



UPPSALA
UNIVERSITET

IT kDV 24 040

Degree project 15 hp

June 2024

Security and Application Deployment using Docker

Nathalie Borglund

Bachelor's Programme in Computer Science



UPPSALA
UNIVERSITET

Abstract

In the dynamic world of software development, containerization has emerged as a pivotal technology, offering flexibility and efficiency in deploying applications. Among the numerous tools available for container management, Docker provides a comprehensive solution for building, shipping, and running distributed applications. However, its widespread use brings critical security considerations. Ensuring the security of Docker images and containers is essential to protect against vulnerabilities. Docker environments can be vulnerable to various attacks, such as malicious interference, data breaches, resource monopolization, command injections, and other external threats. These vulnerabilities are concerning because they can lead to compromised data integrity, unauthorized data manipulation, and overall system instability. Due to Docker's growing popularity, Mirakelbolaget seeks to investigate how Docker can be implemented securely. This study focuses on securing Node.js applications, which pose significant security risks due to their extensive dependencies on open-source modules. Isolating these applications in Docker containers provides a controlled environment, simplifying management and deployment. Base images are the foundation of your program and can introduce vulnerabilities. This study transitions from the 'node:14' base image to 'node:20.12.1-bookworm-slim' showing a significant reduction in vulnerabilities and improved efficiency. By utilizing vulnerability scanning tools such as Snyk and Docker Scout, the study was able to identify and mitigate these vulnerabilities effectively. The risk of unauthorized access and data breaches can be mitigated by applying the principle of least privilege. This can be achieved through non-root user configurations, which on the WSL 2 Linux Subsystem did not require further adjustments. Additionally, implementing read-only volumes and carefully configuring volume mount points to the host is necessary to ensure robust security. Preventing a single container from monopolizing the entire system's resources is essential to maintaining overall system performance and stability. Effective resource, which was achieved through controlled CPU and memory usage. To protect against command injection attacks or access through external communication, improved internal and external access configurations were implemented. These configurations isolated containers and managed external access securely. This was achieved through the application of the principle of least privilege, which mitigated command injection attacks. Additionally, the use of a reverse proxy container proved effective in enhancing security for external communication. Overall, this investigation offers guidelines for Mirakelbolaget to build secure, efficient Docker environments, addressing how Docker can enhance security and simplify the deployment of Node.js applications in a business context.

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisor: Michal Borglund Subject reader: Yuan Yao

Examiner: Johannes Borgström

Table of Contents

1	Introduction	2
1.1	Problem Description	3
2	Background	4
2.1	Container Technology	4
2.1.1	Container vs Virtual machines	4
2.2	Docker Overview	4
2.2.1	Docker Images	5
2.2.2	Dockerfiles	6
2.2.3	Docker Containers	7
2.2.4	Docker Network	7
2.2.5	Docker Container on an Ubuntu 15.04 image	8
2.3	Data Management and Persistence in Docker	9
2.3.1	Docker Compose	10
2.3.2	Docker Swarm	10
2.3.3	Kubernetes and Docker	10
3	Security Considerations	11
3.1	Docker Images	11
3.2	Docker Containers	11
3.3	Operating System Specific Security	12
3.4	Docker Networking	12
3.5	Volume Management	12
3.6	General Security Practices	13
4	Related work	14
5	Implementation	16
5.1	Vulnerability Scanning	16
5.1.1	Snyk	16
5.1.2	Docker Scout	16
5.2	Privileges	16
5.2.1	Non-root User in Windows	16
5.2.2	Read-only Volumes	17
5.2.3	Volumes For Data Persistence	17
5.3	Resource Management	18
5.3.1	Simulating CPU Stress	18
5.3.2	Simulating Memory Stress	18
5.4	Network Security	19
5.4.1	Injection Vulnerability Test	19
5.4.2	Node.js Application Configuration and Network Testing	21
5.4.3	Docker Network Configuration direct exposing Node.js Container	22
5.4.4	Docker Network Configuration with reverse proxy	22

6	Results and Discussion	24
6.1	Vulnerability Scanning	24
6.1.1	Snyk	24
6.1.2	Docker Scout	26
6.1.3	Evaluating Docker Scout and Snyk	30
6.2	Privileges	30
6.2.1	Non-root User in Windows	30
6.2.2	Read-only Volumes	30
6.2.3	Volumes For Data Persistence	31
6.3	Resource Management	31
6.3.1	Simulating CPU Stress	31
6.3.2	Simulating Memory Stress	32
6.4	Network Security	32
6.4.1	Injection Vulnerability Test	32
6.4.2	Docker Network Configuration direct exposing Node.js Container	33
6.4.3	Docker Network Configuration with reverse proxy	33
7	Conclusion	35

