necessary in certain scenarios such as running Docker inside Docker or passing a device to a container, **privileged** mode is sufficient to escape the container and compromise the host, as illustrated in [64]. To avoid such a risk, fine control of capabilities (when standard privileges are insufficient) is advisable. According to the principle of least privilege, all capabilities should be dropped and then only the necessary ones added.

Another option Docker and runc provide to secure the deployment is **--no-new-privileges** flag. This option prevents a process inside the container from gaining additional privileges after the container has started. For example, programs with the **setuid** option such as **sudo** will have no effect when this option is enabled [65]. Internally, to enable this feature runc enforces the **PR_SET_NO_NEW_PRIVS** flag to the container process [66].

Mounting directories

Another misconfiguration that could potentially lead to serious security issues is allowing containers to access sensitive directories. The NIST Application Container Security Guide states that directories containing system information, such as **/boot** and **/etc**, should never be mounted on a container, as these directories control basic system functionality [67]. To illustrate, the **/etc** directory contains system configurations such as user passwords and cron jobs. Access to the **/bin** directory allows an intruder to override system-wide executables. The **/var/log** directory may contain information about intruder activity. [2].

Mounting Docker daemon sockets is another potentially harmful practice. Sending requests to the Docker socket is equivalent to sending instructions to the daemon, which gives

the user root privileges on the host. The CIS Docker Benchmark recommends auditing access rules and permissions for the socket, enforcing TLS authentication, and avoiding mounting it inside a container [68].

2.3.2 Kernel vulnerabilities

Container virtualization is essentially operating system virtualization when we refer to traditional runtimes like runc or LXC. This implies that the isolation between container processes is accomplished by kernel security measures, primarily namespaces and cgroups. It also implies that each container on the host has access to the same kernel as normal applications.

These circumstances explain why vulnerabilities in the mechanisms involved in process isolation can lead to the container escape and affect the host and other containers. While the Linux kernel has been widely used up until now, it is complex software that is still under development. Several isolation features have been gradually added to the kernel, including the user namespace, which was introduced in version 3.8 of the kernel in 2013. The time namespace (which is not currently regulated by the OCI specification), was developed in version 5.6 of the kernel in 2020. Between these two events, there was also the release of version 4.5 of the kernel with support for cgroups v2 [69–71]. This rate of change exposes certain parts of the kernel to attackers from time to time.

To illustrate, the history of cyber attacks has seen several vulnerabilities that could be exploited to breach container isolation. One such vulnerability is CVE-2022-0492, which was discovered in cgroups v1. Essentially, under certain circumstances, containers can execute arbitrary code with full privileges on the host due to a misconfiguration of the release agent after the termination of a process in the control group [72].

Another recent kernel vulnerability, CVE-2022-0847 (“DirtyPipe”), affects read-only mounted volumes and containers created from the same image. This vulnerability arises due to incorrect access rules for memory page cache. Any unprivileged process can override any file that it has read access to [73].

While the first vulnerability can be addressed with layered security measures, such as activating the seccomp filter, the second vulnerability (“DirtyPipe”) can only be eliminated by updating the kernel and installing security patches [74].

2.3.3 Runtime vulnerabilities

Another key component of container virtualization is the container runtime. This software manages isolated environments, manipulating container images and issuing system calls to create control groups, unshare namespaces and execute new processes.

Despite extensive usage over the years, container runtimes have not been without their share of vulnerabilities, as well as the Linux kernel. We have already mentioned the runc vulnerability CVE-2019-5736 (“Runcescape”), which affected runc, containerd, Docker and other runtimes.

Other runtimes are occasionally reported to be vulnerable, although not all vulnerabilities guarantee the ability for an attacker to escape the container.

2.3.4 Application-level vulnerabilities

Finally, the most common type of vulnerability to observe is at the application level, and these are continuously reported in every piece of software released into the wild. Typically, it is the application running inside a container that is accessible from the outside environment. While the purpose of exposing the application to the internet is to provide a service for external users, it also opens up the possibility for intruders to access the system.

The process of addressing software vulnerabilities involves continuously monitoring whether the software installed in a container contains any known and reported vulnerabilities. This process can be automated using various software scanners, which are typically integrated into the CI/CD pipeline that builds container images.

It is more efficient to scan container images rather than individual containers, as images include all the packages that will be executed in a container. Regular scanning and updating of images are essential for maintaining a secure environment.

The CIS Docker Benchmark suggests scanning images frequently and rebuilding them if necessary to apply security patches, upgrade package versions, and ensure compliance with best practices. This ensures that the containers are always running on the latest, most secure version of the software [68].

When developing containerized applications, special attention should be paid to third-party packages and dependencies. Vulnerabilities in widely used software can be particularly harmful, as they expose entire systems to exploitation. A well-known example of this is the Log4Shell vulnerability, which affected countless systems running Java applications to remote code execution attacks [75].

As can be seen, the process of scanning and updating is similar to a race against an intruder. In order to reduce the chances of a successful attack in the event of a possible breach, it is important to implement in-depth protection for the container environment and the host from compromised applications. Later, we will discuss various techniques for strengthening security in order to achieve additional protection.

Overall, a comprehensive approach to container security requires a multi-layered de-

fense. An administrator must address both insecure configurations in the environment and keep track of emerging vulnerabilities, starting from the host kernel and finishing with the containerized applications.

2.4 Container escape

The container threat model cannot be reduced to a single vector of malicious processes escaping the container. As mentioned in [55], protection should be provided from the container to the host, to other containers, and vice versa. However, our particular interest lies in the case of a container escape, which can be considered as a violation of the core principles of containerization.

The NIST Application Container Security Guide defines a container escape as a situation in which a malicious application running inside a container can attack surrounding containers or the host system [67]. The aim of a container is to isolate the process running inside it from the rest of the system, so any breach of this isolation can have detrimental consequences.

