Evaluating the security of containers through reduction of potentially vulnerable components

Michael Wager
Faculty of Computer Science
University of Applied Sciences Augsburg
Augsburg, Germany
January 2024
mail@mwager.de

Abstract

The use of software containers has become a fundamental practice in modern application development, offering benefits such as portability, scalability, and efficiency. However, containers are not immune to security vulnerabilities, which can pose significant risks to the applications they host. This work investigates the security implications of reducing potentially vulnerable components within container images. The research is guided by two primary research questions. First, a popular set of base images was compared to base images built using component reduction methods available ("distroless images"). Second, exploitability and impact of typical vulnerabilities found by image scanners were analysed. This work concludes that the usage of component reduction methods significantly reduces the amount of security vulnerabilities within container images. It also finds that the probability of exploitation of the majority of vulnerabilities found by scanners is very low, but employing them still is a strategically sound decision.

I. Introduction

Containerization technology has revolutionized the way software applications are developed, deployed, and managed. By leveraging operating system virtualization and the Linux kernel's features, containers provide several advantages compared to traditional methods, including improved scalability, faster deployment, resource efficiency, and simplified application management. By encapsulating applications and their dependencies, containers enable consistent execution across different environments, facilitating portability and flexibility. One can think of it as a simple package that encapsulates the whole application and its dependencies. It enables developers to package their applications once and deploy them anywhere, whether on-premises or in the cloud. Companies such as Amazon, Google, and Microsoft are investing heavily in containerization technology and offering container-based services to their customers. The popularity of containerization technology is expected to grow in the coming years, with many large organizations adopting it as a standard for software development and deployment, especially with regards to digitalization and the cloud movement of big enterprises in recent years. As it greatly simplifies and enhances the way modern software is developed and deployed to production, it also had great influence on the creation of a concept called DevOps, the combination of development and operations, because it facilitates continuous integration and deployment (CI/CD) workflows, leading to, when correctly applied, higher overall software quality.

Docker, introduced in 2013, also played an important role in popularizing containerization and revolutionizing software development practices. It basically provides an easy-to-use platform for building, packaging, and distributing applications as lightweight containers. The ease of use and flexibility offered by Docker quickly propelled its adoption within the software development community.

While Docker simplified the process of container creation and management, orchestrating and scaling containers across a cluster of machines remained a complex task. In response to this challenge, Google introduced Kubernetes in 2014 as an open-source container orchestration platform. Kubernetes, often referred to as "K8s," provides a framework for automating the deployment, scaling, and management of containerized applications. It offers features such as container scheduling, load balancing, and automated scaling, allowing developers to focus on application logic rather than infrastructure management. Kubernetes quickly gained traction and became the de facto standard for container orchestration.

As containers are hosted on so called container registries, all the mentioned success also leads to a giant number of public container image repositories and therefore unfortunately also to potential issues regarding security. Most of these containers contain a lot of components which are not needed by the application running in production (e.g. shells, package managers, binaries/files with special permissions etc.) and these components often have high or even critical vulnerabilities according to the Common Vulnerability Scoring System (CVSS) which could be potentially exploited by malicious actors. A report[1] from Sysdig claims „that 75% of containers have "high" or "critical" patchable vulnerabilities". So the real pity is that it should be relatively easy to patch those vulnerabilities.

A. Motivation

Teams responsible for software product security are trying to motivate teams to scan their container images for security vulnerabilities. Often these process is introduced later in the development lifecycle, leading to a large number of these findings. Knowledge of security is often rare among software engineers which is the reason for an immense effort on the side of both teams. Especially remediating these findings can be a complex task: vulnerabilities and their impact in the context of the running application need to be analysed and if necessary should be fixed. Another option would be to accept the risk of deploying applications with security vulnerabilities to production. As this process is complex and highly time consuming, one idea would be to reduce components inside the images to a minimum and just put in the things needed to run an application: basic folder structure, the correct runtime environment and the source code and its dependencies. In fact, there are projects making exactly this possible. One example is the project called "distroless" from Google [1], promising a smaller attack surface and less noise of container security scanners. It is very common to say the less components a container has, the more secure it is because this implies a smaller attack surface [2]. According to Pratyusa K. Manadhata [3] a "larger attack surface leads to a larger number of potential attacks on a system". However, there are doubts [2] that these concepts called "distroless" really lead to more security which is one reason for the idea of this work.

Apart from simply reducing components, it would be interesting to know if certain vulnerability findings are actually exploitable at all. Compliance policies dictate that these issues need to be remediated, else an application is not allowed to be deployed to production. What if these vulnerabilities are false positives and there is just no possibility to exploit them?

[1]https://sysdig.com/content/c/pf-2022-cloud-native-security-and-usage-report

[2]https://www.redhat.com/en/blog/why-distroless-containers-arent-security-solution-you-think-they-are

B. Research objective

This paper will analyse approaches for reducing the attack surface of container images through reducing vulnerable components and aims to evaluate if certain component reduction strategies decrease the findings of security scanners in a significant way. Also this work will analyse actual exploitability and impact of certain scanner findings with the goal of creating better statements and recommendations for development teams. Therefore the following research questions (RQs) shall be answered:

1) **RQ1**: Does the reduction of components significantly reduce the amount of vulnerabilities within the container image?
2) **RQ2**: Are typical vulnerabilities found through container security scanners actually exploitable and therefore a risk to the application?

II. Background

This chapter gives an overview of the concepts necessary to understand in order to grasp the following research and evaluation. Containers will be explained, followed by a presentation of the basics of vulnerability assessment regarding containers and container image scanners.

A. Containers

Before we can apply security mitigations against container based threats, we need to understand how containers work. This chapter describes the basic Linux mechanisms like Control Groups and Capabilities, which enable the foundation of containers.

When asking the question "What exactly is a container?", a lot of developers do not really know the history and concepts behind it. Nevertheless this is important to understand before we can dive deeper into security topics. In Linux, a process is simply a running instance of a program. A program is a set of instructions that can be executed by the Central Processing Unit (CPU) to perform a specific task. When a program is executed, it is loaded into memory and becomes a process. Each process in Linux is assigned a unique process ID (PID), which is used to identify it. The PID is a numeric value that is used by the operating system to manage and control the process. Linux provides a variety of commands and tools that can be used to view and manage processes, such as the "ps" command, which can be used to display information about running processes, and the "kill" command, which can be used to terminate a process. Linux processes are designed to run independently of one another, which means that they have their own memory space and resources. This allows multiple programs to be executed simultaneously on a single system, which is a key feature of modern operating systems like Linux. The ability to run multiple processes concurrently is essential for modern computing, as it allows users to perform multiple tasks at the same time without experiencing performance issues.

Basically a "container" is just a Linux process ("containerized process") isolated through certain kernel features like namespaces, capabilities, Control Groups and chroot ([4]). For example, chroot enables the process child to get a new filesystem root preventing the process to read the filesystem of the parent environment. This way it is possible to make the environment, when viewed from inside a container, to look like it is running in a complete operating system with a filesystem structure like a full Linux system. The Linux feature capabilities divides privileges traditionally associated with the superuser into distinct units. For example, giving a process the capability "CAP_NET_RAW" allows the process to bind to any address for networking. Control Groups (cgroups) limit the resources that a process can use, such as memory, CPU and network input/output, while namespaces control what it can see. By putting a process in a namespace, you can restrict the resources that are visible to that process.

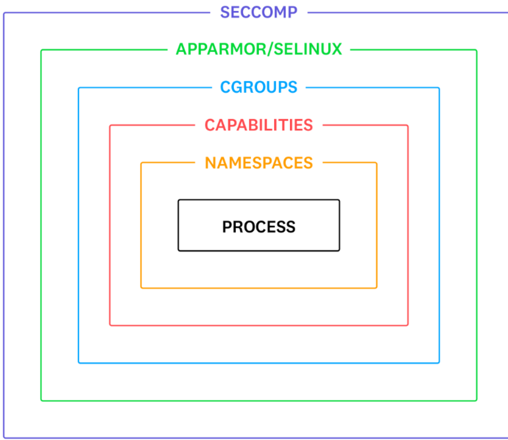

Figure 1. Container Isolation [5]

Figure 1 shows a simple illustration of the mentioned isolation. Apart from the mentioned Linux kernel features, App Armor and Seccomp are further Linux security mechanisms used to enhance the isolation of processes. These are not directly associated with containers but can be used additionally to reach even more isolation and security.