1 Introduction

In the dynamic world of software development, containerization has emerged as a pivotal technology, offering flexibility and efficiency in deploying applications. Docker encapsulates applications within containers, ensuring consistent operation across different environments. However, this modern deployment strategy introduces significant security challenges. While beneficial for many reasons, the isolation provided by Docker containers also introduces security concerns. Docker containers often run with high privileges, together with several stacked image layers containing potential vulnerabilities. These vulnerabilities could compromise the entire container ecosystem without proper management, leading to severe security hazards.

Mirakelbolaget has recognized these challenges and has commissioned this thesis to investigate the security of Docker environments. The purpose of this study is not only to enhance the company's understanding of Docker security but also to serve as a guideline for developers at Mirakelbolaget, aiding them in building and maintaining secure, efficient, and resilient Docker environments.

The thesis is structured to address these various security concerns, divided into several key sections, each focusing on different aspects of Docker security:

1. **Vulnerability Scanning (Section 5.1):** This section evaluates Docker and Snyk, highlighting their role in real-time vulnerability scanning and the importance of integrating these tools into the CI/CD process. A comparative analysis helped to identify the unique strengths and weaknesses of each tool and explore the benefits of employing both to enhance security coverage.
2. **Privileges Management (Section 5.2):** Discusses the implications of running Docker containers with non-root user configurations, the effectiveness of read-only volumes, and the persistence of data across container lifecycles. This section underscores the critical part of configuring privileges and data access to prevent unauthorized actions within containers. This includes using non-root users in WSL 2 OS, read-only volumes, and ensuring persistent data only be used when necessary.
3. **Resource Management (Section 5.3):** Examines Docker's capability in managing system resources to prevent resource exhaustion. Simulated stress tests illustrate how Docker enforces resource limits, ensuring that containers do not exceed their allocated resources while maintaining system stability and preventing potential denial-of-service attacks. This includes managing CPU and memory usage effectively.
4. **Network Security (Section 5.4):** Focuses on Docker's network configurations, exploring how internal and external network access policies can safeguard containers from unauthorized access. The role of network isolation and the use of reverse proxies are discussed to demonstrate how these strategies protect containerized applications from network-based threats.

By offering detailed insights into Docker image management and security practices, this thesis aims to help developers at Mirakelbolaget build and maintain secure, efficient, and resilient Docker environments. Through practical implementation and testing, particularly with Node.js and MongoDB on the WSL 2 operating system, this study provides comprehensive guidelines and solutions for enhancing the security and efficiency of Docker deployments in a business context.

Although Docker primarily operates on Linux, it is also compatible with Windows and macOS. Notably, Docker Desktop is not supported on Windows server versions 2019 and 2022. However, alternatives such as Docker on WSL 2 or Hyper-V can be utilized in such scenarios. One of Docker's strengths is its ability to quickly set up complex solutions[1].

1.1 Problem Description

While Docker offers significant benefits in scalability, portability, and efficiency, it also introduces new security challenges that must be addressed to maintain a safe ecosystem of containerized applications. This thesis focuses on addressing several key security issues related to Docker, particularly for Node.js applications:

- How does Docker's containerization technology impact the security posture of web applications, particularly those developed with Node.js?
- What are the most frequent security vulnerabilities associated with Docker containers, and how can they be systematically identified and mitigated?
- What are the best practices for configuring Docker networks to secure internal and external access, preventing unauthorized access, and mitigating risks such as command injection?

By addressing these questions, this thesis provides valuable insights and practical solutions for enhancing the security of Dockerized Node.js applications, contributing to best practices for maintaining a secure and reliable Docker environment.

2 Background

2.1 Container Technology

Container technology is a method of virtualization that allows applications to run on isolated user spaces called containers. It encapsulates the application's code, runtime, system tools, libraries, and settings. Containers share the machine's kernel and do not require an operating system per application, making them more portable and efficient than traditional virtual machines (VMs). Platforms that offer container technology have become popular in the industry for their ability to simplify complex technologies and processes into more manageable and deployable units. Containers have their own file systems, networking, and isolated process space, which make them suitable for deploying microservices and ensuring that software will behave the same way regardless of where it is deployed[2].