Container escape scenarios may occur due to software vulnerabilities, primarily at the system level. We have previously discussed runtime vulnerabilities such as Runcescape and kernel vulnerabilities (“DirtyPipe”). Application-level vulnerabilities do not directly lead to container escape, as processes are restricted within the container. Nevertheless, these vulnerabilities provide an opportunity for an intruder to initiate an attack.

Another reason for container escapes is insecure configurations. As mentioned previously, running a container with the `--privileged` flag makes escaping trivial. Therefore, it is recommended not only to disable options that are considered risky, but also to harden the configuration in the event of an escape happening. Most importantly, this includes enabling user mapping (rootless mode) or setting user ID, so the process inside the container has limited privileges on the host system. For systems where running processes poses a high risk of escaping, it makes sense to strengthen container isolation and enforce additional security measures [2].

2.5 Security hardening

As can be understood from the above, container virtualization does not provide a perfect level of isolation. Due to the complex mechanisms involved, the entire system is prone to vulnerabilities and misconfigurations at every stage of the container lifecycle. In order to reduce the risk of compromising applications, the host or other containers, it is possible to utilize additional security measures provided by the Linux kernel, runtimes or

third-party developers. In harsh environments, it may be reasonable to enhance protection even further by employing mixed, non-traditional forms of containerization.

2.5.1 Kernel security mechanisms

Most aspects of containerization are managed by the Linux kernel, which makes use of various kernel features to do so. We have already discussed some of these features, such as cgroups, namespaces, and Linux capabilities. Additionally, another important mechanism that helps to reduce the ability of processes to execute malicious instructions is the use of seccomp filters. Furthermore, Linux security modules (LSM) provide an additional layer of protection against potential threats. In addition to these software-based security measures, there are several hardware security technologies that can also be used in conjunction with containers to enhance overall security.

Seccomp

Seccomp (Secure Computing) is a Linux kernel feature that limits the set of system calls a process can execute. This security measure allows running a process in a restricted mode, where only a limited number of system calls are allowed: `read`, `write`, `exit` and `sigreturn` [76]. This way, the process can only read from and write to existing files, return from signal handling and exit (however, it is impossible to issue `open` system call to access new files). This restricted access to system resources prevents malicious software from causing harm to the system or the host.

As this set of system calls is not sufficient for most applications to operate normally, another mode of seccomp was developed. In filtering mode, a Berkeley Packet Filter (BPF) program is used to determine whether a process should be terminated, logged or simply allowed to proceed with the system call [77].

BPF filters are a flexible tool that limits the instructions available to a process to a necessary subset according to a predefined process profile. This feature is particularly suitable for containerized processes, as containers typically perform fixed tasks that do not require the entire set of system calls.

Setting BPF profiles is supported by several container runtimes. Docker recommends using the default seccomp profile, which restricts 44 system calls out of more than 300 [78]. This profile prevents an isolated process from modifying the kernel keyring, adjusting the system time, loading kernel modules, mounting partitions, tweaking swap, rebooting the host and performing other actions that are not typical for containers. runc and crun both support the seccomp feature in OCI bundles [79], and it is possible to configure how to handle specific system calls and vary actions based on arguments of the call [80]. LXC has

support for seccomp sandboxing as well [81].

To take the sandboxing process to the next level, it is recommended to create a customized profile that more strictly restricts system calls. Several tools may assist with this task, such as the `strace` command, `falco2seccomp` or Tracee, which are based on eBPF (Extended Berkeley Packet Filter).

Several security solutions based on the seccomp mechanism have been proposed in the literature. Lei et al. suggested a dynamic approach that automatically traces the system calls made by an application during the booting and running stages, and then applies restrictions in real-time. This approach significantly reduces the number of exposed system calls without causing noticeable performance degradation [82].

Another solution was described by Lopes et al. in 2020. The researchers developed a tool that integrates into the building pipeline and compiles a whitelist profile. It has been shown that implementing a seccomp profile can protect an application from several attacks, including zero-day vulnerability exploitation [83].

Linux security modules

The Linux Security Module is a framework that allows developers to implement new kernel extensions that can perform additional security checks. This is achieved by inserting hooks when a user-space process makes a system call to sensitive kernel objects. The LSM framework is mainly used to enforce a security policy based on Mandatory Access Control (MAC). At the moment, the Linux kernel ships with the following security modules: AppArmor, SELinux, LoadPin, Smack, TOMOYO and Yama.

AppArmor

AppArmor (Application Armor) is a security feature that implements path-based access control and is part of the Linux kernel since 2010. It is included in various Linux distributions, such as Debian and Ubuntu [84].

AppArmor assigns individual executables a policy that limits the set of files, capabilities, and network and memory that an application can access. AppArmor can be used in two modes: learning mode, which traces all the rules in a given profile that an application violates, and enforcing mode, where it prohibits the actions forbidden by the profile. The logs from learning mode can help to adjust the profile so that it matches the typical execution activity. When there are no more violations, AppArmor can be switched to enforcing mode.

Similar to seccomp, AppArmor is a part of the OCI specification. This means that runc and crun, as well as Docker, can execute containers with AppArmor profiles. Docker

even provides a default AppArmor profile that is moderately protective, but widely compatible with various types of software [85]. The same is true for LXC.

In addition to the default profile, there are predefined profiles for popular software prepared by the developers. Additionally, a custom profile can be created using specialized software. In 2021, Zhu et al. proposed an AppArmor profile generator called Lic-Sec. The researchers demonstrated the potential of their tool to protect against zero-day attacks [86].

SELinux

SELinux is another security extension that was merged into the Linux kernel in 2003. This complex technology operates with several key concepts. Firstly, files, processes and applications are labelled with special context information that is stored either as a filesystem attribute or within the kernel. The labels are composed of SELinux users, roles, types and levels. In a basic scenario, only the type is used to determine access rules. Next, there is a policy that allows or denies a process access to other objects based on their shared labels. This allows an application to work with its own files without the permission to modify other parts of the system [87].