B. Container Vulnerability Assessment

Vulnerability Assessment is the process of analysing vulnerabilities with the goal of remediation. Remediation is typically a 4-step process: Finding, prioritizing, fixing and monitoring vulnerabilities. A Software Vulnerability is a flaw or misconfiguration in an application, library, function that can be exploited. If vulnerabilities are found, e.g. by vendors or security researchers, they get published and assigned an ID, the so called CVE, short for Common Vulnerabilities and Exposures. CVE is a list of publicly disclosed computer security flaws. Each vulnerability gets assigned a CVE ID number to refer to. Also, these CVEs get a score to express their severity, or criticality, the so called CVSS, short for Common Vulnerability Scoring System, "an open framework for communicating the characteristics and severity of software vulnerabilities" ([8]). It calculates a numerical score from 0-10 indicating the severity of a vulnerability. Severities can be low, medium, high or critical and may get calculated based on three metric groups: Base, Temporal, and Environmental:

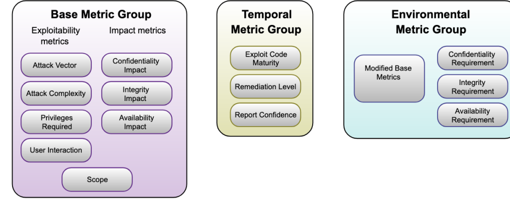

Figure 2. CVSS Metric groups [8]

The Base group is focussed on characteristics of a vulnerability that are constant over time and consists of Exploitability metrics and Impact metrics. The Temporal group is focussed on characteristics of a vulnerability that change over time, like the maturity of available exploit code, and the environmental group is focussed on characteristics of a vulnerability that are unique to a user's environment. Users of CVSS should use the Base Group Score with Temporal and Environmental Scores specific to their use case to produce a severity more accurate for their organizational environment. All these groups will later be especially important in order to answer RQ2. This work deals with the current version CVSS3.1, the new version 4.0 with a target official publication date of October 31 2023 will not be used ([3]).

The Impact metrics are based on the three protection goals of information security: Confidentiality, Integrity and Availability. This work is inspired by the CID Triad definition as defined in [9], because it is already focused on containers:

- Confidentiality (C) describes the attackers' ability to access information in the exploited container.
- Integrity (I) refers to the attackers' ability to modify information in the exploited container.
- Availability (A) indicates the impact on the accessibility of the exploited container.

All of CIA metrics have three possible values: None, Low and High and vulnerabilities consisting of "High" in each Impact metric are defined as High Impact (HI) vulnerabilities, as defined in [9]. Also, the definition for High likely Exploitation (HE), composed of "AC: Low, UI: None, PR: None", shall be used in this work. It defines a vulnerability as highly likely exploitable if the attack vector is low and not user interaction or special privileges are required ([9]).

Important to note is that also the availability of pubic exploit code is important: the presence of a simple-to-use

[3]https://www.first.org/cvss/v4-0/

exploit would increase the CVSS score, while the creation of an official patch would decrease it. Unfortunately often there are patches available but still the issues don't get fixed which could be that CVSS sets a smaller score if a patch is available.

Vulnerability Assessment for containers might be slightly different compared to traditional operation systems ([4]). When looking at containerized environments, many of the expectations for regular OS environments simply do not hold true, and these differing factors come into play when assessing any given vulnerability. For example, in a traditional OS a mentioned privillege escalation could make an attacker to compromise the host, while in a container environment, it permits the attacker to compromise the container. So for example confidentiality metric of CVSS could be moved from HIGH to LOW for containers. Therefore it is possible that the severity of a vulnerability is different in the context of a certain container distribution.

Also, when analysing vulnerabilities found in container images, often times of course generic assumptions need to be made. Normally it is very important to assess vulnerabilities also in the context of an application, i.e. looking at their specific use case, source code, architecture etc. For example, a use-after-free vulnerability related to the bluetooth stack of the kernel ([5]) with Attack vector: Adjacent, may not be exploited in the context of a web application, because an attacker would have to be physically within proximity of the container, which is most likely running in a large cloud data center like Amazon Web Services.

C. Container Image Vulnerability Scanners

As mentioned in the introduction, the adoption of containers also brings new challenges in terms of security. Containerized environments require robust security measures to protect against potential vulnerabilities and threats. Container security scanners play a crucial role in identifying and mitigating security risks by analysing container images and runtime environments. This section provides an overview of the functionality of container security scanners and presents the scanner used for evaluation later in this paper: twistcli from PrismaCloud.

Container security scanners are specialized tools designed to assess the security posture of containers, specifically focusing on vulnerabilities and misconfigurations. These scanners automate the process of analysing container images and runtime environments, enabling developers and security teams to proactively identify and remediate security issues. The scanning process typically involves the following steps:

1) **Image Analysis**: Container images are analysed to identify potential vulnerabilities, or to find secrets or misconfigurations. This analysis includes examining the software packages and dependencies within the image, checking for known vulnerabilities, and comparing against vulnerability databases and security advisories.
2) **Configuration Audit**: Scanners assess the container's configuration to ensure best practices are followed and potential misconfigurations are flagged. This includes verifying access control settings, network configurations, and security-related parameters.

The following scanner will be used as part of the evaluation in this paper:

PrismaCloud Twistcli [10]: Twistcli is a container security scanner developed by Twistlock (now part of Palo Alto Networks). It offers comprehensive vulnerability management and compliance features. Twistcli provides vulnerability scanning for both container images and running containers, offering a holistic view of the containerized environment's security. For comparing against vulnerability databases and security advisories, twistcli is using the so called Intelligence Stream of PrismaCloud ([6]), a "real-time feed that contains vulnerability data and threat intelligence from a variety of certified upstream sources". Additionally, PrismaCloud has a dedicated research team which will analyse vulnerabilities and eventually flag them with a custom ID even before a CVE is published. Per default, PrismaCloud is using CVSS version 3 and the National Vulnerability Database (NVD), but in some cases the returned severity of a vulnerability might differ from the CVSS base score. This is because PrismaCloud is using the vendor's calculation of the impact if available. It shows the vendor's calculation when reporting issues from an image running the vendor's OS, and falls back to default CVSS calculation ([11], Example: [7]). This is a good example of utilizing the environmental feature of CVSS. Also when analysing images from RedHat, twistcli automatically uses their ratings [8].

III. Related work

This chapter presents the review of existing literature and research relevant to the topic of this paper. Multiple papers and resources are recommending "distroless" approaches or at least using "minimalistic technologies like Alpine Linux" or "minimal base images" ([2], [4], [12], [13], [9], [10], [11]). It is very common to say the less

[4]https://www.redhat.com/en/blog/containers-vulnerability-risk-assessment

[5]CVE-2022-42896

[6]Docs of Prisma Cloud Intelligence Stream

[7]CVE-2022-48522 in Ubuntu security tracker

[8]https://access.redhat.com/security/updates/classification/

[9]https://canonical.com/blog/combining-distroless-and-ubuntu-chiselled-containers

[10]https://owasp.org/www-project-kubernetes-top-ten/2022/en/src/K02-supply-chain-vulnerabilities

[11]https://developer.ibm.com/learningpaths/scan-container-images-for-vulnerabilities/best-practices-for-container-image-security/

components a container has, the more secure it is because this implies a smaller attack surface. Nevertheless the goal is to find scientific prove for all of these statements. A few interesting papers have been found and will be presented here.

Minification to reduce the number of vulnerabilities: According to [14], "a strong linear relationship is found between the number of detected vulnerabilities and the number of packages present in the image". Bhupinder Kaur et al. clearly state that "minification reduces the number of vulnerabilities". Software components removed during their minification process were for example compilers, unused Python packages and unused file compression utilities. As shown in their results, it is common for container images to hold hundreds of vulnerabilities, including several of critical severity. In the discussion section it is stated that it would be more practical for software developers to identify and remove unnecessary dependencies when they build containers, based on their knowledge of the application. They argue that this is often a complex, error prone and highly manual task. Therefore it should make more sense for developers to already have minimal secured base images available. All these findings clearly add to the motivation of this research.

