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 Shadrunov Aleksei
student of group BIB201, 4th year of study

Approved by Supervisor
Assistant Professor, V. V. Bashun

Moscow 2024

Contents

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 occured 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 bandwith. 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 intracted 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 resouces more efficiently compared to virtual machines. For that reason namespaces are particularly useful for containerization [7].

2.1.4 Capabilities

Finally,

2.2 Runtimes

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 compregensive works was published by Omar Javed and Salman Toor in 2021 [8]. 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 [9]. 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.

4.2 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.

4.2.1 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 1. The result was then saved to the JSON file.

The next step is to combine search results from each query and sample a reasonably

```python
def get_page(page, query):
url = "https://hub.docker.com/api/content/v1/products/search"
params = {
    "page_size": 100,
    "q": query,
    "source": "community",
    "type": "image",
    "page": page
}
response = requests.get(url, params=params)
response.raise_for_status()
data = response.json()
return data["summaries"]
```

Listing 1 – Query images

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.

4.2.2 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 2. 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.

```python
pref, suf = image.split("/")
link = f"https://hub.docker.com/v2/namespaces/{pref}/repositories/{suf}/tags"
res = requests.get(link)
res = res.json()
up = res["count"]
iteration = 0
total = 0
while total < 10 and iteration < 100:
    page_num = random.randint(1, up)
    response = requests.get(link, params={"page": page_num, "page_size": 1})
    tag = response.json()["results"][0]
    if "amd64" in [i["architecture"] for i in tag["images"]]:
        tags.append(tag)
        total += 1
    iteration += 1
```

Listing 2 – Query tags

4.2.3 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 3 – 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 4 – 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 5 – Run Trivy scanner

4.2.4 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: Apr. 23, 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: Apr. 23, 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: Apr. 23, 2024]

[5] M. Kerrisk, "Namespaces in operation, part 1: namespaces overview," *lwn.net*, Jan. 4, 2013. [Online]. Available: https://lwn.net/Articles/531114. [Accessed: Apr. 24, 2024]

[6] "Docker security," *docs.docker.com*, Feb. 21, 2024. [Online]. Available: https://docs.docker.com/engine/security. [Accessed: Apr. 24, 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] 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

[9] 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.

A Clair docker-compose.yaml

```
services:
  postgres:
    image: docker.io/library/postgres:12
    networks:
      - clair-local
    ports:
      - 5432:5432
    environment:
      POSTGRES_PASSWORD: clair
      POSTGRES_USER: clair
      POSTGRES_DB: clair
    restart: unless-stopped
  clair:
    image: clair-local:latest
    networks:
      - clair-local
    ports:
      - 6060:6060
    restart: unless-stopped
    volumes:
      - ./local-dev/clair/:/config
    environment:
      CLAIR_MODE: combo
      CLAIR_CONF: /config/config.yaml
networks:
  clair-local:
    driver: bridge
```