The security model described above is known as type enforcement. Another possible policy is multi-level security, although it is not as widely used. SELinux operates in three modes: enforcing, permissive and disabled. Enforcing mode restricts access to only what is permitted by the security policy. Permissive allows applications to bypass rules, but logs any errors, which can be used to adjust the security policy based on the environment. When SELinux is disabled, it is completely turned off.

It is possible to use SELinux with Docker containers, runc, crun and other OCI-compliant runtimes. OCI specifications include a setting for the SELinux label that the containerized process will use. In this case, SELinux must be enabled on the host as well [88]. LXC containers also support this feature.

Other modules

Along with the most widely used security extensions (AppArmor and SELinux), the Linux kernel includes four other ones: LoadPin, Smack, TOMOYO, and Yama.

The LoadPin security module is designed to ensure that kernel files are loaded from the same file system, providing an additional layer of security when that file system is located on a read-only storage. This ensures that the kernel files remain immutable and therefore trustworthy. This approach is not directly related to containers, as it aims to secure the entire system. It is only applicable in simple situations, such as booting from a single read-only image [89].

Yama is another security module with a specific task. It can limit a process's ability to trace the execution of another process, its registers and memory. As a result, a compromised process cannot steal sensitive data from other processes that run with the same permissions [90]. Yama has also been mentioned among other mandatory access control tools that are suitable for securing containers [90].

Tomoyo and Smack are two alternative MAC implementations. Smack, or Simplified Mandatory Access Control Kernel, was released in 2018 and works similarly to SELinux. It also uses a labeling system, but has a simpler interface [91]. Tomoyo was integrated into the kernel in 2009. However, no integration with the OCI specification can be traced for either solution.

Hardware modules

In addition to the software-based security solutions discussed above, literature also mentions hardware-based security features, such as the Trusted Platform Module (TPM) and the Intel Software Guard Extensions (SGX).

TPM (trusted platform module) is a hardware component that verifies the firmware and software running on a device. There are indications that this technology is being used in containerized environments. Benedictis et al. have proposed an integrity verification mechanism for Docker containers based on TPM and Remote Attestation technology [92]. Rocha et al. have also mentioned the possibility of using virtual TPMs in the kernel or in a separate container [93].

The same authors refer to a trusted execution technology on Intel platforms. Intel Software Guard Extensions (SGX) provides a process with a dedicated area in memory that is protected from the kernel and other processes. This technology can be used as an extra layer of security when running containers in an untrusted environment.

2.5.2 Security features of non-traditional runtimes

Above, we have discussed additional kernel security features that can enhance the isolation of containers and protect against various, even currently uncovered vulnerabilities. However, when it comes to kernel vulnerabilities, these security techniques may be ineffective. In such cases, further protection can be found in technologies that blend the distinction between containers and virtual machines.

We have already mentioned several alternative runtime environments. gVisor is described as a user-space kernel because it intercepts system calls and implements them in the user space. Kata containers are executed within a QEMU virtual machine. This idea is further developed in Firecracker, which are lightweight AWS virtual machines with reduced

boot times. Finally, Unikernels compile the application with its system library dependencies, so the resulting image can be run by a hypervisor.

Literature contains numerous studies on the performance of different runtimes, but there is a lack of security analysis in the context of runtimes. In 2020, Flauzac et al. compared LXC/LXD, Singularity, runc, Kata Containers and gVisor in terms of their isolation features. They found that runc and LXC both support all the security features, such as cgroups, capability management, AppArmor and seccomp, and even enforce some of these features by default. In comparison, Kata Containers and gVisor should be given more careful consideration.

Kata Containers support cgroups and capabilities, but other features are not possible or necessary due to the hypervisor virtualization Kata containers are based on. gVisor, on the other hand, uses seccomp-based isolation, so it enables seccomp and capability support by default. It does not support cgroups or AppArmor profiles, as it isolates containers from the host using virtualization of system calls and additional filtering would be redundant [36].

2.5.3 Scanning tools

In order to ensure safety inside the virtual environment, various automation tools can be implemented. As previously discussed, the sources of security threats include mistakes in configuration and vulnerabilities in software, which can reside in the kernel, runtime, or the containerized application itself. To address these aspects of the attack surface, different tools can be used.

Infrastructure benchmarks

To evaluate the security of the entire system, special benchmark tools were developed. These tools are based on the CIS Docker Benchmark, which provides secure configuration guidelines for host configuration, the Docker daemon, configuration files, and images. They also provide guidelines for building and running containers, as well as optional guidelines for Docker Swarm [68].

docker-bench-security is an open-source project on GitHub that contains a script for evaluating the security of current configuration. This bash script checks the system against all recommended configurations in the relevant CIS publication and generates a security report for the user. The project is updated regularly and currently supports version 1.6.0 of the CIS Benchmark [94].

A similar project, docker-bench, is developed by Aqua Security company, which specialises in cloud-native security. This tool was written in Go and currently supports the CIS Docker Benchmark version 1.3.1 [95].

Overall, any of these tools are useful in ensuring that best practices are implemented and that no obvious security issues can be exploited by malicious actors.

The idea of self-assessment of the security of the deployment environment was further developed in 2021 by Sengupta et al. The researchers proposed a tool called Metapod that focuses on the usability and visibility of security measures [96]. The application consists of a backend and frontend, which together allow developers to monitor and adjust current security settings in deployments. For example, the application allows users to control resources, set process fork limits and check the health of containers. Additionally, this tool is integrated with the Snyk vulnerability scanner for checking container images and it also tracks compliance with the CIS Docker Benchmark.

Vulnerability scanners

We have already mentioned Snyk, which is a vulnerability scanner. Scanning tools are an effective mechanism for assessing whether a running application has any known vulnerabilities that could be exploited by an intruder.

To successfully implement vulnerability scanning, several conditions must be met. Firstly, when scanning images and not running containers, it implies the immutability of the containers. This means that no software changes are made during runtime. All libraries, dependencies and code are installed during the build process. Secondly, images need to be scanned regularly, as the database of known vulnerabilities is continuously updated. This ensures that any new vulnerabilities are detected as soon as they are discovered.