Base Systems Security Analysis: Arkadiusz Maruszczak et.al. [15] are comparing common base images with google distroless. Their results already answer a part of RQ1. Because google distroless is based on debian, also Debian Bullseye was selected for comparison. Additionally, they added an Alpine-based image.

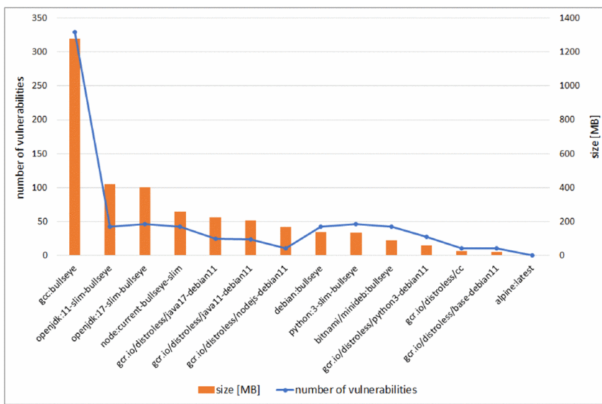

Figure 3. Correlation between size of the image and number of vulnerabilities detected in it [15]

Figure 3 shows the correlation between size of images and the number of vulnerabilities detected in it. Positive correlation can be seen for some images (Python, Node.js and GCC images), also for google distroless, but not for the java images. Even the distroless java images still contain a high amount of findings. This should later be evaluated with the goal of reproduction and further analysis. Also they conclude that alpine images can be seen as direct competitor to google distroless. Although it focusses only on the comparison of base images with images from the google distroless project, it is now possible to compare the results of this work with their results in order to try to reproduce and verify their findings.

Comparing Severity and Exploits: Luca Allodi et. al. [16] are providing good reasoning regarding RQ2, as most organisations are using CVSS alone in order to prioritize vulnerabilities to remediate. They confirm that security configuration manager software usually rely on vulnerability data from the National (U.S.) Vulnerability Database [12] (NVD) for their assessments. The product security team the author of this paper works in also just blindly dictates that "all findings of container scanners with high or critical severity according to the CVSS need to be fixed". However, the paper concludes that "fixing a vulnerability just because it was assigned a high CVSS score is equivalent to randomly picking vulnerabilities to fix" and also that "The CVSS base score alone is a poor risk factor from a statistical perspective". They go further and came to the conclusion that "the existence of proof-of-concept exploits is a significantly better risk factor" and that "fixing in response to exploit presence in black markets yields the largest risk reduction". Figure 4 shows the data sources used: besides ExploitDB, they are using "EKITS" (Black-marketed exploits) and "SYM" (Vulnerabilities exploited in the wild) as additional source for calculating risk.

DB	Content	Collection method	#Entries
NVD	CVEs	XML parsing	49,599
EDB	Publicly exploited CVEs	Download and Web parsing to correlate with CVEs	8,122
SYM	CVEs exploited in the wild	Web parsing to correlate with CVEs	1,277
EKITS	CVEs in the black market	Ad-hoc analysis + Contagio's Exploit table	103

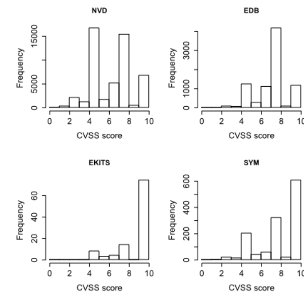

Figure 4. Summary of Datasets and Distribution of CVSS scores per dataset [16]

Worth noting is that public exploitation data is often hard to find. A common assumption made in academy

[12]http://nvd.nist.gov

and industry alike is that proof-of-concept exploit data can be used to measure the state of security of a software component, but even if public exploits exist, this says nothing about exploitation "in the wild".

More interesting regarding RQ2, according to [17] only "2%-5% of published vulnerabilities have observed exploits in the wild".

Data-Driven Exploit Predictions: Jay Jacobs. et. al. [18] are presenting a community driven, automated model to calculate the probability of observing exploitation activity in the next 30 days. They also claim that *numerous prior studies have shown that CVSS does not perform well when used to prioritize* remediation. The result of the paper is the Exploit Prediction Scoring System (EPSS): a scoring system that can accurately calculate the likelihood of exploits in the wild. It estimates the probability that a vulnerability will experience exploitation activity in the wild. It accomplishes this entirely by data-driven, empirical analysis and it is fully automated. Among others, they are using the following data sources for the calculation: ExploitDB, GitHub, Metasploit, KEV [19], twitter feeds mentioning CVEs. Each CVE gets an estimate of 0-1 (0% - 100%).

Empirical Study of Exploitability & Impact: Mubin Ul Haque et. al. conducted an empirical study to investigate the exploitability and impact of security vulnerabilities in base-images [9]. A collection of 1,983 distinct security vulnerabilities within base images has uncovered 13 relevant findings. They aim to provide developers with insights into potential security risks associated with base images, encouraging them to thoroughly assess the security of base images before building their applications. For researchers, the paper emphasizes the necessity of creating tools to reduce the vulnerability of these base images to exploitation. They used the NVD to study the exploitability and impact of all found vulnerabilities, and also ExploitDB, metasploit and VulHub. They mention, that usage of well analysed and verified base images, reduces number of vulnerabilities in the final image. Additionally it was declared, that size of the image may not always correlate with the number of vulnerabilities, but a trend can be noted.

IV. Research process

This chapter will describe the research done in order to answer the research questions. First, existing and popular component reduction tools available and their functionality will be presented. Also, the automated process of building, scanning and analysing the relevant base images will be explained. To answer RQ2, comprehensive analysis of found vulnerabilities was done with focus on exploitability and impact.

A. Existing component reduction methods

This section will present existing component reduction tools and document their functionality.

1) Alpine: Alpine is a very small and general purpose linux distribution with focus on resource efficiency and security. It is using busybox so containers built off it do have a shell by default. As Apline Linux is focussing on resource efficiency, it is very small compared to traditional Linux distributions ([20]). To make this possible, Alpine uses the standard C library *musl* instead of *glibc* or the Init-System OpenRC instead of *systemd*. For example, *musl* is a way more simple implementation which prioitizes memory safety to avoid side effects and to prevent unexpected behaviour. This is very important to know, as certain applications need *glibc* and therefore cannot run using the *musl* library. Alpine is hosted on DockerHub and also provides ready to use runtime images like *node:20-alpine*.

As it is so small and focusses on security it is an often recommended component reduction method to build minimal containers and therefore will be included in the evaluation of this paper.

2) Google Distroless: Google Distroless [1] is an open source project by Google, published in 2007. It promises a reduced attack surface of docker containers and therefore reduce the noise of container scanners. They are providing production ready runtime images for popular runtimes like Java and Node.js. Distroless images can be built 2 ways: using inside your Dockerfile (pretty easy) or build images yourself using a build tool called *bazel*. Building an own Distroless image using a Dockerfile is pretty simple. It is utilizing the docker concept of so called multi stage builds:

```
FROM node:20 AS build-env
COPY . /app
WORKDIR /app

RUN npm ci --omit=dev

FROM gcr.io/distroless/nodejs20-debian11
COPY --from=build-env /app /app
WORKDIR /app
CMD ["hello.js"]
```

The example *Dockerfile* shows the instructions to build a simple node.js based image. In the first stage, *node:20* is used as base image, followed by copying the source code in and installing the applications dependencies. Then, in the second stage, *gcr.io/distroless/nodejs20-debian11* is used as base image, built artefacts are copied over and the application will get executed. Google distroless does not officially support the PHP runtime and research showed a fork of the project [21]. As PHP is a relevant runtime, this fork will be added to the comparison later.

3) RedHat UBI micro: RedHat UBI (Universal base images) "micro" [22] is a project by RedHat which enables trimming the size of images. The benefit here is having the

same Security response team and hardening as with any image based on RedHat enterprise linux. Unfortunately they are not providing production ready images so the build process is a little bit more complicated. Reading a blogpost from RedHat, criticizing google distroless [13] until the end, it is just an advertisement for their UBI micro project. As they are excluding a package manager and all of its dependencies which are normally pulled into a container image, it makes sense including them into the evaluation of this paper.

