# Docker: Advantages and Security Implications on a Python Client-Linux Application

ALEXANDRA DIANA STEFANIA MURGEA, University of Twente, The Netherlands

Over the years, Docker has gained a high level of attention among enterprises, individual users and researchers. With its success, the number of vulnerabilities and malicious intents increased, raising security concerns among the public. Numerous studies have already been conducted that aimed to identify, categorize, assess or mitigate these vulnerabilities. Although more scarce, some studies also discussed the practical process of dockerization, however with a focus on web services or data processing software. This paper aims to blend elements of a literature review and an empirical study to offer an overview of the advantages and security impacts of Docker and present new insights from dockerizing and performing a security assessment on a Python Client-Linux application. Furthermore, we investigate the feasibility of introducing some recommended security best practices in the context of our application. The chosen running environment for the application is a Raspberry Pi 4 and the security assessment is performed using a security tool called Snyk.

Additional Key Words and Phrases: Docker, security impact, Python application, vulnerability assessment

## 1 INTRODUCTION

Nowadays, cloud-native systems are among the most widely used platforms for application development and deployment, offering several benefits such as lightweight virtualization, resource efficiency and streamlined DevOps processes [19, 27]. Numerous services like Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS), use virtualization techniques like hypervisor or containerization to pool and share resources across networks [3, 19].

As the industry evolved new requirements emerged, striving for tighter development cycles, cost-efficient infrastructures and regular product deliveries which generated an increase in the adoption of containers [14]. Compared to other widely employed alternatives like virtual machines (VMs) that rely on hypervisor-based virtualization and offer higher data security, containers provide a level of performance close to bare-metal and the ability to execute many versions of an application smoothly on the same computer [14, 18]. Their flexibility expanded their use to various application domains such as Internet of Things (IoT) services, smart cars, service meshes and many more, catching the attention of large-scale organizations like Amazon, Spotify and Netflix [23].

From the perspective of market popularity, Docker encountered the highest response, reaching the level of "market leader among all container solutions" according to [18]. However, as the industry demand and usage of Docker increased over time, its security implications became a subject of interest among many enterprises, individual users and researchers. In fact, the security concerns that containers face nowadays, among which we may list attackers pilfering confidential information, denial of service attacks on application containers, and privilege escalation to abuse the underlying host resources, constitute one of the main obstacles to their broader adoption [23, 26, 27].

Previous works exist on Docker and other containerization techniques which aimed to identify, categorize or find mitigation solutions to different vulnerabilities, to assess or employ various security tools or frameworks, or to conduct surveys or studies on Docker images and container security. Although more scarce, some studies also discussed the practical process of dockerization. However, the focus was often placed on applications related to web services or data processing software. This paper aims to blend elements of a literature review and an empirical study in order to offer a general overview of the advantages and security implications of using Docker and present new insights from dockerizing and performing a security assessment on a Python Client-Linux application.

By Python Client-Linux application we describe a Python written program which runs on a Linux operating system (in our case the Raspberry Pi OS) and serves as a client in a client-server architecture with the goal of requesting and receiving services or information from the server. It is also important to note that IoT applications such as ours are tightly coupled to the hardware which made it challenging to address some of the application specific requirements during the analysis [8].

### 1.1 Research question

To guide the research process, the following main research question (MRQ) has been formulated, which was further sustained by three sub-research questions (SRQ):

> **MRQ:** What is the effect of dockerization on a Python Client-Linux application?
> **SRQ1:** What are the general advantages of using Docker?
> **SRQ2:** What are the most common Docker security vulnerabilities?
> **SRQ3:** To what extent can security guidelines be employed in the Dockerization process?

To answer these research questions, a mixed approach was employed: SRQ1 and SRQ2 were answered fully through a literature review, whereas SRQ3 implied both a literature review on existing security guidelines and a hands-down dockerization process and security assessment.

The rest of this paper is organized as follows. Section 2 constitutes the literature review part of the research including a short introduction to Docker, an overview of its general benefits, a discussion on existing research on Docker's security vulnerabilities and a summary of possible mitigation solutions. Section 3 describes the steps taken to dockerize the application and explains why and how the

*TScIT 41, July 5, 2024, Enschede, The Netherlands*

security scanner was used. Section 4 presents an overview of the results gathered from the security assessment. Section 5 discusses and analyzes the obtained results in comparison to the literature. Section 6 concludes the paper by providing an answer to the research questions and offering suggestions for further work.

## 2 LITERATURE REVIEW

The literature review was completed using three search engines, namely: IEEE Xplore, Google Scholar and Scopus. The criteria for choosing the papers involved the level of relevance, as well as the publication year, encompassing papers from the time frame marked by the debut of Docker to the present, namely from 2013 to 2024, with a preference for more recent works. The later was considered an important factor particularly when addressing security vulnerabilities given the continuously evolving nature of both threats and mitigations.