2.1.1 Container vs Virtual machines

In contrast to container technology, virtual machines (VMs) offer a different approach to virtualization. VMs work by simulating the hardware of a computer, with each VM running its version of an operating system, managed by a hypervisor. The main advantage of VMs is their ability to run different operating systems on the same physical hardware. However, they tend to use more resources because each VM requires its full system setup. Containers virtualize at the operating system level and enable multiple containerized applications to run in isolated spaces while sharing the same OS kernel. This approach offers benefits such as reduced size and performance efficiency due to less overhead than traditional VMs[3].

VMs abstract hardware to create multiple simulated environments with separate OS via a hypervisor. Containers virtualize the OS and enable multiple containerized applications to run in isolated spaces while sharing the same kernel[3].

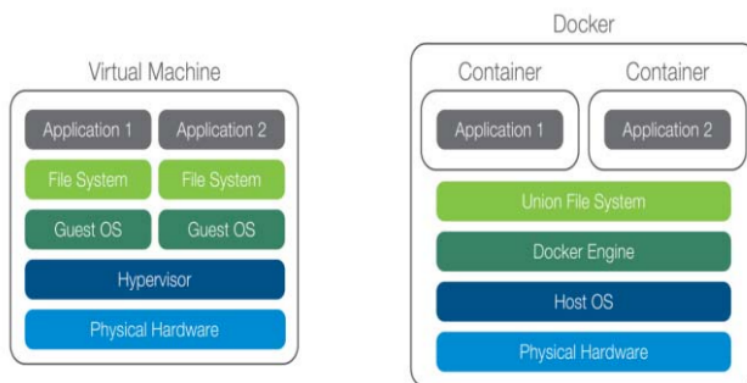


Figure 1: Virtual Machine vs Container [4]

2.2 Docker Overview

Docker is a software development tool that is an open-source platform for developing, transporting, and running applications. It simplifies the process of building and running applications by employing containerization technology. This technology allows developers to package their applications and dependencies into portable containers that can be deployed and executed on any system with Docker installed, making them independent of the underlying infrastructure. At the

core of Docker's functionality is the Docker Engine, a powerful layer that enables the creation and running of containers.

The architecture of Docker is based on a client-server model where the Docker client communicates with the Docker daemon. This communication can occur within the same system, or the Docker client can connect to a remote Docker daemon. Management of Docker containers is facilitated through the Docker API or the command-line interface (CLI)[5].

- **Docker Host:** The environment on which the Docker and Docker daemon runs.
- **Docker Daemon (dockerd):** A background service that listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The daemon can also collaborate with other daemons to manage Docker services across multiple machines.
- **Docker Client (docker):** Most Docker interactions are facilitated through the Docker client. It's through this interface that commands such as 'docker run' are passed to the Docker daemon. The client employs the Docker API for container management. A client is not limited to interacting with a single daemon.
- **Docker Desktop:** An interface through which users interact with Docker, executing commands like 'docker run' to signal the daemon's operations.
- **Docker Registries:** Repositories for storing Docker images, on Docker Hub.

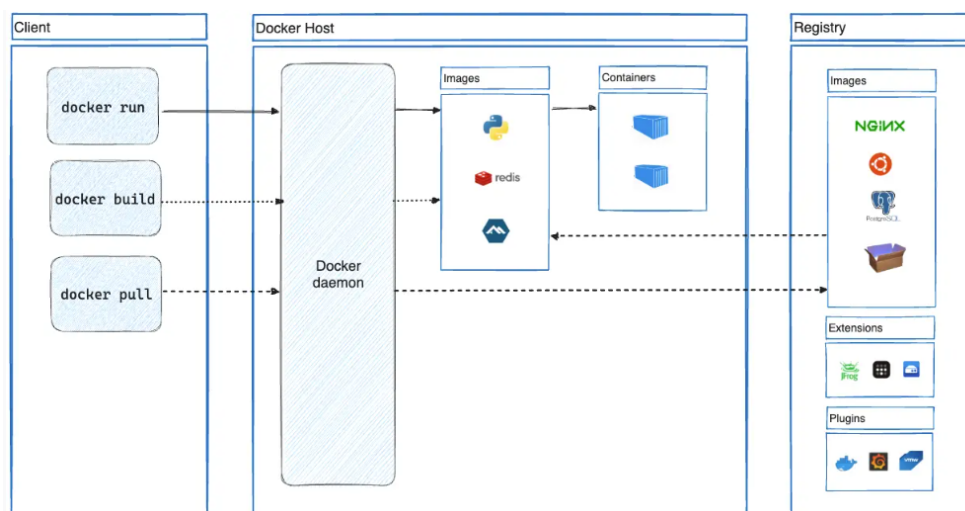


Figure 2: Docker Architecture [5]