4) Ubuntu Chisel: Ubuntu "chisel" [23] is an open source project by Canonical, highly inspired by google distroless. They are providing some production ready images, but mostly also building it yourself is needed for certain runtimes like Node.js. They advocate the creation of images from the ground up ("FROM SCRATCH"). Starting from a blank slate is ideal for hosting statically compiled programs, particularly when image size and build times are crucial factors. However, using a completely empty starting point like 'scratch' might not be the best choice when you're working with interpreted languages that rely on complex environmental requirements. In such cases, you'll have to regularly update your base image to include the latest versions of those necessary components. Being based on Ubuntu, they promise having critical and high severity vulnerabilities fixed within 24h which is a good reason to include them into the evaluation.

5) Chainguard images: Chainguard, a security vendor founded in 2021, offers minimal, ready to use images for a wide range of runtimes, including Node.js, PHP and Ruby [24], calling it the *next generation of distroless images*. They are promising "hardened images with 0-known vulnerabilities, a minimal footprint, and SBOMs.". Especially the promise of having "0 vulnerabilities" seems to be very interesting. All these runtime images are based on "Wolfi", an open source minimal base image, supporting both relevant c libraries glibc and musl. Also interesting, these images contain a shell which could ease development, but also opening the door for a class of attacks where an attacker can get a shell inside the container. It is definitely worth including them into the evaluation.

B. Data generation

This section will describe the process of data and statistic generation.

The process of building the base images, scanning them and analysing the findings was automated using the Python programming language. The related source code can be found here: github.com/mwager/distroless_evaluation. Also this source code was used to generate all figures displayed in this paper.

[13]https://www.redhat.com/en/blog/why-distroless-containers-arent-security-solution-you-think-they-are

All builds and scans were executed on a MacBook using Vagrant [25] for virtualization of Ubuntu 20.04.6 LTS. The Docker version is 24.0.6, build ed223bc, version of twistcli is 30.03.122. The codebase basically contains three important python files:

- **scan.py** - builds and scans all relevant base images using docker and twistcli, which creates the JSON files containing the scanner results for later analysis
- **analysis.py** - analyses the JSON files generated by scan.py and adds additional information to each vulnerability (e.g. EPSS exploit probability). Writes *results/FINAL.json*
- **results.py** - reads *results/FINAL.json* generated before and helps generating statistics

The file **scan.py** contains all relevant base images for further analysis.

Also, a report from Sysdig [26] on Container Security shows that Node Package Manager (NPM) and Java are the most popular open-source non-OS packages (Figure 5) used in containers, which is also why this work will put focus on these languages regarding the selection of vulnerabilities to analyse for RQ2.

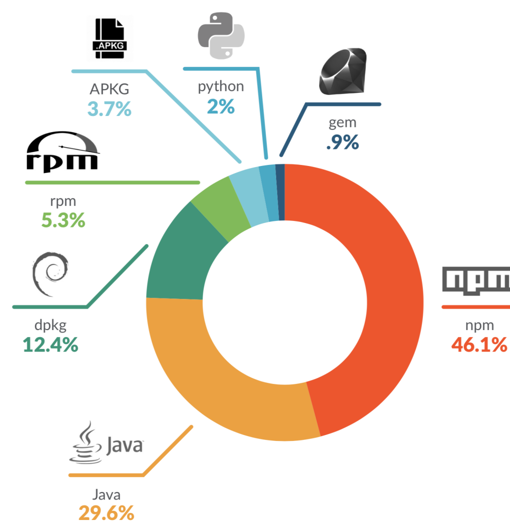

Figure 5. Top 3rd-party libraries used in containers [26]

To get more CVE information for a given scanner finding, first idea was to use the CVE Vulnerability API ([27]) and searchsploit to check for public exploits [28]. Also, The CISA Known Exploited Vulnerabilities Catalog ([19]), a database of security issues in applications that have been published and leveraged by malicious actors, could be used to match with the CVEs of the scanner results (currently containing 1014 items). However, as the work in [18] already doing this inclusion of the mentioned additional data sources, only the existence of a public exploit from *twistcli* and the EPSS-score will be added to each vulnerability using their API [29].

twistcli categorizes vulnerabilities into two categories: Operating System (OS)-based and Runtime-based. An OS-based vulnerability could be for example related the C library used by the operating system, or binaries available within the distribution (e.g. curl, openssl, gcc), while Runtime-based vulnerabilities are more related to the programming environment (e.g. nodejs NPM packages).

C. Exploitability Analysis

To answer RQ2 (2), the exploitability and impact of vulnerabilities should be analysed. This section will take a closer look at a selection of vulnerabilities reported by *twistcli* in the data generation phase. Typical exploits happening inside a container are related to privilege escalation, access control or installing malware. For each vulnerability found, the Common Vulnerabilities and Exposures (CVE) and its CVSS scoring will be checked, as well as the existence of public proof-of-concept exploits or if there is data of exploitation in the wild available.

Relevant criteria for the selection of vulnerabilities to analyse are:

1) **Severity**: To align with management policies of remediating only vulnerabilities of high and critical severity according to the CVSS we only select high and critical findings
2) **Category**: Findings are selected for OS-related and Runtime-based

Additionally, manual intervention was taken place: when filtering for high and critical findings, 15 critical and 80 high were left. Also, a lot of these remaining findings were related to the same packages like *curl* or *openssl*, so per package, only one was picked. Regarding Runtime-based, focus was put on at least one per relevant runtime. As these criteria are aligned with real-world practise it was decided against randomly picking them.

1) OS related vulnerabilities: Certain OS-related vulnerabilities have been manually picked from the scan results and will be analysed in this section.

CVE-2020-1751

An out-of-bounds write vulnerability was found in glibc before 2.31 when handling signal trampolines on PowerPC. Specifically, the backtrace function did not properly check the array bounds when storing the frame address, resulting in a denial of service or potential code execution. The highest threat from this vulnerability is to system availability

CWE-787: Out-of-bounds Write ("The product writes data past the end, or before the beginning, of the intended buffer.")

Found in image: node_16-slim:latest on a Debian GNU/Linux 10

CVSS v3.1 Base score: 7.0 (HIGH)

CVSS v3.1 Vector: Vector: CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:H/A:H

Analysis:

As the Attack vector is "local", this vulnerability is not possible to exploit remotely. An attacker would need direct shell access inside the running container. Also, Attack complexity is high, meaning it is not trivial to exploit. Regarding the impact, all three protection goals are HIGH (CIA).

When accessing the image via a shell, we want to check if the vulnerable version of glibc is indeed installed:

```
$ ldd --version
ldd (Debian GLIBC 2.28-10+deb10u2) 2.28
```

So it matches the vulnerable version of glibc specified in the CVE report. No form of authentication is required for exploitation. Technical details are known, but there is no publicly available exploit. Upgrading to version 2.31 of glibc would fix the vulnerability. Nevertheless upgrading this manually during the image build process is complex and therefore it is to be assumed that this vulnerability is available in a lot of production applications running on the mentioned base image. However, without shell access an attacker cannot exploit this vulnerability.

CVE-2023-4911

A buffer overflow was discovered in the GNU C Library's dynamic loader ld.so while processing the GLIBC_TUNABLES environment variable. This issue could allow a local attacker to use maliciously crafted GLIBC_TUNABLES environment variables when launching binaries with SUID permission to execute code with elevated privileges.

CWE-787: Out-of-bounds Write ("The product writes data past the end, or before the beginning, of the intended buffer.")

Found in image: ibmjava_jre:latest / Ubuntu 22.04.3 LTS

CVSS v3.1 Base score: 7.0 (HIGH)

CVSS v3.1 Vector: Vector: CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

Analysis:

As the Attack vector is "local", this vulnerability is also not possible to exploit remotely. An attacker would need direct shell access inside the running container. The difference to the glibc issue before is that now, Attack complexity is low, meaning it is probably easy to exploit. Also, a public exploit is available [14]. Regarding the impact, all three protection goals are HIGH (CIA). This vulnerability is very current (October 2023) and no fix is available yet.

```
# get a shell inside the container:
$ docker run -it --entrypoint /bin/bash ibmjava_jr

# check glibc version
root@f819297027ed:/# ldd --version
```

[14] https://www.openwall.com/lists/oss-security/2023/10/03/2

```
ldd (Ubuntu GLIBC 2.35-0ubuntu3.3) 2.35

# Try the POC
$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=A" "Z=`printf '%08192x' 1`" /

Usage:
su [options] [-] [<user> [<argument>...]]
...
```

The listing tries to execute the POC against version 2.35 like described in the public exploit report does not lead to a segmentation fault, although there is no fix available yet. Also reading the explanation of the exploit report it does not look easy to exploit it at all, so it seems strange that Attack complexity is low according to the NVD. This is probably just because there is a public exploit available. It is very doubtful that an attacker without comprehensive knowledge can exploit this easily because it would require a lot of knowledge in exploiting buffer overflows. Also, as this vulnerability again needs shell access, an attacker cannot exploit this vulnerability without first making its way into the running container (e.g. through a vulnerability in the custom application which uses the mentioned base image).