### 2.1 Docker and basic concepts

To provide some context, the concept of virtualization refers to the creation of a simplified abstract environment for software programs which allows them to be moved or redeployed on any platform that provides the same virtual environment [17]. Although containers have been present for a long time, it was not until the debut of Docker in 2013 that this type of technology started gaining more and more interest [15]. Docker is an open-source virtualization technology that allows system architects and software developers to install, test, and deploy applications in various operating system environments [20]. At the foundation of Docker we have two essential components: the virtualization solution known as Docker engine and the Docker Hub platform which is a cloud-based registry service dedicated to the storage and sharing of Docker images [18].

Compared to other container alternatives like LXC, OpenVZ or Linux-Vserver, Docker distinguished itself through factors such as speed, simplicity, and the existence of a standardized Dockerfile format for creating and managing the software containers [14, 17]. Without changing the program's underlying architecture, the deployment of an application into the Docker platform is possible by simply preparing an instruction file, also known as Dockerfile [12]. This Dockerfile consists of a set of instructions used to create a Docker image by building stacked layers, each describing the files that are added, modified or removed by the Dockerfile commands [17]. Once an image is built, the application can run inside a newly created instance that gets further referred to as a container [29].

### 2.2 Advantages of using Docker

In order to better understand the benefits brought by container technologies, it is important to understand the context that drove some of the changes in implementation. We will start by discussing their counterpart, namely the hypervisor-based virtualization which has been and continues to be widely employed in the world of cloud computing.

Virtual machines or VMs are the most popular form of hosted hypervisor virtualization and are well known for offering an isolated, reliable and fully operational environment for software applications [19]. However, their execution entails the usage of the host machine's resource pool and the creation of a new copy of the OS, libraries and programs each time a new instance is created [23]. As a result, the host needs to accommodate two operating systems along with the newly introduced virtual layer, thus affecting the overall performance [14].

Container technologies emerged to address these concerns, their design enabling the sharing of the operating system kernel and resources with the host [23]. What these technologies have in common is that they provide smaller footprints, increased scalability and performance, as well as the ability to package application binaries along with their dependencies and libraries into standalone artefacts [24, 25]. Their characteristic of being lightweight resulted from their lack of kernel and some system libraries, which reduced image sizes to a few megabytes and sped up the boot process [14]. According to Sultan et al. (2019) [23], this difference can be rather significant given that "a container can start in 50 milliseconds while a VM might take as long as 30–40 seconds to start".

Despite the previously mentioned features, it was Docker's specific implementations and additions which allowed it to dominate the market over the other competing containerization alternatives. Among these, we can name: the ability to produce and sustain the largest number of virtual environments on a physical hardware, the presence of a standardised Dockerfile format, increased compatibility with third-party softwares, as well as, the presence of its unique Union File System [18]. The latter allows for the verification of the stored images originating from DockerHub and provides accessibility to different file system in a consistent and centralized manner [18].

Lastly, Docker can also prove itself beneficial in a company context given its potential to reduce software development and shipping time [18]. Normally, when moving from one computing environment to another, be it across devices, teams or into production, it is required to install all necessary services and dependencies on the new host machine. However, this process is time-consuming and prone to errors or misunderstandings generated from unclear instructions or incompatibilities with versions, related libraries, security guidelines, network architecture or storage [19]. Docker images, on the other hand, are portable and have all services and dependencies needed already packaged inside, requiring only to run one command for each application to pull the image and start it inside a container.

### 2.3 Security implications

Although virtual machines are considered to provide a lower performance when compared to containers, their counterargument is that they employ better data protection. The applications executed from within a virtual machine are restricted to only be able to access the VM's kernel, thus adding an extra layer of defence between the host and applications [18]. On the other hand, Docker's security relies on two Linux-specific features: namespaces which isolate OS resources and provide an autonomous environment for container processes, and cgroups which regulate and limit resource utilization [9, 18].

Despite seemingly safe, because they combine and isolate all dependencies into one single component, the kernel-sharing characteristic makes containers and therefore the host, vulnerable to different security attacks [19, 30]. As a result, the security implications of Docker and similar containerization techniques have been approached in different manners among the research community.

Some researchers have laid a foundation by identifying and categorizing possible vulnerabilities associated with Docker. For instance, the survey presented by Sultan et al. (2019) [23] discusses six main risk categories in respect to containers, namely involving: images, registries, orchestration, containers, host OS and others. The four different scenarios discussed host protection from containers, safeguarding a container from its own applications, inter-container security, and defence against dishonest or malevolent hosts [23]. The study made by Patra et al. (2022) [18] uses a similar threat modelling strategy, however, with the addition of a fifth use case dedicated to docker engine protection from applications, containers, and hosts. In contrast, Martin et al. (2018)'s [14] analysis considers a different vulnerability division across five categories based on the distinct layers of the Docker ecosystem: weaknesses within images, threats related to Docker or lib-container, unsafe production system configuration, vulnerabilities in the transmission, validation or decompression of images, and vulnerabilities in the Linux kernel.

Another category of papers has focused on assessing different security frameworks such as STRIDE, DockerChannel, CONSERVE or SEAF designed to find or evaluate threats that may result in security breaches or information leakages such as root access misuse of host resources, DoS attacks, and the gain of unauthorized access to sensitive data, source codes, and passwords [4, 5, 10, 26]. A case study by Wang et al. (2023) [25] even showcases their own prediction-based online anomaly detection model, which models prediction errors and identifies abnormalities using LSTM neural networks and a dynamic sliding window technique.