There are a number of scanning tools available, both open source and commercial. Trivy, Clair and Anchore are some examples of open source solutions. Companies that offer proprietary solutions include JFrog, Palo Alto and Aqua. Additionally, image registries such as Docker Trusted Registry and Harbor by the CNCF also provide scanning capabilities. Some of these scanners can also provide additional insights, such as detecting malware and identifying insecure configurations during the build process.

Along with static image scanning, researchers have proposed alternative approaches. In 2020, Brady et al. developed a dynamic analysis solution that starts an image for a certain period of time to monitor file changes, network traffic and running processes during container runtime. This allows researchers to detect abnormal behaviour patterns that may indicate potential malicious activity [97].

3 Security analysis of hardening techniques

4 Vulnerability detection

4.1 Previous studies

The researches upon the quality and effectiveness of vulnerabilities detection tools have already been attempted. One of the most comprehensive works was published by Omar Javed and Salman Toor in 2021 [98]. The approach described in the paper was based on several open source scanners (namely, Clair, Anchore, and Microscanner). After inspecting 59 Docker images for Java-based applications, the authors calculate detection coverage and detection hit ratio metrics and conclude that the most accurate tool (Anchore) was omitting around 34% of vulnerabilities. A major limitation of their work is a relatively small number of images used to evaluate the performance of scanning tools as well as the range of scanners examined.

Another paper by K. Brady et al. described the CI/CD pipeline which combined static and dynamic analysers [99]. A set of 7 images was submitted to Clair and Anchore scanners and original dynamic scanner based on Docker-in-Docker approach. The results clearly show the importance of dynamic detection method, however, no direct comparison of Clair and Anchore was conducted. Similar research was conducted in 2021, however, a pair of SonarQube and VirusTotal was used [100].

Massive research was conducted in 2020 by security company Prevasio. As the report states, all 4 million of public images from Docker Hub were scanned with static analyser (Trivy) and dynamic analyser, and more than half of all images were discovered to have critical vulnerabilities [101]. Significant number of images were containing dynamically downloaded payload, cryptominers and other malicious software.

4.2 Static analysers

Vulnerability scanners are special tools that identify known software vulnerabilities inside containers. Two approaches to this issue are known. First, it is possible to analyse versions of all the software in container image and check its safety. This approach is called static analysis, as it does not require any container to run, which speeds up the process. However, static analysis has significant limitations, because the vulnerability databases take time to include recently discovered vulnerabilities, and container must not download any other packages besides what is included in the image. Finally, malicious but not yet reported packages cannot be discovered by static analysers. Nevertheless, this technology is extensively used in CI/CD pipelines to minimize risks of attack and container compromisation.

The opposite way to detect vulnerabilities in containers is to observe its runtime behaviour. Tools implementing this idea are called dynamic analysers. They may gather data about running processes, network and disk usage and discover suspicious activity patterns such as requests to C2 servers or compiling executables. This approach can effectively detect malicious images in the cases when static scanners are powerless. However, they require extensive resources and time usage, as each scan takes tens of seconds to complete, and may be unpredictable in special cases, when malicious applications disguise themselves and imitate normal behaviour unless started in the production environment. Further discussion will be held about static analysers.

The process of image attestation with static analyser may be described as following. First, the scanner compiles the list of all installed packages and libraries inside the image along with their versions. Next, each package is checked upon publically available vulnerability databases and marked as vulnerable or safe to use. This explains why various mistakes could easily occur during the detection process, or why reports from different scanners are not identical. Each scanner gets vulnerability information from various sources including per-distribution security advisories (Ubuntu Oval database, Debian Security Tracker, RHEL Oval database, to name but a few) or general vulnerability databases such as NVD (National Vulnerability Database). The information in these sources are of various degrees of relevance. Special treatment must be provided to the vulnerabilities that will not be fixed due to its negligibility. Finally, names of the packages may vary between distributions. All these reasons lead to false negative results, when a vulnerable package is marked as save. On the contrary, some packages are organised in groups, and incorrect treatment of the whole group because of one package can probably cause false positive results. Overall, the scanning report should be carefully considered in each individual case [2].

4.3 Existing tools

The range of available static vulnerability scanners is wide.

4.3.1 Clair

4.3.2 Trivy

4.3.3 Docker Scout

4.4 Methodology

The process of data acquisition and the following analysis for this part of research could be divided into several stages Firstly, a set of container images and tags for scanning was composed. For this purpose Docker Hub API were employed. Then, each image was

submitted to each scanning tool and vulnerability report was stored on disk in JSON format. After scanning, reports were combined and number of vulnerabilities for each image was calculated. Based on this calculations the metrics of detection quality were aggregated for each scanning tool.

4.4.1 Images selection

To select the most popular (according to the number of pulls) images from Docker Hub, the following algorithm was developed.

At first, we composed a list of search queries. The list consists of combinations of two latin letters starting from **aa** and finishing with **zz**. Each of $26 \cdot 26 = 676$ combinations was then used to query the list of corresponding images from Docker Hub, as shown in Listing 2. The result was then saved to the JSON file.

```
1 def get_page(page, query):
2     url = "https://hub.docker.com/api/content/v1/products/search"
3     params = {
4         "page_size": 100,
5         "q": query,
6         "source": "community",
7         "type": "image",
8         "page": page
9     }
10    response = requests.get(url, params=params)
11    response.raise_for_status()
12    data = response.json()
13    return data["summaries"]
```

Listing 2 – Query images

The next step is to combine search results from each query and sample a reasonably sized subset of the most popular images. We parse each JSON file and drop duplicating images. After all, the set of 3694651 images is sorted by **popularity** parameter and 0.999 percentile is selected. The resulting 3695 images are saved for further processing.

4.4.2 Tags selection

Each image exists in multiple versions which are referred to as image tags. We have to select a number of tags for each image to scan. Tags could also be built for a specific architecture and OS platform. We are primarily interested in **amd64** and **Linux** tags.