CVE-2022-32221
When doing HTTP(S) transfers, libcurl might erroneously use the read callback (‘CURLOPT_READFUNCTION‘) to ask for data to send, even when the ‘CURLOPT_POSTFIELDS‘ option has been set, if the same handle previously was used to issue a ‘PUT‘ request which used that callback. This flaw may surprise the application and cause it to misbehave and either send off the wrong data or use memory after free or similar in the subsequent ‘POST‘ request. The problem exists in the logic for a reused handle when it is changed from a PUT to a POST.
CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
Found in image: piotrkardasz-php-distroless_8.1 / Distroless (based on Debian GNU/Linux 11)
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
Attack vector is network so in theory, this vulnerability could be exploited remotely. This vulnerability was found in the image *piotrkardasz-php-distroless*, which is an open source fork of google distroless which aims to support PHP. As seen in the analysis below, this image has a significant amount of vulnerabilities compared to the official google distroless images, therefore it should be assumed that the project has not successfully reached its goal. Nevertheless exploitability should be checked. Libcurl is a widely used open-source library for making network connections written in C and there are bindings for PHP [15], allowing PHP developers to make HTTP requests and interact with web services using libcurl functionality. Therefore it should be assumed that this is the reason this vulnerability was only found in the mentioned image and not in one of the official google distroless images. As PHP could be using the functionality, exploitation could be possible. However, even as there is a public exploit available, the example code is based on the C programming language and exploitation of this through special crafted HTTP requests against a running PHP application seems very unlikely. An attacker would also need to inject PHP code into an application in order to exploit and if code injection is possible there would be definitely better exploits.

CVE-2023-0464
A security vulnerability has been identified in all supported versions of OpenSSL related to the verification of X.509 certificate chains that include policy constraints. Attackers may be able to exploit this vulnerability by creating a malicious certificate chain that triggers exponential use of computational resources, leading to a denial-of-service (DoS) attack on affected systems. Policy processing is disabled by default but can be enabled by passing the ‘-policy’ argument to the command line utilities or by calling the ‘X509_VERIFY_PARAM_set1_policies()’ function.
CWE-295: Improper Certificate Validation
Found in image: node_18.14.1-alpine:latest / Alpine Linux v3.17
CVSS v3.1 Base score: 7.5 (HIGH)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
Analysis:
Attack vector is network and this attack seems not to be very complex. Regarding the impact, only availability is high meaning only denial of service can be achieved when exploiting. There is no public exploit available and the official severity reported by openssl itself is "low" [16]. An attacker could craft a malicious certificate chain to trigger denial of service of the system. They could also use a botnet to overload the app.

```
$ sudo docker run -it --entrypoint\\
/bin/sh node_18.14.1-alpine
/#
/# openssl
/bin/sh: openssl: not found
/# find / -name openssl
/usr/local/include/node/openssl
```

The listing shows a look into the image. The *openssl* binary is not available, but we can see that the nodejs runtime is using a custom (and vulnerable) installation of

[15] https://www.php.net/manual/en/intro.curl.php
[16] https://www.openssl.org/news/secadv/20230322.txt

openssl. So exploitability could be possible but it would only have impact on availability.

CVE-2018-12886
stack_protect_prologue in cfgexpand.c and stack_protect_epilogue in function.c in GNU Compiler Collection (GCC) 4.1 through 8 (under certain circumstances) generate instruction sequences when targeting ARM targets that spill the address of the stack protector guard, which allows an attacker to bypass the protection of -fstack-protector, -fstack-protector-all, -fstack-protector-strong, and -fstack-protector-explicit against stack overflow by controlling what the stack canary is compared against.
CWE-209: Generation of Error Message Containing Sensitive Information
Found in image: node_16-slim:latest / Debian GNU/Linux 10 (buster)
CVSS v3.1 Base score: 8.1 (HIGH)
CVSS v3.1 Vector: Vector: CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
Attack vector is network, but looking at the public explot it seems very complex to trigger it. Also there is no relation to how this should be possible over network. Regarding impact, all three protection goals would be violated. An attacker would need to first get shell access in order to stack-trace-exploit a certain binary which was compiled with a vulnerable gcc version. Highly unlikely and also not interesting from an attacker point of view. Also the exploit works only against ARM architectures. So it seems like gcc is lying around in the image and the scanner just detects and reports it but this would very unlikey to be a real risk.

CVE-2019-8457
SQLite3 from 3.6.0 to and including 3.27.2 is vulnerable to heap out-of-bound read in the rtreenode() function when handling invalid rtree tables.
CWE-125: Out-of-bounds Read
Found in image: php_fpm-buster:latest / Debian GNU/Linux 10 (buster) and node:14-slim and node:16-slim
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
The relevant question here would first be: is the application using SQLite at all. If not, not exploitable. And even then it seems pretty convoluted - one would need to create an invalid database file and convince SQLite to load it and then get to leak some heap data. Extremely unlikely.

CVE-2019-17498
In libssh2 v1.9.0 and earlier versions, the SSH_MSG_DISCONNECT logic in packet.c has an integer overflow in a bounds check, enabling an attacker to specify an arbitrary (out-of-bounds) offset for a subsequent memory read. A crafted SSH server may be able to disclose sensitive information or cause a denial of service condition on the client system when a user connects to the server.
CWE-190: Integer Overflow or Wraparound
Found in image: php_fpm-buster:latest / Debian GNU/Linux 10 (buster)
CVSS v3.1 Base score: 8.1 (HIGH)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:N/A:H
Analysis:
Question would be: is the image - next to running an application - also running an SSH server accepting incoming connections. This is extremely unlikely. And even in this extremely implausibly case, this vulnerability could only get exploited if the SSH server was configured in a certain case. The public exploit referenced from the NVD page unfortunately leads to an HTTP 404 status.

2) Runtime related vulnerabilities: Also, certain Runtime-related vulnerabilities have been manually picked from the scan results and will be analysed here.

CVE-2023-3824
In PHP version 8.0. before 8.0.30, 8.1.* before 8.1.22, and 8.2.* before 8.2.8, when loading phar file, while reading PHAR directory entries, insufficient length checking may lead to a stack buffer overflow, leading potentially to memory corruption or RCE.*
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
Found in image: php_fpm-buster:latest / Debian GNU/Linux 10 (buster)
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
This vulnerability affects the mentioned PHP versions. According to the NVD information, Attack vector is Network and complexity is low. All three protection goals may be violated. This vulnerability relates to the handling of *phar* (PHP Archive) files. Phar files are a way to package PHP applications into a single archive for distribution and execution. It looks like when these PHP versions read the directory entries within a phar file, it does not perform sufficient length checking on the data it reads. This means an attacker would need to deploy a malicious phar file containing directory entries with excessive or incorrect data to overflow the buffer used by PHP to store this data.

```
$ sudo docker run -it \\
--entrypoint /bin/sh php_fpm-buster
#
# php --version
```

```
PHP 8.2.7 (cli) (built: Jun 13 \\
2023 11:38:13) (NTS)
```

Checking inside the container we can confirm the vulnerable PHP version. So an application inside this container could be exploitable. However, an attacker would need to be able to inject a malicious phar and somehow be able to upload it to the server (e.g. file uploads) and finally trigger processing it via the server's PHP interpreter. If successful, the attacker might gain control over the PHP process and potentially execute arbitrary code on the target server, leading to remote code execution. In summary, this vulnerability is not exploitable without taking into account the context of an application.