The evolution in the number and impact of security vulnerabilities posed by Docker images has also been under investigation. A longitudinal study conducted by Mills et al. (2023b) [16] on 380 container images during three different time-frames revealed that despite the increase in the number of vulnerabilities with time, less than 1% of them were deemed as high risk. Meanwhile, other studies emphasised the need for more updates of the third-parties packages contained within the images and the potential for de-duplicating registries and reducing image sizes by eliminating redundant files [28, 29]. In addition, Lin et al. (2020) [13] raises attention to a few more problems like the use of outdated base OS images, improper and troublesome use of the "latest" tag and disregard for image versioning.

## 2.4 Security guidelines

As a response to the security risks identified in the different Docker components, researchers have taken the initiative to examine potential mitigation techniques and propose best practices in order to minimize the effects of using this technology.

Among the most common exploits, denial of service (DoS) attacks can do serious harm to Docker containers by overloading them with requests or overusing the host resources, which can cause system failures, slowdowns and render other programs and services inoperable [11]. To mitigate this risk, it is advisable to configure resource quotas, a process targeted to restrict the amount of memory and processing power a container can utilize, such that less system resources can be accessed by the malicious code [18].

On a more general level, it is recommended to ensure periodic update of Docker, the host OS and the base image to include the newest security changes [18]. Vulnerability scans should also be employed regularly on both images and applications to identify and address potential issues early on [23, 26]. These solutions can prevent remote code execution, container escape attacks and DoS [23].

Measures can also be taken in the early stages of creating the Dockerfile to limit the overall attack surface which maximizes its effect by either: following a multi-stage build approach where the Dockerfile is structured to have a stage for building the image and another for running, using an intermediate container that transfers only the essential binaries and dependencies or by choosing a distroless image which is essentially a minimal version that lacks a few components like shells and package managers [26]. In addition, Wong et al. (2023) [26] also advises users to pay attention to the origin of any image fetched from DockerHub, opting for "official" images over “verified”, “certified” and “community”. Using untrusted images can lead to an attack on all containers within the host [23].

Another aspect worth considering is running the container with the least level of privilege whenever feasible and trying to avoid dangerous docker run options by limiting file permissions, configuring TLS for DockerHub or hardening host configuration [18, 26, 30]. This approach is efficient towards preventing tampering, DoS attacks and gaining unauthorized access that may lead to resource depletion or data breaches [23].

Lastly, the network and API are crucial to Docker security since the containers employ them for communication, hence it is advised tracking their status and enabling communication among containers only when necessary to minimize the chance of ARP spoofing, DoS attacks and network-based intrusion [18, 23].

## 3 METHODOLOGY

Through this experiment, we aimed to better understand the process of dockerization and its effects on the security level of a Python Client-Linux application in order to make a comparison with the currently available information and hopefully provide some new insights. To commence, it is important to note that the chosen running environment for the application was a Raspberry Pi 4 and the security assessment was performed using a security tool called Snyk.

## 3.1 Dockerization process

In order to set up the environment, the first step involved installing Docker on the Raspberry Pi which was achieved by following the instructions on the official website under the Debian section. Each operating system has a dedicated set of steps and different installation alternatives, however we opted for using the AP repository. At this step, it is recommended to run the basic "hello-world" image which will display a confirmation message to ensure that Docker works as intended.

*3.1.1 Requirements file.* In real-world scenarios, most applications make use of different third-party libraries or need to import functions from different classes. In our context, these are the so-called libraries and dependencies which need to be packaged together with the application in the image for a correct execution. Fortunately, this process can be automated by activating the virtual environment and running the "pip freeze < {file-name}" command which will generate a list of all installed packages in the current environment and redirect the output to a file. These commands should be run in the home directory of the application.

*3.1.2 Dockerfile.* The next step involved the creation of the Dockerfile. It is important that the name is not modified and no extension is added, otherwise the file will not be recognized. An overview of the general structure of the Dockerfile used for the dockerization of our Python Client-Linux application can be seen in Figure 1. The first line "FROM {image-name}" will pull a base image from the Docker public repository Docker Hub which will serve as the first layer of the image. It is possible to just include the name of the application, in our case "python" and that will pull the latest version available, however, as the application was run using python:3.9, we opted to use the same version. The "WORKDIR /directory-name" instruction is responsible for setting the default directory where commands will be run inside the container. Next, the command "COPY" is responsible for copying files and/or directories from the host into the file-system of the image. In our case, the dot points to the current directory on the host whose contents will be copied to the home path in the image. The image and, therefore, container will not be able to access anything situated outside of this directory that could be found on the host machine.

The instructions of type "RUN" are used to carry out any shell operation that can be performed in the terminal, including those for modifying environment variables and installing software. An example can be found in the following two lines which are both meant to install the necessary dependencies that the application needs inside the container. Often, the latter command would suffice, however, there are cases where specific dependencies cannot be installed via "pip" and generate an error during the building stage of the image and therefore need to be installed separately using "apt-get install". The following three commands aim to create a new non-root user, assign it the necessary permissions for the desired directory and specify that this user will responsible for the execution of the remaining commands using "USER {user-name}" instruction . This is one of the security practices which will be discussed in section 5.