The request used for generating the list of tags for each image is demonstrated in Listing 3. We randomly select 10 tags for each image that satisfy mentioned requirements. As the result, the set of approximately 30000 tags is composed and stored in file.


```

1  pref, suf = image.split("/")
2  link = f"https://hub.docker.com/v2/namespaces/{pref}/repositories/{suf}/tags"
3  res = requests.get(link)
4  res = res.json()
5  up = res["count"]
6  iteration = 0
7  total = 0
8  while total < 10 and iteration < 100:
9      page_num = random.randint(1, up)
10     response = requests.get(link, params={"page": page_num, "page_size": 1})
11     tag = response.json()["results"][0]
12     if "amd64" in [i["architecture"] for i in tag["images"]]:
13         tags.append(tag)
14         total += 1
15     iteration += 1

```

Listing 3 – Query tags

4.4.3 Scan images

Next steps were performed on a virtual machine with 16 vCPU and 32 Gb RAM. As a rule, each scanning tool must be installed on the machine, and then each image tag is passed to the tool with command line client.

Clair

To run Clair scanner locally, its Docker images were used from official repository (<https://github.com/quay/clair>). A special docker-compose file listed in Appendix A helps to run database and scanner containers, and `clairctl` command line tool actually pulls the image from Docker Hub and passes it to the scanner. The deployment process is described in the manual (https://quay.github.io/clair/howto/getting_started.html). An example of command which must be executed is listed below.

```
clairctl report --out json fluent/fluent-bit:2.0.8-debug > report.json
```

Listing 4 – Run Clair scanner

Scout

Scout scanner is developed by Docker and shipped as a plugin for the docker CLI tool. Installation process is described in the manual (<https://docs.docker.com/scout/install/>).

To scan image with docker scout, the following command must be issued.

```

docker scout cves --format sarif \
    --output report.json \
    stakater/reloader:SNAPSHOT-PR-586-UBI-0db6f802

```

Listing 5 – Run Docker Scout scanner

Trivy

Trivy scanner is an open source tool developed by cybersecurity company Aqua Security and is a replacement for their previously well-known vulnerability detector, microscanner. The installation process is rather straightforward and it is described in the manual page (<https://aquasecurity.github.io/trivy/v0.50/getting-started/installation/>). To scan an image, trivy tool must be invoked by the following command:

```
trivy image --format sarif -o report.json golang:1.12-alpine
```

Listing 6 – Run Trivy scanner

4.4.4 Analysis of obtained scan reports

After submitting the set of images to each of scanning tool, the reports must undergo the further analysis to determine the effectiveness of the studied software. As the results are presented in JSON format, python libraries such as `json` and `pandas` provide us with enough features to calculate the desired quality metrics.

Quality metrics

https://en.wikipedia.org/wiki/Precision_and_recall

5 Conclusion

References

- [1] R. Osnat, “A brief history of containers: From the 1970s till now,” *aquasec.com*, Jan. 10, 2020. [Online]. Available: <https://www.aquasec.com/blog/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>. [Accessed: May 5, 2024]
- [2] L. Rice, *Container security: Fundamental technology concepts that protect containerized applications*. O’Reilly Media, 2020.
- [3] “cgroups(7) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man7/cgroups.7.html>. [Accessed: May 5, 2024]
- [4] “namespaces(7) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man7/namespaces.7.html>. [Accessed: May 5, 2024]
- [5] M. Kerrisk, “Namespaces in operation, part 1: namespaces overview,” *lwn.net*, Jan. 4, 2013. [Online]. Available: <https://lwn.net/Articles/531114>. [Accessed: May 5, 2024]
- [6] “Docker security,” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/security>. [Accessed: May 5, 2024]
- [7] N. Yang, C. Chen, T. Yuan, Y. Wang, X. Gu, and D. Yang, “Security hardening solution for docker container,” in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2022, pp. 252–257.
- [8] “capabilities(7) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man7/capabilities.7.html>. [Accessed: May 5, 2024]
- [9] “Running containers,” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/reference/run>. [Accessed: May 5, 2024]
- [10] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, “Performance evaluation of container runtimes,” in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, 2020.

- [11] I. Lewis, “Container runtimes part 1: An introduction to container runtimes,” *ianlewis.org*, Dec. 6, 2017. [Online]. Available: <https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r>. [Accessed: May 5, 2024]
- [12] “Open Container Initiative.” [Online]. Available: <https://opencontainers.org/>. [Accessed: May 5, 2024]
- [13] “Open container initiative runtime specification,” *github.com*, May 16, 2023. [Online]. Available: <https://github.com/opencontainers/runtime-spec/blob/main/spec.md>. [Accessed: May 5, 2024]
- [14] X. Wang, J. Du, and H. Liu, “Performance and isolation analysis of runc, gvisor and kata containers runtimes,” *Cluster Computing*, vol. 25, pp. 1497–1513, 2022.
- [15] S. Hykes, “Introducing runc: a lightweight universal container runtime,” *docker.com*, Jun. 22, 2015. [Online]. Available: <https://www.docker.com/blog/runc>. [Accessed: May 5, 2024]
- [16] “runc(8) system manager’s manual,” *man.archlinux.org*. [Online]. Available: <https://man.archlinux.org/man/extra/runc/runc.8.en>. [Accessed: May 5, 2024]
- [17] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García, and P. Orviz, “Enabling rootless linux containers in multi-user environments: The udocker tool,” *Computer Physics Communications*, vol. 232, pp. 84–97, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010465518302042>. [Accessed: May 5, 2024]
- [18] “runc-create(8) system manager’s manual,” *man.archlinux.org*. [Online]. Available: <https://man.archlinux.org/man/runc-create.8.en>. [Accessed: May 5, 2024]
- [19] “fifo(7) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man7/fifo.7.html>. [Accessed: May 5, 2024]
- [20] TheDiveO, “Libcontainer: What happens after the invocation of /proc/self/exe init,” may 10, 2023. [Online]. Available: <https://stackoverflow.com/a/76220640>. [Accessed: May 5, 2024]
- [21] “nsexec.c,” *github.com*. [Online]. Available: <https://github.com/opencontainers/runc/blob/main/libcontainer/nsenter/nsexec.c>. [Accessed: May 5, 2024]