CVE-2023-32002
The use of ‘Module._load()‘ can bypass the policy mechanism and require modules outside of the policy.json definition for a given module. This vulnerability affects all users using the experimental policy mechanism in all active release lines: 16.x, 18.x and, 20.x. Please note that at the time this CVE was issued, the policy is an experimental feature of Node.js.
Found in image: node_18.14.1-alpine:latest / Alpine Linux v3.17
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
This vulnerability affects the mentioned Node.js versions. According to the NVD information, Attack vector is Network and complexity is low. All three protection goals may be violated. Node.js has a security mechanism which is designed to control and restrict which modules can be loaded. In could be possible that a module gets loaded even if it doesn't meet the criteria specified in the policy. An attacker would somehow need to be able to inject a payload which directly goes into the module load call, which could lead to unauthorized code get loaded. It is an experimental feature and no public exploit is available. In summary it is highly unlikely to be exploited, but it would need to be considered in the context of a concrete application.

CVE-2022-25883
Versions of the package semver before 7.5.2 are vulnerable to Regular Expression Denial of Service (ReDoS) via the function new Range, when untrusted user data is provided as a range.
CWE-1333: Inefficient Regular Expression Complexity
Found in image: node_18.14.1-alpine:latest / Alpine Linux v3.17
CVSS v3.1 Base score: 7.5 (HIGH)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
Analysis:
This vulnerability affects a Node.js package called "semver", the semantic version parser for NPM. If an application is not using this package, it is not exploitable. If an app would use it, it could be vulnerable to Regular Expression Denial of Service (ReDoS) via the function new Range, when untrusted user data is provided as a range. It would result in a long running calculation of the regular expression engine, resulting in denial of service. There is a public exploit and a fix available. Probability of exploitation within the next 30 days (EPSS) equals 0.0009%.

CVE-2023-26136
Versions of the package tough-cookie before 4.1.3 are vulnerable to Prototype Pollution due to improper handling of Cookies when using CookieJar in rejectPublicSuffixes=false mode. This issue arises from the manner in which the objects are initialized.
CWE-1321: Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution')
Found in image: node_14-slim:latest / Debian GNU/Linux 10 (buster)
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
This vulnerability affects a Node.js package called "tough-cookie", an open source library supporting cookie handling. If an application is not using this package, it is not exploitable. If an app would use it, an attacker can expose or modify a limited amount of property information on certain objects. There is a public exploit and a fix available. Probability of exploitation within the next 30 days (EPSS) equals 0.00173%.

CVE-2022-48565
An XML External Entity (XXE) issue was discovered in Python through 3.9.1. The plistlib module no longer accepts entity declarations in XML plist files to avoid XML vulnerabilities.
CWE-611: Improper Restriction of XML External Entity Reference
Found in image: amazonlinux_2:latest
CVSS v3.1 Base score: 9.8 (CRITICAL)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Analysis:
This vulnerability affects a certain Python library inside the amazon RHEL image. If an application is not using Python, it is not exploitable. If an app would use it, a potential XML exploit would be possible. There is a public exploit and a fix available. Probability of exploitation within the next 30 days (EPSS) equals 0.00129%.

CVE-2023-24329

An issue in the urllib.parse component of Python before 3.11.4 allows attackers to bypass blocklisting methods by supplying a URL that starts with blank characters.
CWE-20: Improper Input Validation
Found in image: amazonlinux_2:latest
CVSS v3.1 Base score: 7.5 (HIGH)
CVSS v3.1 Vector: Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N
Analysis:
This vulnerability affects a certain Python library inside the amazon RHEL image. If an application is not using Python, it is not exploitable. If an app would use it, it would enable an attacker to bypass the protections set by the developer for scheme and host. It only affects integrity. There is a public exploit and a fix available. Probability of exploitation within the next 30 days (EPSS) equals 0.0007%.

3) Demo of a vulnerable application: In order to actually exploit certain vulnerabilities found by scanners, a malicious actor would first somehow need to gain shell access inside the container (CVSS attack vector: local). The following example shall demonstrate this using a simple vulnerable node.js application, based on [17]. This application is vulnerable to path traversal and remote code execution:

```
const { exec } = require("child_process");
...
app.get("/", (req, res) => {
  if (req.query.q) {
    exec(req.query.q,
    function (error, stdout, stderr) {
      res.send(stdout);
    })
  }
})
```

The listing shows a node.js web application controller which accepts incoming HTTP connections on "/". It will pass a given query parameter "q" from the URL directly to a system command in *exec(req.query.q,...)* (Injection) and returns the output of the command in the HTTP response. This way, an attacker could craft special payloads in order to exploit this vulnerability. For example:

```
curl localhost:3000?q=ls
```

This command would return the directory contents from within the container, which is already very bad regarding confidentiality. Using *netcat*, a special TCP program available in most Linux distributions, it would be possible to spawn a reverse shell to an external server, giving this server full shell access inside the container, with even more consequences regarding integrity and availability:

```
# payload:
$ curl localhost:3000?q=\\
nc%2099.222.11.33%204445%20-e%20/bin/bash

# attacker listens on a remote machine
# for incoming connections:
$ nc -lvp 4445
```

[17]https://github.com/mwager/nodejs_exploit

As soon as this payload gets executed, the remote machine accepts the incoming connection and gained full shell access. If the image was built without an explicit USER command, the attacker would have root access inside the container, which is the same as having root on the host. Unfortunately, Docker images are built as root by default and root inside the container is the same root as on the host [4].

Finally, building the image using google distroless, with only adding 3 lines to the *Dockerfile*, this attack would not be possible anymore, because the application could not spawn a shell and would just throw an exception:

```
# stage 2 - switch to distroless
FROM gcr.io/distroless/nodejs18-debian12:nonroot
COPY --from=base /base /base
WORKDIR /base
```

The example shows that vulnerabilities also need to be analysed in the context of a certain application.

V. Results

This chapter will present the results of the research with respect to the two research questions.

A. Amount and Severity of Vulnerabilities

This section will present the results regarding the amount and severity of findings in order to answer RQ1 (1).

Category	Amount
OS-based	531
Runtime-based	32

Table I
Amount of vulnerabilities by category

Table I shows the distribution of vulnerabilities by category, clearly as expected the amount of OS-based is way higher. Public exploits exist only for 48 vulnerabilities, of which 45 for OS-based and 3 for the Runtime-based ones.