Sequentially, the "ENV" commands are responsible for setting environmental variables which are meant to change the behaviour of the application. The "PYTHONPATH" ensures that the specific path is included in the search for imports. The other environmental variables are very specific to the current application and therefore out of the scope of this discussion. Finally, the "CMD" instruction specifies which command should be executed when the container is started, usually the one for starting the application. The difference between the "CMD" and the "RUN" command is that the latter is run during the building stage and adds a new layer each time to the image, while the former describes the runtime behaviour without adding any new layer to the image.

```
# Official Python base image
FROM python:3.9-slim

# Working directory inside the container
WORKDIR /home

# Copy necessary files from host into the container
COPY . /home

# Install dependencies which are not available via pip
RUN apt-get update && \
    apt-get install -y \
        dependency 1 \
        dependency 2 \
        ...

# Install dependencies from requirements.txt file
RUN pip install --upgrade pip && \
    pip install -r /home/requirements.txt

# Create a new user
RUN useradd -m -u 1000 app_user

# Give priviledges to the new user
RUN chown -R app_user:app_user /home

# Switch to new user
USER app_user

# Set environmental variables
ENV PYTHONPATH "${PYTHONPATH}:/path-to-import-modules"
ENV env1 = val1
ENV env2 = val2

# Command to run the application after the contaiiner is started
CMD ["python", "/path-to-main-application", "start"]
```

Fig. 1. Dockerfile overview

*3.1.3 Building the image and running it in a container.* The command for building the Docker image is rather straightforward "sudo docker build -t {image-name} .". The -t flag is used in order to give a name to the image which is also known as a tag and the " ." at the end implies that the Dockerfile is located in the current directory. After that, every command in the Dockerfile will be executed, creating the layers of the image. Lastly, to run a specific image inside a container, we made use of the "sudo docker run -v /path1:/path2 {image-name}" command. The -v /path1:/path2 option mounts /path1 from the host computer to /path2 inside the container. This was essential in the context of our application because of the need for file sharing regarding the contents of the USB.

## 3.2 Security Assessment

Security scanning tools are essential for locating and fixing vulnerabilities, hence proactively reducing the risk of exploitation in production environments by covering different techniques such as static, dynamic, software composition analysis (SCA), and container security scanning [21].

Multiple such tools are available, however, Snyk was chosen for our analysis as its functionalities seemed the most fitted to our context. Some relevant features offered by Snyk include: inter-container security scanning, precise vulnerability mapping, low false positive rate, security enhancing suggestions or even immediate fixes, an intuitive user interface and GitHub integration [1, 2]. Among the programming languages supported by Snyk we can name: Python, Ruby, Node.js, Java, and Scala [2].

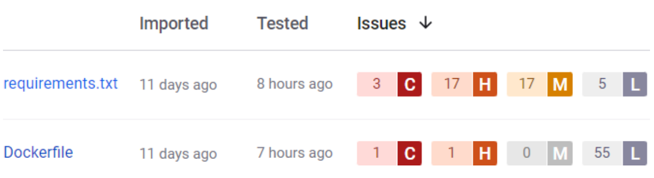
There are multiple ways of interacting with the tool including the CLI, IDEs, API, however, we opted for logging in to the Snyk Web UI platform through GitHub account linkage.

According to the documentation, Snyk Risk Scores are automatically applied for any vulnerability-type issues, allowing for a risk-based prioritization strategy and calculated based on the threat's possible impact and exploitability level [22]. At the end of the scanning, the tool assigns each issue a risk score, ranging from 0 to 1000 [22]. This is further used to derive four levels of significance, namely critical, high, medium and low in an equally distributed manner (intervals of 250) [22]. In addition, the interface provides a list view which allows the users to inspect each vulnerability individually, including aspects such as vulnerability name, issue severity, security score, exploit maturity issue type and more. Examples of how the overall security assessment could look like, as well as, figures of detailed views of threats and generated security improvement suggestions can be found in section 4.

As a final step, we performed an analysis on the results by looking into each vulnerability's CWE (Common Weakness Enumeration) or CVE (Common Vulnerabilities and Exposures) code to find out the use case, how it can be exploited and which solutions could prevent the attack from happening. These information were gathered from the two official websites [6, 7] of the specific codes which can be found in the description of each vulnerability. More details on the results can be found in Section 5.

## 4 RESULTS

In this section, we included the output generated by the security assessment tool on the Dockerfile and requirements file (see Figure 2), as well as a few examples, showcasing detailed views of some of the vulnerabilities identified (see Figure 3 and Figure 4) and presenting two of the suggestion offered in order to improve some of the issues (see Figure 5 and Figure 6).

