

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	3
1 Introduction	4
2 Theoretical background	5
2.1 Isolation features	5
2.1.1 Chroot	5
2.1.2 Cgroups	5
2.1.3 Namespaces	6
2.1.4 Capabilities	7
2.2 Runtimes	7
2.2.1 OCI specification	8
2.2.2 Traditional runtimes	8
3 Security analysis of hardening techniques	10
4 Vulnerability detection	11
4.1 Previous studies	11
4.2 Static analysers	11
4.3 Methodology	12
4.3.1 Images selection	13
4.3.2 Tags selection	13
4.3.3 Scan images	13
4.3.4 Analysis of obtained scan reports	15
5 Conclusion	16
References	17
A Clair docker-compose.yaml	20

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-minrootfs-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

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 [18]. 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 [19]. 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 [20].

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 [21]. 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 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.

4.3.1 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.

4.3.2 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.

4.3.3 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.

```

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

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

4.3.4 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: April 25, 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: April 25, 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: April 25, 2024]
- [5] M. Kerrisk, “Namespaces in operation, part 1: namespaces overview,” *lwn.net*, Jan. 4, 2013. [Online]. Available: <https://lwn.net/Articles/531114>. [Accessed: April 25, 2024]
- [6] “Docker security,” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/security>. [Accessed: April 25, 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: April 25, 2024]
- [9] “Running containers,” *docs.docker.com*, Feb. 21, 2024. [Online]. Available: <https://docs.docker.com/engine/reference/run>. [Accessed: April 25, 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: April 25, 2024]
- [12] “Open container initiative.” [Online]. Available: <https://opencontainers.org/>. [Accessed: April 25, 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: April 25, 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: April 25, 2024]
- [16] “runc(8) system manager’s manual,” *man.archlinux.org*. [Online]. Available: <https://man.archlinux.org/man/extra/runc/runc.8.en>. [Accessed: April 25, 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: April 25, 2024]
- [18] 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>
- [19] 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.
- [20] 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.

- [21] 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: April 25, 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
```