2.2.1 Docker Images

Docker images are static templates made of read-only layers used for creating containers. Each layer in an image represents a set of changes from the layer before it, making each image immutable. This immutability is achieved through the use of a Dockerfile, which is a plain text file containing commands that a user can execute to assemble an image. Commands like 'RUN', 'COPY', 'ADD' add a new layer for each instruction that is executed. Each adjustment in the Dockerfile creates a new layer that stacks on top of the previous one. If these layers are reused in future builds, Docker can skip redoing these steps, making the build process faster thanks to a feature called caching[6].

Docker images are distributed through repositories called registries, with Docker Hub being the most well-known. Docker's official images, hosted on Docker Hub include a range of secure repositories containing operating systems, programming languages, and tools like Ubuntu, Python, Node.js, and MongoDB. These images are among the most secure on Docker Hub and are designed to minimize exposure to common vulnerabilities[7].

The "Supported tags and respective Dockerfile links" section in the documentation for Docker Official Images is a valuable resource that lists all the available versions or configurations of a Docker image, where each image is tagged with a specific label. These tags help users to identify and select precise versions of an image that fits their specific needs. Each tag is linked to the Dockerfile that was used to create that specific image, this allows users to review the exact build instructions and components included in each image[8].

It is important to note that the 'latest' tag may not always be the most convenient for your specific use case. This tag can sometimes include more than needed, such as tools like Git. Using 'latest' might not be a good practice since it can lead to unwanted compatibility issues, for example, if the Node.js version changes it could cause the application to break. In cases where image sizes and resource efficiency are important smaller images can be more beneficial. Docker Official Images offers 'slim' or 'alpine' versions of standard images, that provide just the essentials for the needed application[8].

For instance, the Node Docker Official Image built on a Debian base, offers different image variants for different deployment needs. The standard Debian tags like 'node:19-bullseye' provide a stable environment, while slim tags such as 'node:19-bullseye-slim' reduce the image size by reducing unnecessary packages. The Alpine tags for example 'node:19-alpine' offer the smallest footprint and are most suitable for environments where minimal resource usage is critical[9].

Similarly, the MongoDB Docker Official Image creates an environment for deploying MongoDB databases, using base images like Ubuntu. It provides variants like 'mongo:5.0-ubuntu' for reliable setups, 'mongo:5.0-slim' for reduced image size, and 'mongo:5.0-windowsservercore' for integration with Windows-based systems. These options ensure that developers can select the most appropriate Docker image based on their specific needs[10].

2.2.2 Dockerfiles

A Dockerfile is a template for building Docker images. It is a plain text file containing commands and instructions that tell Docker how to assemble an image. Docker images are built automatically by reading these instructions from a Dockerfile through the 'docker build' command[11].

The Dockerfile begins with a 'FROM' instruction which, specifies the base image from which you are building. This base image is the starting point from where you build your image. The 'FROM' instruction can appear multiple times within a single Dockerfile to support multi-stage builds and allows optimizing Docker images by reducing the size and securing contents. A Dockerfile typically should conclude with either a 'CMD' or an 'ENTRYPOINT' instruction. These instructions define what executable is run and how it is run when a Docker container is started from the image.

'CMD' instruction is used to provide default commands and parameters that will be executed when the container runs. Essentially, the 'CMD' tells the Docker container what command it should execute by default when it starts up. When an 'ENTRYPOINT' instruction is used, it will always be executed when the container starts up. The 'CMD' can then be used to provide default arguments to the 'ENTRYPOINT'. If both 'ENTRYPOINT' and 'CMD' are used together,

'ENTRYPOINT' specifies the program to run and 'CMD' specifies the arguments that should be passed to that program. The arguments in 'CMD' can be overridden by providing new command-line arguments when the container is started. Without one of these instructions, the container will not perform any action automatically when starting. To clearly define what the built image actually should do upon startup is it preferable to use either 'ENTRYPOINT' or 'CMD'. [11]