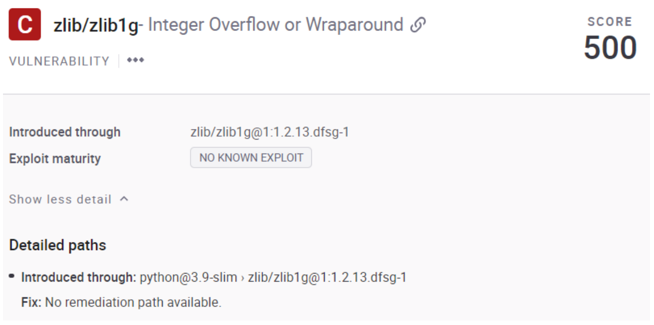


| | Imported | Tested | Issues ↓ |
|---|---|---|---|
| requirements.txt | 11 days ago | 8 hours ago | 3 C 17 H 17 M 5 L |
| Dockerfile | 11 days ago | 7 hours ago | 1 C 1 H 0 M 55 L |

Fig. 2. Security assessment result

## 5 DISCUSSION

The output generated as a result of the vulnerability scanning (see Figure 2) discovered a total of 99 issues in the Dockerfile and requirements file. Through a simple calculation we can observe that, 58% are deemed as low risk and less than 4% can be found in the highest risk category.



C zlib/zlib1g- Integer Overflow or Wraparound

VULNERABILITY

SCORE 500

Introduced through zlib/zlib1g@1:1.2.13.dfsg-1

Exploit maturity NO KNOWN EXPLOIT

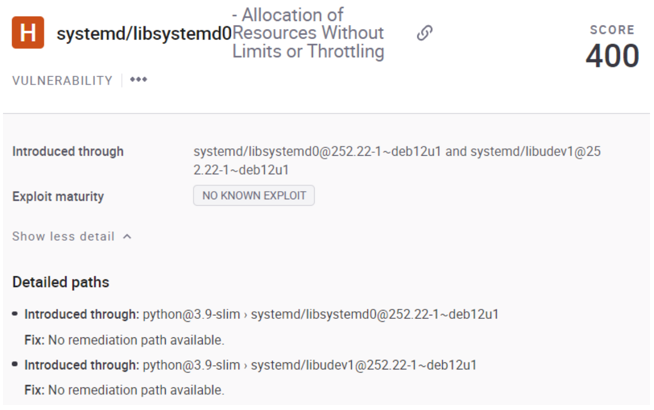
Show less detail

Detailed paths

- Introduced through: python@3.9-slim › zlib/zlib1g@1:1.2.13.dfsg-1

  Fix: No remediation path available.

Fig. 3. Detailed view Dockerfile critical vulnerability



H systemd/libsystemd0 - Allocation of Resources Without Limits or Throttling

VULNERABILITY

SCORE 400

Introduced through systemd/libsystemd0@252.22-1~deb12u1 and systemd/libudev1@252.22-1~deb12u1

Exploit maturity NO KNOWN EXPLOIT

Show less detail

Detailed paths

- Introduced through: python@3.9-slim › systemd/libsystemd0@252.22-1~deb12u1

  Fix: No remediation path available.

- Introduced through: python@3.9-slim › systemd/libudev1@252.22-1~deb12u1

  Fix: No remediation path available.

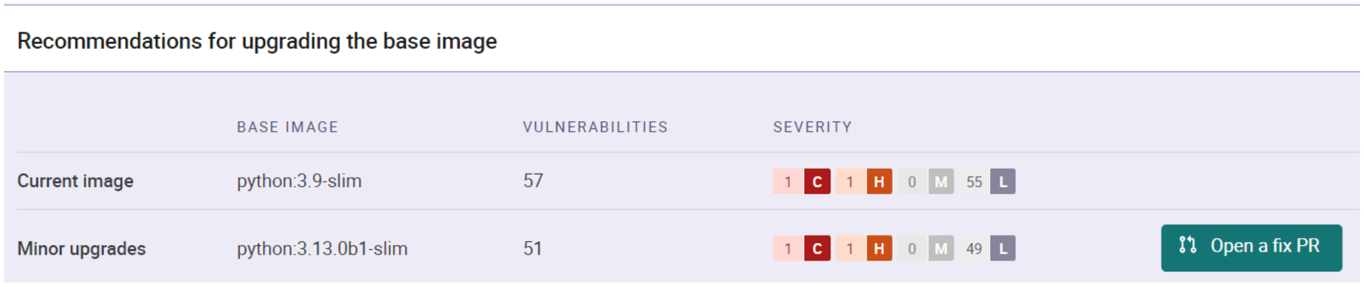
Fig. 4. Detailed view Dockerfile high vulnerability

### 5.1 Dockerfile security assessment

Focusing on the Dockerfile, 57 issues were identified: 1 critical, 1 high and 55 low. We will discuss in detail the two ranked the highest and summarize the findings for the low-risk category.

Firstly, the critical issue (see Figure 3) "CWE-190: Integer Overflow or Wraparound" is said to be introduced by zlib/zlib1g which is a version of a data compression library and can be caused by exceeding an integer type when performing an arithmetic operation. This vulnerability can have a significant impact on the application when used during memory allocation, data handling or buffer management given an attack can be triggered through user inputs. Possible exploits include remote code execution, DoS attacks or data corruption. The recommended mitigation strategies discuss input validation, revising the choice of certain function for safer alternatives and performing regular updates and code audits.