- [22] Terenceli, “runc internals, part 3: runc double clone,” *terenceli.github.io*, Dec. 28, 2021. [Online]. Available: <https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2021/12/28/runc-internals-3>. [Accessed: May 5, 2024]
- [23] R. Kumar and B. Thangaraju, “Performance analysis between runc and kata container runtime,” in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–4.
- [24] Y. Avrahami, “Breaking out of docker via runc — explaining cve-2019-5736,” *unit42.paloaltonetworks.com*, Feb. 21, 2019. [Online]. Available: <https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/>. [Accessed: May 5, 2024]
- [25] “On-entry container attack - cve-2016-9962,” *access.redhat.com*, Jan. 11, 2017. [Online]. Available: <https://access.redhat.com/security/vulnerabilities/cve-2016-9962>. [Accessed: May 5, 2024]
- [26] “opencontainers/runtime-spec — implementations,” *github.com*, Sep. 28, 2021. [Online]. Available: <https://github.com/opencontainers/runtime-spec/blob/main/implementations.md>. [Accessed: May 5, 2024]
- [27] “containers/crun,” *github.com*. [Online]. Available: <https://github.com/containers/crun>. [Accessed: May 5, 2024]
- [28] G. E. de Velp, E. Rivière, and R. Sadre, “Understanding the performance of container execution environments,” in *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, ser. WOC’20. New York, NY, USA: Association for Computing Machinery, 2021, pp. 37–42. [Online]. Available: <https://doi.org/10.1145/3429885.3429967>
- [29] H. Gantikow, S. Walter, and C. Reich, “Rootless containers with podman for hpc,” in *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21-25, 2020, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 343–354. [Online]. Available: https://doi.org/10.1007/978-3-030-59851-8_23
- [30] “crun(1) general commands manual,” *man.archlinux.org*. [Online]. Available: <https://man.archlinux.org/man/crun.1.en>. [Accessed: May 5, 2024]

- [31] “CVE-2021-30465,” may 27, 2021. [Online]. Available: <https://www.cve.org/CVERecord?id=CVE-2021-30465>. [Accessed: May 5, 2024]
- [32] A. Randal, “The ideal versus the real: Revisiting the history of virtual machines and containers,” *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: <https://doi.org/10.1145/3365199>
- [33] S. Senthil Kumaran, *Practical LXC and LXD: linux containers for virtualization and orchestration*. Springer, 2017.
- [34] A. Obradov, S. Usorac, and M. Antic, “Android runtime service optimization for execution inside lxc,” in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2021, pp. 995–997. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9597181>
- [35] “lxc(7) — linux manual page,” *man7.org*, Jun. 16, 2022. [Online]. Available: <https://man7.org/linux/man-pages/man7/lxc.7.html>. [Accessed: May 5, 2024]
- [36] O. Flauzac, F. Mauhourat, and F. Nolot, “A review of native container security for running applications,” *Procedia Computer Science*, vol. 175, pp. 157–164, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S187705092031704X>
- [37] I. Lewis, “Container runtimes part 3: High-level runtimes,” *ian-lewis.org*, Oct. 30, 2018. [Online]. Available: <https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes>. [Accessed: May 5, 2024]
- [38] B. Weissman and A. E. Nocentino, “Installing kubernetes,” in *Azure Arc-enabled Data Services Revealed: Deploying Azure Data Services on Any Infrastructure*, Berkeley, CA: Apress, 2022, pp. 65–87. Accessed: May 5, 2024. [Online]. Available: https://doi.org/10.1007/978-1-4842-8085-0_4
- [39] “Getting started with containerd,” *github.com*, Feb. 28, 2024. [Online]. Available: <https://github.com/containerd/containerd/blob/main/docs/getting-started.md>. [Accessed: May 5, 2024]
- [40] I. Miell and A. Sayers, *Docker in practice, second edition*. Shelter Island, NY: Manning Publications, 2019. Accessed: May 5, 2024. [Online]. Available: <https://www.manning.com/books/docker-in-practice-second-edition>

- [41] “Alternative container runtimes,” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/alternative-runtimes>. [Accessed: May 5, 2024]
- [42] H. Gantikow, S. Walter, and C. Reich, “Rootless containers with podman for hpc,” in *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21-25, 2020, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 343–354. [Online]. Available: https://doi.org/10.1007/978-3-030-59851-8_23
- [43] “containers/buildah,” *github.com*. [Online]. Available: <https://github.com/containers/buildah>. [Accessed: May 5, 2024]
- [44] J. Struye, B. Spinnewyn, K. Spaey, K. Bonjean, and S. Latre, “Assessing the value of containers for nfvs: A detailed network performance study,” in *2017 13th International Conference on Network and Service Management (CNSM)*, 2017, pp. 1–7. [Online]. Available: <https://ieeexplore.ieee.org/document/8256024>
- [45] A. Acharya, J. Fanguède, M. Paolino, and D. Raho, “A performance benchmarking analysis of hypervisors containers and unikernels on armv8 and x86 cpus,” in *2018 European Conference on Networks and Communications (EuCNC)*, 2018, pp. 282–9. [Online]. Available: <https://ieeexplore.ieee.org/document/8443248>
- [46] “rkt,” *github.com*. [Online]. Available: <https://github.com/rkt/rkt>. [Accessed: May 5, 2024]
- [47] Anjali, T. Caraza-Harter, and M. M. Swift, “Blending containers and virtual machines: a study of firecracker and gvisor,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 101–113. [Online]. Available: <https://doi.org/10.1145/3381052.3381315>
- [48] “google/gvisor,” *github.com*. [Online]. Available: <https://github.com/google/gvisor>. [Accessed: May 5, 2024]
- [49] A. Randazzo and I. Tinnirello, “Kata containers: An emerging architecture for enabling mec services in fast and secure way,” in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 209–214. [Online]. Available: <https://ieeexplore.ieee.org/document/8939164>