Docker utilizes a layered architecture. Each instruction in a Dockerfile adds a new layer to the image, with Docker caching these layers. This means that if your files get changed, Docker only rebuilds the layers that have changed. [11]

Dockerfiles support multi-stage build, which helps create smaller and more efficient images. With this method, you can first use a base image that includes all the tools needed to build your application. Once the application is built, it can be transferred to a new, clean image, removing unnecessary components such as the building tools needed in the earlier stage. This final image contains only your application and its direct dependencies, without the building tools. [11]

2.2.3 Docker Containers

A Docker container is a lightweight, portable package that includes everything needed to run a software application, such as code, runtime, system tools, libraries, and settings. This packaging ensures that containers are isolated from each other and the host system through the use of namespaces. Namespaces provides a layer of isolation in Docker that makes each container appear as if it has its independent environment, including its own network interfaces, process IDs, and file systems[5].

Containers are created from Docker images, which are blueprints of the library and dependencies that are needed for the application. When a container runs, it adds a thin writable layer on top of this image. This writable layer captures all filesystem changes during a container's execution, ensuring that the underlying image remains unchanged. By executing a container, Docker adds a new writable layer that allows the application inside the container to function [12].

Docker simplifies container management through its command-line interface (CLI) or via API calls, which enables users to easily create, start, stop, move, or delete containers. The portability of containers also means they can be seamlessly moved across different Docker hosts[5]. Docker's approach to packaging and running applications in isolated containers allows multiple containers to operate simultaneously on a single host. This isolation, strengthened by namespaces and security features, makes containers a tool for distributing applications[13].

2.2.4 Docker Network

Docker networks enable containers to communicate with each other and with non-Docker workloads. By default, containers are provided with networking capabilities and can establish outgoing connections. However, a container is only aware of its network interface, which is provided with an IP address, a gateway, a routing table, a DNS service, and other necessary networking details. If the 'none' network driver is used, the container is completely isolated, meaning it cannot detect or identify whether other containers are part of Docker workloads or not. [14]

Networks Drivers in Docker:

- **Bridge:** The default network driver.
- **Host:** Removes network isolation between the container and the Docker host, sharing the host's networking namespace.
- **None:** Provides complete isolation by disabling all networking.

- **Overlay:** Connects multiple Docker daemons and enables swarm services to communicate with each other.
- **Ipvlan and Macvlan:** Allows you to assign an IP address or MAC address to a container, making it appear as a physical device on your network.

Containers can also be connected directly to another container's network using the `--network container:<name|id>` option.

Containers do not expose their ports by default. To make them accessible from outside the Docker host ports can be published using `--publish` or `-p` flags to map container ports to host ports.

`-p 8090:90` maps port 8090 on the host to port 90 on the container. `-p 127.0.0.1:8090:90` allows only calls from localhost.

Containers receive an IP address from the subnet of the network they are connected to. The default container hostname is its ID but can be set to a custom hostname using the `--hostname` flag. [14]

Containers typically use the same DNS servers as the host by default. It is possible to specify different DNS servers using the `--dns` flag when creating or running a container. When a container is connected to a user-defined network, it uses Docker's integrated DNS server, which sends requests to find internet addresses to the DNS server on the host computer. If containers are on the same user-defined network, they still need their ports to communicate with each other. However, they do not need their ports published to the host and can directly communicate using the internal IP addresses or container names. [14]

2.2.5 Docker Container on an Ubuntu 15.04 image

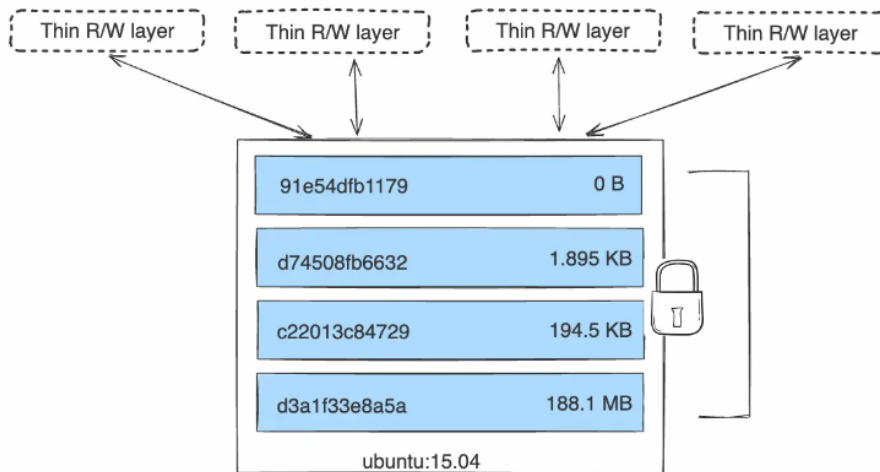


Figure 3: Docker Container based on an Ubuntu 15.04 image [12]