Secondly, the vulnerability ranked as high (see Figure 4) "CWE-770: Allocation of Resources Without Limits or Throttling" signalised that no restrictions are set on the amount of memory or resources which can be allocated to the container. This can give attacker the opportunity to utilize many resources, spamming a large number of requests or even monopolizing the entire quota preventing the other components from receiving their share. Among the security implications we can name DoS, system crashes and



Recommendations for upgrading the base image

| | BASE IMAGE | VULNERABILITIES | SEVERITY |
|---|---|---|---|
| Current image | python:3.9-slim | 57 | 1 C 1 H 0 M 55 L |
| Minor upgrades | python:3.13.0b1-slim | 51 | 1 C 1 H 0 M 49 L |

Fig. 5. Base image update

performance issues. Solutions to this problem include: configuring resource limits, monitoring processes and input validation.

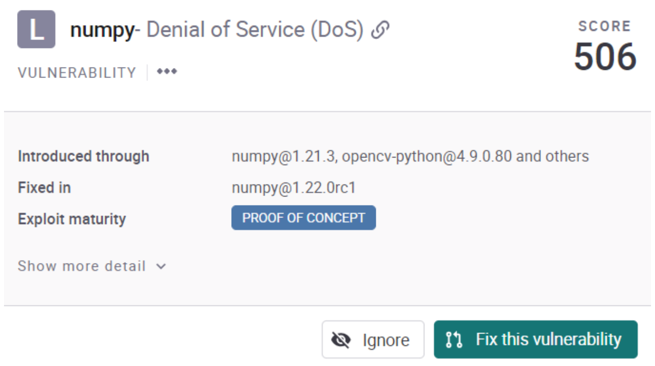
Finally, the other low-risk vulnerabilities were concerning: resource exhaustion, incorrect permission assignment, improper validation of certificates or ckecksums, out-of-bounds, uncontrolled recursion, link following, cryptographic issues, code injection, information exposure, improper input validation or authentication. Furthermore, the ways in which they could be exploited included: confidential wrap tokens, arbitrary code executions, modify or read memory and application data, DoS, privilege gain, protection mechanism bypass and data corruptions and employ similar mitigation strategies to the ones already mentioned.

Out of the 57 issues, 6 can be solved by choosing a more recent version of the base image as it can be seen in Figure 5.

## 5.2 Requirements file security assessment

In the case of the requirements file, the tool identified 42 issues: 3 critical, 17 high, 17 medium and 5 low. The three critical issues concerned heap-based buffer overflow, arbitrary code execution and improper following of a certificate chain of trust. Meanwhile, the high risk vulnerabilities included: injections, out-of-bound reads, licensing issues, DoS, uncontrolled resource consumption and buffer overflow.

Out of the 42 issues detected, 34 can be resolved through a simple update of the dependency used which is automatically suggested and fixable by Snyk (see Figure 6).



L numpy- Denial of Service (DoS)

VULNERABILITY

SCORE
506

| | |
|---|---|
| Introduced through | numpy@1.21.3, opencv-python@4.9.0.80 and others |
| Fixed in | numpy@1.22.0rc1 |
| Exploit maturity | PROOF OF CONCEPT |

Fig. 6. Dependency update

## 5.3 Literature comparison

During the dockerization process we could appreciate some of the benefits of using Docker. The Dockerfile format was convenient and simple to use, multiple instances could be created and ran on the same host and the time necessary to start the application from within the container was very short, almost imperceptible matching the statement provided by Martin et al. (2018) [14].

Furthermore, the vulnerabilities found as a result of the security assessment corresponded with the ones listed in the literature. Similar to the longitudinal study of Mills et al. (2023b) [16], we also found that most of the threats listed were deemed as low-risk, although his study included a larger range of risks thus rendering a lower percentage.

### 5.3.1 *Security guidelines feasibility.*

The other aspect which we wanted to investigate through the dockerization of the application was the feasibility of employing security guidelines. As mentioned at the end of Section 5.1 and Section 5.2, a simple update on the base image and application's dependencies was possible and had the potential to reduce a good amount of the vulnerabilities identified. The other measure that was applicable in our case was the configuration of resource quotas by adding the option "–memory" and "–cpus" when running the container.

On the other hand, limiting the privilege level of the container was only partially possible, due to some processes and components requiring root privileges and therefore we could not switch to a non-root user through the entire process. Lastly, the two-stage build with a distroless image was consider, however the attempt turned out unsuccessful. This could have happened due to a variety of reasons such as the tight coupling of the application with the hardware given the necessity to access the authorization files on the USB or perhaps because of the strict path dependencies required by the application.

## 6 CONCLUSION

In this paper, we dockerized and performed a security assessment on an application and compared our findings with existing literature. Furthermore, we investigated the feasibility and potential effect of introducing some of the recommended security best practices in the context of our application. Given the limited time frame, this research focused on dockerizing a single Python Client-Linux application. However, valuable insights were gathered from some of the characteristics which influenced the dockerization process and could serve as a base for similar applications.