- [50] X. Wang, J. Du, and H. Liu, “Performance and isolation analysis of runc, gvisor and kata containers runtimes,” *Cluster Computing*, vol. 25, no. 2, pp. 1497–1513, 2022. [Online]. Available: <https://doi.org/10.1007/s10586-021-03517-8>
- [51] D. Williams, R. Koller, M. Lucina, and N. Prakash, “Unikernels as processes,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 199–211. [Online]. Available: <https://doi.org/10.1145/3267809.3267845>
- [52] “nabla-containers/runnc,” *github.com*. [Online]. Available: <https://github.com/nabla-containers/runnc>. [Accessed: May 5, 2024]
- [53] Z. Jian and L. Chen, “A defense method against docker escape attack,” in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, ser. ICCSP ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 142–146. [Online]. Available: <https://doi.org/10.1145/3058060.3058085>
- [54] M. Bélair, S. Laniepce, and J.-M. Menaud, “Leveraging kernel security mechanisms to improve container security: a survey,” in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3339252.3340502>
- [55] S. Sultan, I. Ahmad, and T. Dimitriou, “Container security: Issues, challenges, and the road ahead,” *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. [Online]. Available: <https://ieeexplore.ieee.org/document/8693491>
- [56] “OWASP Top Ten,” *unit42.paloaltonetworks.com*, Oct. 1, 2021. [Online]. Available: <https://owasp.org/www-project-top-ten>. [Accessed: May 5, 2024]
- [57] “Getting started,” *linuxcontainers.org*. [Online]. Available: <https://linuxcontainers.org/lxc/getting-started>. [Accessed: May 5, 2024]
- [58] A. Sarai and A. Suda, “The state of rootless containers,” presented at the Open Source Summit North America 2018, New York, NY, USA, Aug. 31, 2018. [Online]. Available: <https://ossna18.sched.com/event/FAPP/the-state-of-rootless-containers-aleksa-sarai-suse-llc-akihiro-suda-ntt>

- [59] “user_namespaces(7) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/user_namespaces.7.html. [Accessed: May 5, 2024]
- [60] “Run the docker daemon as a non-root user (rootless mode),” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/security/rootless>. [Accessed: May 5, 2024]
- [61] “opencontainers/runc,” *github.com*. [Online]. Available: <https://github.com/opencontainers/runc>. [Accessed: May 5, 2024]
- [62] A. Sarai, “CVE-2019-5736: runc container breakout (all versions),” *seclists.org/*, Feb. 12, 2019. [Online]. Available: <https://seclists.org/oss-sec/2019/q1/119>. [Accessed: May 5, 2024]
- [63] “Kep-127: Support user namespaces,” *github.com*, Feb. 8, 2024. [Online]. Available: <https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/127-user-namespaces/README.md>. [Accessed: May 5, 2024]
- [64] “Understanding docker container escapes,” *blog.trailofbits.com*, Jul. 19, 2019. [Online]. Available: <https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes>. [Accessed: May 5, 2024]
- [65] R. McCune, “Docker capabilities and no-new-privileges,” *raesene.github.io*, Jun. 1, 2019. [Online]. Available: <https://raesene.github.io/blog/2019/06/01/docker-capabilities-and-no-new-privs>. [Accessed: May 5, 2024]
- [66] “No new privileges flag,” *kernel.org*. [Online]. Available: https://www.kernel.org/doc/html/v4.19/userspace-api/no_new_privs.html. [Accessed: May 5, 2024]
- [67] M. Souppaya, J. Morello, and K. Scarfone, “Application Container Security Guide,” National Institute of Standards and Technology, Tech. Rep., 2017. [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-190>
- [68] “CIS Docker Benchmarks,” Center for Internet Security, Tech. Rep., 2019. [Online]. Available: <https://www.cisecurity.org/benchmark/docker>
- [69] “KernelNewbies: Linux_3.8,” *kernelnewbies.org*, Dec. 30, 2017. [Online]. Available: https://kernelnewbies.org/Linux_3.8. [Accessed: May 5, 2024]

- [70] “KernelNewbies: Linux_4.5,” *kernelnewbies.org*, Dec. 30, 2017. [Online]. Available: https://kernelnewbies.org/Linux_4.5. [Accessed: May 5, 2024]
- [71] “KernelNewbies: Linux_5.6,” *kernelnewbies.org*, Apr. 23, 2020. [Online]. Available: https://kernelnewbies.org/Linux_5.6. [Accessed: May 5, 2024]
- [72] Y. Avrahami, “New Linux Vulnerability CVE-2022-0492 Affecting Cgroups: Can Containers Escape?” *unit42.paloaltonetworks.com*, Mar. 3, 2022. [Online]. Available: <https://unit42.paloaltonetworks.com/cve-2022-0492-cgroups>. [Accessed: May 5, 2024]
- [73] R. McCune, “Dirty Pipe Linux Vulnerability: Overwriting Files in Container Images,” *aquasec.com*, Mar. 8, 2022. [Online]. Available: <https://www.aquasec.com/blog/cve-2022-0847-dirty-pipe-linux-vulnerability/>. [Accessed: May 5, 2024]
- [74] R. McCune, “Linux Kernel Vulnerability: Escaping Containers by Abusing Cgroups,” *aquasec.com*, Mar. 9, 2022. [Online]. Available: <https://www.aquasec.com/blog/new-linux-kernel-vulnerability-escaping-containers-by-abusing-cgroups>. [Accessed: May 5, 2024]
- [75] “CVE-2021-44228 Detail,” *National Vulnerability Database*, Apr. 25, 2024. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2021-44228>. [Accessed: May 5, 2024]
- [76] “seccomp(2) — linux manual page,” *man7.org*, Dec. 22, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man2/seccomp.2.html>. [Accessed: May 5, 2024]
- [77] “Seccomp BPF (SECure COMputing with filters),” *kernel.org*. [Online]. Available: https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html. [Accessed: May 5, 2024]
- [78] “Seccomp security profiles for docker,” *docs.docker.com*. [Online]. Available: <https://docs.docker.com/engine/security/seccomp>. [Accessed: May 5, 2024]
- [79] “Linux container configuration,” *github.com*. [Online]. Available: <https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md>. [Accessed: May 5, 2024]
- [80] “lxc.container.conf,” *linuxcontainers.org*, Jun. 03, 2021. [Online]. Available: <https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html>. [Accessed: May 5, 2024]