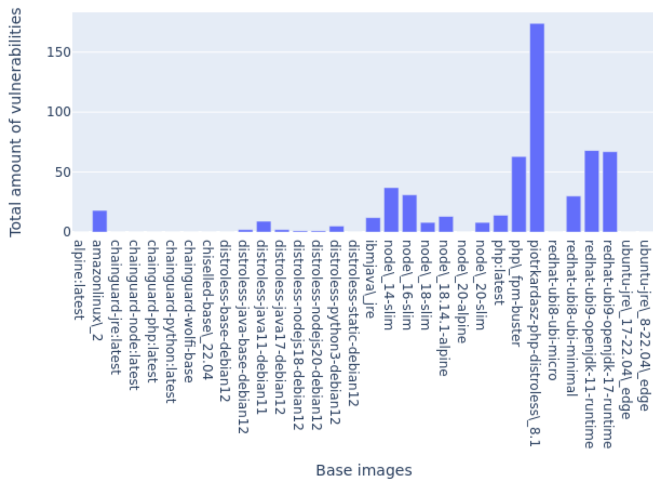

Figure 6. Total amount of vulnerabilities

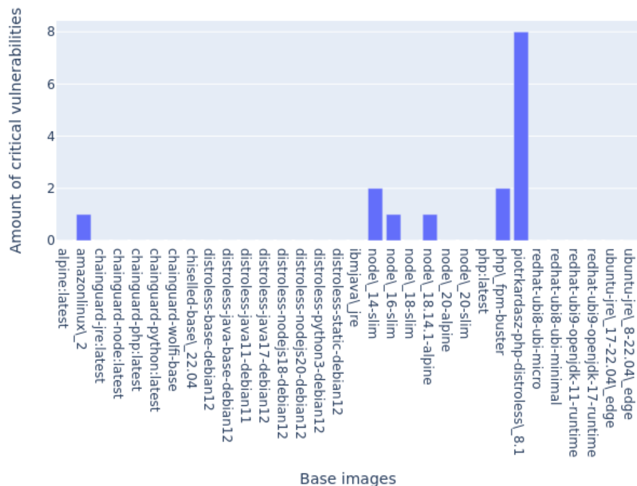

Figure 7. Amount of vulnerabilities with critical severity

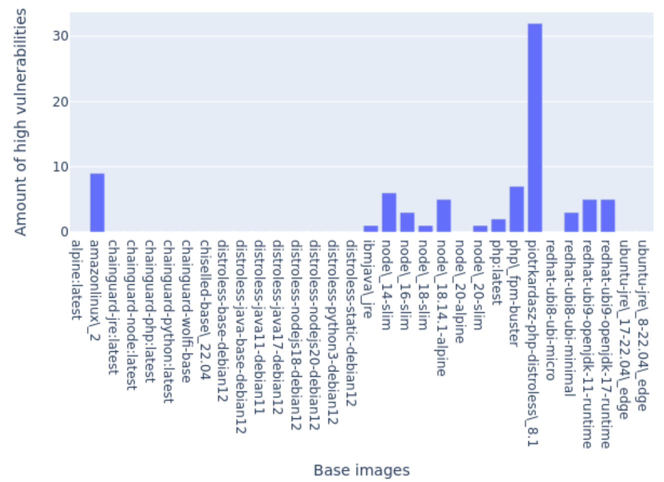

Figure 8. Amount of vulnerabilities with high severity

Figures 6, 7 and 8 visualize the total amount and the amount of critical and high vulnerabilities for all scanned images. Especially the PHP-, Node.js- and Java-images show a high amount of findings when not using component reduction tools. Alpine, most google distroless, UBI micro and especially all chainguard images do not contain any findings. The google distroless images for java contain findings so it was possible to reproduce the work of [15]. Also their conclusion that alpine is a direct competitor was reproduced. To add to the work presented in the related paper, we compare not only google distroless with some debian related images, but also all the other presented component reduction tools, additionally comparing images from RedHat, which also contain a fair amount of findings. The distroless fork of PHP contains a very high amount, but this is a fork of a master student and seems not to be maintained anymore. It was added because google distroless does not officially support the PHP runtime, but now it is clear that this fork should not be used. Alternatives like the chainguard PHP image performs way better (zero findings).

Image	total	critical	high	medium	low
alpine_latest:latest	0	0	0	0	0
amazonlinux_2:latest	15	1	7	7	0
chainguard-jre_latest:latest	0	0	0	0	0
chainguard-node_latest:latest	0	0	0	0	0
chainguard-php_latest:latest	0	0	0	0	0
chainguard-python_latest:latest	0	0	0	0	0
chainguard-wolfi-base:latest	0	0	0	0	0
chiselled-base_22.04:latest	0	0	0	0	0
distroless-base-debian12:latest	3	0	0	0	3
distroless-java-base-debian12:latest	2	0	0	0	2
distroless-java11-debian11:latest	9	0	0	0	9
distroless-java17-debian12:latest	2	0	0	0	2
distroless-nodejs18-debian12:latest	4	0	0	0	4
distroless-nodejs20-debian12:latest	4	0	0	0	4
distroless-python3-debian12:latest	9	0	0	1	8
distroless-static-debian12:latest	0	0	0	0	0
ibmjava_jre:latest	12	0	1	3	8
node_14-slim:latest	37	2	6	4	25
node_16-slim:latest	31	1	3	2	25
node_18-slim:latest	8	0	0	2	6
node_18.14.1-alpine:latest	13	1	5	7	0
node_20-alpine:latest	0	0	0	0	0
node_20-slim:latest	8	0	0	2	6
php_fpm-buster:latest	63	2	7	7	47
php_latest:latest	14	0	0	2	12
piotrkardasz-php-distroless_8.1-debug	178	9	29	90	50
redhat-ubi8-ubi-micro:latest	0	0	0	0	0
redhat-ubi8-ubi-minimal:latest	32	0	3	14	15
redhat-ubi9-openjdk-11-runtime:latest	71	0	5	45	21
redhat-ubi9-openjdk-17-runtime:latest	70	0	5	44	21
ubuntu-jre_17-22.04_edge:latest	0	0	0	0	0
ubuntu-jre_8-22.04_edge:latest	0	0	0	0	0

Table II
All scanned images and their vulnerability distribution

To have better visibility of the exact amount of findings per image, table II shows all scanned images and their vulnerability distribution. In total, 32 images were scanned, containing 564 vulnerabilities (critical: 16, high: 80, medium: 234, low: 234). The amount of images without **any** findings equals 13 (40.63%). The **base images** of all 5 presented component reduction methods have **zero findings**. Only the google distroless Java, Node.js and Python images contain a very small amount of especially low and medium severity findings. What stands out is that all chainguard images for all relevant runtime contain zero findings, making those the best performing candidates.

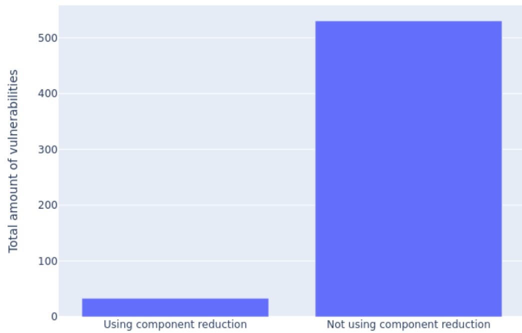

Figure 9. Amount of vulnerabilities when component reduced vs. not component reduced

Figure 9 shows the total amount of vulnerabilities of images using component reduction vs. images not using it. Only 33 of a total of 563 (5.86%) vulnerabilities are left when using component reduction methods, being a significant reduction of security vulnerabilities. Note that these 33 only coming from the google distroless images and are only of low and medium severity.

B. Exploitability of Vulnerabilities

This section will summarize and present the results of the research regarding RQ2 (2).

Attack vector	**No. of vulnerabilities**
Network	345
Local	176
Adjacent	0
Physical	0

Table III
Attack vector metrics

Attack complexity	**No. of vulnerabilities**
Low	415
High	106

Table IV
Attack complexity metrics

The two tables III and IV show the exploitability metrics from the CVSS base metrics, specifically **attack vector** and **attack complexity**. Note that for some vulnerabilities, no CVSS data was available. Around 66% of findings are, according to the CVSS base metrics alone, exploitable via network and around 69% have a low attack complexity. This looks quite surprising at first, but manual analysis in chapter IV-C has shown that even though vulnerabilities are exploitable via network according to the CVSS base metrics, exploitation is still often very unlikely.

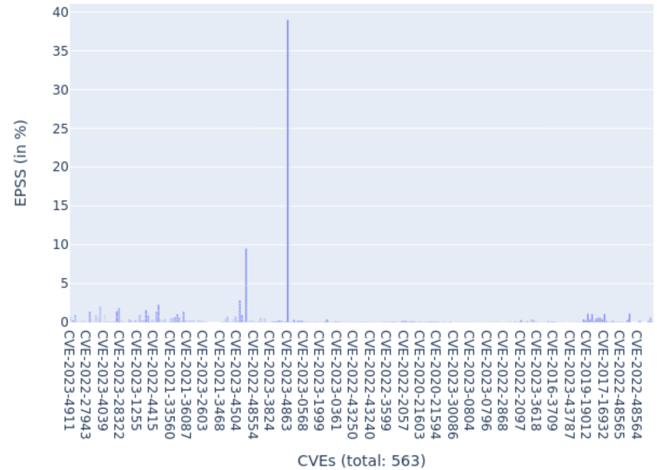

Figure 10. EPSS exploitation probability over found vulnerabilities

Figure 10 shows the EPSS exploitation probability in percent over all found vulnerabilities. Only one of

the vulerabilities, CVE-2023-4863, has an EPSS score of 38.98%, one is around 9%, two are around 4.7%, 5 are around 1% and the whole rest score **below 1%**. So according to the EPSS system, the answer to RQ2 is clearly "No". The exploitation probability within the next 30 days of most of the findings scores below 1%, mostly way below 1% (like e.g. 0.00017%).

VI. Discussion

This chapter will summarize the results and discuss potential conclusions.

A. RQ1

The research question RQ1 was: *Does the reduction of components significantly reduce the amount of vulnerabilities within the container image?*. This question can be answered with a clear Yes.