The main research question for this paper was to investigate the effect of dockerization on a Python Client-Linux application. Our experiment revealed that, although Docker introduced additional vulnerabilities to the application, more than half of them were ranked as low-risk and only a small percentage may have a significant impact on the overall security level of the application.

(SRQ3) In addition, we discovered that some of the risks could potentially be eliminated or at least prevented by introducing security guidelines into the development process such as reducing the attack surface, restricting resource quotas, performing updates on the application and based image, employing minimal privileges whenever feasible, using a multi-staged build approach and monitoring the status and communication of the containers. However, it is important to note that depending on the application and its specific needs, some guidelines might not be feasible as it happened with our IoT application.

(SRQ1) Overall, the Docker technology brings many benefits to the users such as portability, increased performance, reduced footprints, scalability, simplicity of use, high compatibility, consistent filesystem and reduced development and deployment times. (SRQ2) However, it is important to be also aware of the possible threats and take them into account in the early stages in order to avoid various exploits such as: remote code execution, DoS attacks, privilege escalation, data or memory corruption, injections and information leakages.

## 6.1 Future work

Future work could extend this analysis to include a larger and more diverse set of IoT applications. The focus of such an experiment could be on gathering a more comprehensive understanding of the needs and behaviour of this category of applications in relation to the usage of Docker and to the different restriction which may appear. In addition, the problems which occurred with the highest frequency could be further investigated and solutions could be proposed, thus increasing the likeliness of users choosing Docker for their program.

Another interesting direction could focus on experimenting with different dockerization strategies, for instance trying different places of division or numbers of containers for the same application and comparing the scenarios via a security assessment. This idea would research how different levels of isolation, networking and container communication could affect the security of an application. Alongside, the amount of memory and resources should be considered as the number of containers increase in order to not significantly affect the performance.

## REFERENCES

[1] Aman Anupam, Prathika Gonchigar, Shashank Sharma, Prapulla SB, and Anala MR. 2020. Analysis of Open Source Node.js Vulnerability Scanners. *International Research Journal of Engineering and Technology (IRJET)* 07, 05 (May 2020), 2395–0056. arXiv:2395-0072 https://www.irjet.net

[2] Muhammad Abdullah Bin Ashfaq. 2023. *DEVELOPMENT OF A SECURITY TESTING PROCESS FOR YOCTO LINUX-BASED DISTRIBUTIONS.* Master's thesis. Tampere University.

[3] Nabeel Aslam, Arfan Shahzad, Ahsan Ali, and Mohammed Albin Ahmed. 2024. Modelling informal security requirements of cloud computing. *Multidisciplinary Sciences Journal* 6, 7 (Jan 2024), 2024104. https://doi.org/10.31893/multiscience.2024104

[4] Enrico Cambiaso, Luca Caviglione, and Marco Zuppelli. 2023. DockerChannel: A framework for evaluating information leakages of Docker containers. *SoftwareX* 24 (Dec 2023), 101576. https://doi.org/10.1016/j.softx.2023.101576

[5] Libo Chen, Yihang Xia, Zhenbang Ma, Ruijie Zhao, Yanhao Wang, Yue Liu, Wenqi Sun, and Zhi Xue. 2022. SEAF: A Scalable, Efficient, and Application-independent Framework for container security detection. *Journal of Information Security and Applications* 71 (Dec 2022), 103351. https://doi.org/10.1016/j.jisa.2022.103351

[6] MITRE Corporation. 2024. *Common Vulnerabilities and Exposures.* https://www.cve.org/ Accessed: 2024-06-29.

[7] MITRE Corporation. 2024. *Common Weakness Enumeration.* https://cwe.mitre.org/ Accessed: 2024-06-29.

[8] Wei Dong, Borui Li, Gaoyang Guan, Zhihao Cheng, Jiadong Zhang, and Yi Gao. 2020. TinyLink: A holistic system for rapid development of IoT applications. *ACM Transactions on Sensor Networks (TOSN)* 17, 1 (2020), 1–29. https://doi.org/10.1145/3412366

[9] Vipin Jain, Baldev Singh, and Nilam Choudhary. 2022. Audit and Analysis of Docker Tools for Vulnerability Detection and Tasks Execution in Secure Environment. In *Communications in Computer and Information Science*, Vol. 1591. 654–665. https://doi.org/10.1007/978-3-031-07012-9_54

[10] Rodi Jolak, Thomas Rosenstatter, Mazen Mohamad, Kim Strandberg, Behrooz Sangchoolie, Nasser Nowdehi, and Riccardo Scandariato. 2022. CONSERVE: A framework for the selection of techniques for monitoring containers security. *Journal of Systems and Software* 186 (April 2022), 111158. https://doi.org/10.1016/j.jss.2021.111158

[11] Haneul Lee, Soonhong Kwon, and Jong-Hyouk Lee. 2023. Experimental analysis of security attacks for Docker Container Communications. *Electronics* 12, 4 (2023), 940. https://doi.org/10.3390/electronics12040940