- [81] “What is SELinux?” *redhat.com*, Aug. 30, 2019. [Online]. Available: <https://www.redhat.com/en/topics/linux/what-is-selinux>. [Accessed: May 5, 2024]
- [82] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, “Speaker: Split-phase execution of application containers,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer International Publishing, 2017, pp. 230–251. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_11
- [83] N. Lopes, R. Martins, M. E. Correia, S. Serrano, and F. Nunes, “Container hardening through automated seccomp profiling,” in *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, ser. WOC’20. New York, NY, USA: Association for Computing Machinery, 2021, pp. 31–36. [Online]. Available: <https://doi.org/10.1145/3429885.3429966>
- [84] “AppArmor — Linux kernel security module.” [Online]. Available: <https://wiki.apparmor.net/>. [Accessed: May 5, 2024]
- [85] “AppArmor security profiles for Docker,” *docs.docker.com*. [Online]. Available: <https://docs.docker.com/engine/security/apparmor>. [Accessed: May 5, 2024]
- [86] H. Zhu and C. Gehrman, “Lic-Sec: An enhanced AppArmor Docker security profile generator,” *Journal of Information Security and Applications*, vol. 61, p. 102924, 2021. [Online]. Available: <https://doi.org/10.1016/j.jisa.2021.102924>
- [87] T. Cameron, “Security Enhanced Linux for mere mortals,” presented at the Red Hat Summit 2018, San Francisco, CA, USA, May 10, 2018. [Online]. Available: https://youtu.be/_WOKRaM-HI4
- [88] “opencontainers/runtime-spec — configuration,” *github.com*, Sep. 28, 2021. [Online]. Available: <https://github.com/opencontainers/runtime-spec/blob/main/config.md>. [Accessed: May 5, 2024]
- [89] J. Corbet, “The LoadPin security module,” *lwn.net*, Apr. 6, 2016. [Online]. Available: <https://lwn.net/Articles/682302>. [Accessed: May 5, 2024]
- [90] S. Ruffell, “A brief tour of Linux Security Modules,” *starlab.io*, Nov. 21, 2019. [Online]. Available: <https://www.starlab.io/blog/a-brief-tour-of-linux-security-modules>. [Accessed: May 5, 2024]

- [91] “Smack,” *kernel.org*. [Online]. Available: <https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/Smack.html>. [Accessed: May 5, 2024]
- [92] M. De Benedictis and A. Lioy, “Integrity verification of Docker containers for a lightweight cloud environment,” *Future Generation Computer Systems*, vol. 97, pp. 236–246, 2019. [Online]. Available: <https://doi.org/10.1016/j.future.2019.02.026>
- [93] S. Martínez-Magdaleno, V. Morales-Rocha, and R. Parra, “A review of security risks and countermeasures in containers,” *Int. J. Secur. Netw.*, vol. 16, no. 3, pp. 183–190, Jan. 2021. [Online]. Available: <https://doi.org/10.1504/ijsn.2021.117867>
- [94] “docker/docker-bench-security,” *github.com*. [Online]. Available: <https://github.com/docker/docker-bench-security>. [Accessed: May 5, 2024]
- [95] “aquasecurity/docker-bench,” *github.com*. [Online]. Available: <https://github.com/aquasecurity/docker-bench>. [Accessed: May 5, 2024]
- [96] R. Sengupta, R. S. Sai Prashanth, Y. Pradhan, V. Rajashekar, and P. B. Honnavalli, “Metapod: Accessible Hardening of Docker Containers for Enhanced Security,” in *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2021, pp. 01–06. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9622572>
- [97] K. Brady, S. Moon, T. Nguyen, and J. Coffman, “Docker Container Security in Cloud Computing,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0975–0980. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9031195>
- [98] O. Javed and S. Toor, “Understanding the quality of container security vulnerability detection tools,” *CoRR*, vol. abs/2101.03844, 2021. [Online]. Available: <https://arxiv.org/abs/2101.03844>
- [99] K. Brady, S. Moon, T. Nguyen, and J. Coffman, “Docker container security in cloud computing,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0975–0980.
- [100] M. K. Abhishek and D. Rajeswara Rao, “Framework to secure docker containers,” in *2021 Fifth World Conference on Smart Trends in Systems Security and Sustainability (WorldS4)*, 2021, pp. 152–156.

- [101] R. Moshkovich, “Industry’s first dynamic analysis of 4 million publicly available docker hub container images,” *prevasio.io*, Dec. 1, 2020. [Online]. Available: https://uploads-ssl.webflow.com/609ccd86b14ba6e17f9fbce4/643ba6fc599bb82bfdea2456_Red_Kangaroo.pdf. [Accessed: May 5, 2024]

A Clair docker-compose.yaml

```
1 services:
2   postgres:
3     image: docker.io/library/postgres:12
4     networks:
5       - clair-local
6     ports:
7       - 5432:5432
8     environment:
9       POSTGRES_PASSWORD: clair
10      POSTGRES_USER: clair
11      POSTGRES_DB: clair
12      restart: unless-stopped
13   clair:
14     image: clair-local:latest
15     networks:
16       - clair-local
17     ports:
18       - 6060:6060
19     restart: unless-stopped
20     volumes:
21       - ./local-dev/clair:/config
22     environment:
23       CLAIR_MODE: combo
24       CLAIR_CONF: /config/config.yaml
25   networks:
26     clair-local:
27       driver: bridge
```