The images of all presented component reduction methods contain either no vulnerabilities at all, or at least a significantly lower amount compared to the other ones. No findings of critical or high severity were found in any of the component-reduced images. All statements of the papers from the related work chapter stating that component reduction leads to a smaller attack surface can be reproduced. For example, paper [9] mentions that *Firstly, secured base-images propagate less vulnerabilities to applications, thus the attack surface in terms vulnerability exploitation and impact regarding CIA can be kept minimum.* This can clearly be confirmed now with respect to the presented results.

B. RQ2

The research question RQ2 was: *Are typical vulnerabilities found through container security scanners actually exploitable and therefore a risk to the application?*. This question can be answered with a clear No.

Looking only at the CVSS base metrics, like unfortunately lots of organisations are doing, would paint a different picture. But EPSS and manual analysis of a manually picked set of findings showed that a) exploitation probability is **very low (below 1%)** and b) most of the manually analysed vulnerabilities are false positives, are extremely unlikely to be exploitable or need to be analysed again in the context of a concrete application. As stated in [16] *The CVSS base score alone is a poor risk factor from a statistical perspective* and *the existence of proof-of-concept exploits is a significantly better risk factor* this paper found the EPSS score during research which makes sense to use as main argument for answering RQ2. Furthermore, reporting a lot of false positives applies to all types of scanners. They often report the worst case, based on CVSS. While they often detect potential vulnerabilities that could impact system security, it's important to note that not all of these vulnerabilities are exploitable. Frequently, exploiting the system requires chaining multiple vulnerabilities together rather than a single isolated issue.

A great amount of findings also would first need shell access to the container. If this is the case it would often make more sense to search for container escapes. Certainly some vulnerabilities could be used further for privilege escalation and container escapes. This is why using special secured host operating systems like Amazon Bottlerocket [18] are a best practise to prevent further compromise.

Even if many findings are not or only very unlikely to be exploitable, they still have to be analysed and remediated, at best also in the context of the application using the image. This is a very complex and time-consuming task and therefore using component reduction tools in the first place would certainly be better. Additionally, distroless tools without a shell would prevent all vulnerabilities of attack vector "local" as shown in the demo of a vulnerable application in IV-C3. If a container image does not have a shell or shell access configured, it might be more challenging for an attacker to directly access a shell within that container but not impossible. Attackers can attempt various techniques to gain shell access or execute commands within a container. The quote from Red Hat mentioned earlier in IV-A3 highlights a common scenario where attackers exploit vulnerabilities in a system or application to execute malicious code, including potentially bringing their "own" shell into memory. This could be done through techniques like stack overflows or other exploits. Additional security measures like cloud workload protection to prevent and log unauthorized shellcode execution are needed and still recommended.

VII. Conclusion

The overall goal of this paper was to get a better decision basis for development and security teams in order to better support the vulnerability remediation process of container images. We now clearly have a better base of decisions: Component reduced images make applications more secure and significantly reduce the attack surfaces of containers. Generally, they are very simple to integrate into the development life cycle, just in some rare cases more knowledge and workarounds are needed. They additionally support the DevSecOps process in various ways because of their small size. Developer experience is good and the low size of the images additionally lead to lower costs and better resource efficiency. Motivation and knowledge of security unfortunately is often low within application teams, so management buy-in and good awareness and communication channels between security teams and developers is crucial.

Of the five evaluated tools, especially **chainguard images** and **google distroless** stand out. Both drastically reduce the vulnerability findings of image scanners (chainguard even having zero findings for all runtimes), are easy to use and well documented, free to use and do not have any licence restrictions - so these are really best conditions

[18]https://aws.amazon.com/bottlerocket/

for using them. Chainguard even supports Software Bill of Materials (SBOM) generation, optimizing supply chain security, a topic being more and more important in recent years.

Even though most of the findings have a very low probability of exploitation, they still should get remediated, so it still makes more sense to use component reduction in the first place in order to save time and money. Especially analysing scanner findings in the context of an application can be very complex and time consuming, the whole code base has to be reviewed in relation to the vulnerability. Also, knowing now that CVSS alone is not the best criteria for prioritizing vulnerabilities to remediate and more kind of a compliance exercise, additional measures like EPSS could and should be used by security teams. The recently released new version 4 of CVSS could also help, as it attaches greater importance to the environmental and temporal metrics.

VIII. Abbreviations

CPU Central Processing Unit

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

EPSS Exploit Prediction Scoring System

NPM Node Package Manager

NVD National Vulnerability Database

OS Operating System

SBOM Software Bill of Materials

References

[1] "Google distroless on github." [Online]. Available: https://github.com/GoogleContainerTools/distroless

[2] "Static vulnerability analysis of docker images." [Online]. Available: https://iopscience.iop.org/article/10.1088/1757-899X/1131/1/012018/pdf

[3] "An attack surface metric," 2008. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/2008/CMU-CS-08-152.pdf

[4] L. Rice, "Container security," 2020. [Online]. Available: https://learning.oreilly.com/library/view/container-security/9781492056690/

[5] "Container security workshop." [Online]. Available: https://smarticu5.github.io/assets/talks/Steelcon-Container-Security-Workshop.pdf

[6] "Container escape: All you need is cap (capabilities)." [Online]. Available: https://www.cybereason.com/blog/container-escape-all-you-need-is-cap-capabilities

[7] S. P. K. Karl Matthias, "Docker praxiseinstieg," 2020. [Online]. Available: https://www.mitp.de/IT-WEB/Programmierung/Docker-Praxiseinstieg.html

[8] "Cvss specification." [Online]. Available: https://www.first.org/cvss/v3.1/specification-document/

[9] "Well begun is half done: An empirical study of exploitability and impact of base-image vulnerabilities," 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9825857

[10] "Prismacloud twistcli container security scanner." [Online]. Available: https://docs.paloaltonetworks.com/prisma/prisma-cloud/prisma-cloud-admin-compute/tools/twistcli_scan_images#_scan_images_with_twistcli_Dockerless_scan

[11] "Prismacloud cvss scoring calculation." [Online]. Available: https://docs.paloaltonetworks.com/prisma/prisma-cloud/prisma-cloud-admin-compute/vulnerability_management/cvss_scoring

[12] "Nist application container security guide," 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

[13] "Enhancing and integration of security testing in the development of a microservices environment," 2020. [Online]. Available: https://www.utupub.fi/bitstream/handle/10024/150701/orazi_master_thesis.pdf

[14] "An analysis of security vulnerabilities in container images for scientific data analysis," 2021. [Online]. Available: https://academic.oup.com/gigascience/article/10/6/giab025/6291571?login=false

[15] "Base systems for docker containers - security analysis," 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9911523

[16] "Comparing vulnerability severity and exploits using case-control studies." [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2630069

[17] "A peek "under the hood" of modern vulnerability management." [Online]. Available: https://www.cisco.com/c/dam/global/en_hk/assets/pdfs/cisco-how-kenna-works.pdf

[18] "Enhancing vulnerability prioritization: Data-driven exploit predictions with community-driven insights." [Online]. Available: https://arxiv.org/pdf/2302.14172.pdf

[19] "Cisa known exploited vulnerabilities catalog." [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

[20] M. K. Bernd Öggl, "Docker," 2021. [Online]. Available: https://www.rheinwerk-verlag.de/docker-das-praxisbuch-fuer-entwickler-und-devops-teams/

[21] "Google distroless php fork on github." [Online]. Available: https://github.com/piotrkardasz/php-distroless

[22] "Redhat ubi micro." [Online]. Available: https://www.redhat.com/en/blog/introduction-ubi-micro

[23] "Ubuntu chisel." [Online]. Available: https://github.com/canonical/chisel

[24] "Chainguard images." [Online]. Available: https://edu.chainguard.dev/open-source/wolfi/overview/

[25] "Vagrant virtual machine environments." [Online]. Available: https://www.vagrantup.com/

[26] "Sysdig 2020 container security snapshot: Key image scanning and configuration insights." [Online]. Available: https://sysdig.com/blog/sysdig-2020-container-security-snapshot/

[27] "Cve api." [Online]. Available: https://nvd.nist.gov/developers/vulnerabilities

[28] "Searchsploit / exploitdb." [Online]. Available: https://www.exploit-db.com/searchsploit

[29] "Epss api." [Online]. Available: https://www.first.org/epss/api