[12] Yu Li. 2020. Towards fast prototyping of cloud-based environmental decision support systems for environmental scientists using R Shiny and Docker. *Environmental Modelling and Software* 132 (Oct 2020), 104797. https://doi.org/10.1016/j.envsoft.2020.104797

[13] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. 2020. A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 371–381. https://doi.org/10.1109/ICSME46990.2020.00043

[14] A Martin, S Raponi, T Combe, and R Di Pietro. 2018. Docker ecosystem – vulnerability analysis. *Computer Communications* 122 (June 2018), 30–43. https://doi.org/10.1016/j.comcom.2018.03.011

[15] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (March 2014), 2. url={https://dl.acm.org/citation.cfm?id=2600239.2600241}

[16] Alan Mills, Jonathan White, and Phil Legg. 2023. Longitudinal risk-based security assessment of docker software container images. *Computers & Security* 135 (Dec 2023), 103478. https://doi.org/10.1016/j.cose.2023.103478

[17] D Morris, S Voutsinas, N C Hambly, and R G Mann. 2017. Use of Docker for deployment and testing of astronomy software. *Astronomy and Computing* 20 (July 2017), 105–119. https://doi.org/10.1016/j.ascom.2017.06.001

[18] Manoj Kumar Patra, Anisha Kumari, Bibhudatta Sahoo, and Ashok Kumar Turuk. 2022. Docker security: Threat model and best practices to secure a docker container. In *2022 IEEE 2nd International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC).* IEEE, Gunupur, Odisha, India. https://doi.org/10.1109/iSSSC56467.2022.10051481

[19] Devi Priya, Sibi Chakkaravarthy Sethuraman, and Muhammad Khurram Khan. 2023. Container security: Precaution levels, mitigation strategies, and research perspectives. *Computers and Security* 135 (Dec 2023), 103490. https://doi.org/10.1016/j.cose.2023.103490

[20] Rajyashree, Senthilkumar Mathi, Saravanan, and Sakthivel. 2024. An Empirical Investigation of Docker Sockets for Privilege Escalation and Defensive Strategies. *Procedia Computer Science* 233 (2024), 660–669. https://doi.org/10.1016/j.procs.2024.03.255 5th International Conference on Innovative Data Communication Technologies and Application (ICIDCA 2024).

[21] Amarjeet Singh and Alok Aggarwal. 2022. A Comparative Analysis of Veracode Snyk and Checkmarx for Identifying and Mitigating Security Vulnerabilities in Microservice AWS and Azure Platforms. *Asian Journal of Multidisciplinary Research and Review* 3, 2 (Mar 2022), 232–244. https://ajmrr.org/journal/article/view/5

[22] Snyk. 2024. *Risk Score.* https://docs.snyk.io/manage-risk/prioritize-issues-for-fixing/risk-score Accessed: 2024-06-29.

[23] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. 2019. Container security: Issues, challenges, and the road ahead. *IEEE Access* 7 (2019), 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911737

[24] Xili Wan, Xinjie Guan, Tianjing Wang, Guangwei Bai, and Baek-Yong Choi. 2018. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications* 119 (Oct 2018),

97–109. https://doi.org/10.1016/j.jnca.2018.06.007

[25] Renfang Wang, Hong Qiu, Xu Cheng, and Xiufeng Liu. 2023. Anomaly detection with a container-based stream processing framework for Industrial Internet of Things. *Journal of Industrial Information Integration* 35 (Oct 2023), 100507. https://doi.org/10.1016/j.jii.2023.100507

[26] Ann Yi Wong, Eyasu Getahun Chekole, Martín Ochoa, and Jianying Zhou. 2023. On the security of containers: Threat modeling, attack analysis, and mitigation strategies. *Computers and Security* 128 (May 2023), 103140. https://doi.org/10.1016/j.cose.2023.103140

[27] Qingyang Zeng, Mohammad Kavousi, Yinhong Luo, Ling Jin, and Yan Chen. 2023. Full-stack vulnerability analysis of the cloud-native platform. *Computers and Security* 129 (June 2023), 103173. https://doi.org/10.1016/j.cose.2023.103173

[28] Ahmed Zerouali, Tom Mens, and Coen De Roover. 2021. On the usage of JavaScript, Python and Ruby packages in Docker Hub images. *Science of Computer Programming* 207 (2021), 102653. https://doi.org/10.1016/j.scico.2021.102653

[29] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K Paul, Keren Chen, and Ali R Butt. 2021. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (April 2021), 918–930. https://doi.org/10.1109/TPDS.2020.3030582

[30] Hui Zhu, Christian Gehrmann, and Paula Roth. 2023. Access Security Policy Generation for Containers as a Cloud Service. *SN Computer Science* 4, 6 (2023). https://doi.org/10.1007/s42979-023-02186-1

## A APPENDIX A

During the preparation of this work the author utilized ChatGPT 4 for assistance in debugging, providing Dockerfile examples and generating references in the BibTeX format which were further served Overleaf for the creation of the reference list. In addition, the author used Grammarly to check the grammar and spelling, as well as QuillBot for some paraphrasing suggestions. After using these tools and services, all content was thoroughly reviewed and edited as needed. The author takes full responsibility for the final outcome.