Figure 3 illustrates the layered architecture of a Docker container based on an Ubuntu 15.04 image.

'd3a1f33e8a5a' is the base layer that represents the initial Ubuntu image. 'c22013c84729', 'd74508fb6632' and '91e54dfb1179' are layers that were created by instructions such as 'RUN', 'COPY', or 'ADD' in the Dockerfile.

Above the 'Thin Read/Write (R/W) Layers is a thin writable layer unique to the container. This is where all the changes made to the file system during the container's runtime are stored. Modifying existing files or adding new ones are captured in this layer.

If the container is deleted, any changes stored in its layer will be lost. To preserve the data volumes are needed.

On the right side of the layers, there is a reference to the container's size on disk. The virtual size of a container is represented by its 'Thin Read/Write (R/W) Layer and the image total size on disk.

However, since containers can share images the actual size might be smaller.

Multiple containers can share the same read-only layers, making the actual size smaller and more storage efficient than virtual machines.

The Docker command '`docker ps -s`' allows you to see the storage usage of each container. This includes not just the thin R/W layer but also any additional storage like logs, volumes, and checkpoints. [12]

2.3 Data Management and Persistence in Docker

Docker provides flexibility in how data is stored and managed, with each type of mount offering a different use case. Figure 5 illustrates the different data storage and mounting options for handling data within containers.

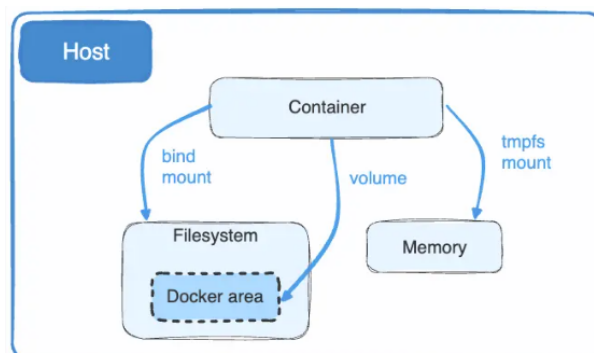


Figure 4: Docker's data storage options [15]

- **Host:** Represents the physical or virtual machine Docker is running on. This could be the local machine, cloud-based server, or any environment that supports Docker.
- **Bind Mount:** The type of mount allows you to map a file or directory from the host to a container. It enables containers to access and modify files on the host file system.
- **Volume:** Docker managed mount that is used to persist data generated by and used by Docker containers. It is stored within the host filesystem but is managed in isolation from the host's core operations. Unlike bind mounts, they are not dependent on the host's filesystem structure.
- **tmpfs mount:** In-memory storage option. Useful for temporary data that does not need to be persisted across container restarts.

All files created inside a container are stored on a writable container layer by default, this layer does not persist after the container is destroyed. Docker employs storage drivers to manage the

container's filesystem, which is composed of multiple layers within an image, and these layers can be shared across various containers.

These drivers use a union filesystem approach that is facilitated by the Linux kernel, which can introduce performance overhead compared to direct data volume writes. [15]

2.3.1 Docker Compose

Docker Compose is a tool used to define and manage multi-container Docker applications. With Compose it is possible to create a YAML file to configure the application's services, networks, and volumes, simplifying the process of arranging services. [16] Docker Compose is possible to use in both development, production, and testing environments. [17] Docker Compose optimizes the development process by changing the configurations used to create a container. When a service is restarted without any changes, Compose can efficiently reuse the existing containers, thereby speeding up the building process. The application lifecycle can easily be managed through commands like `'docker compose up -d'` to start all services, and `'docker compose down'` to stop them. Additionally, `docker compose logs -f` can be used to show logs and service status. [16]

2.3.2 Docker Swarm

Docker Swarm is a container orchestration tool, used for managing clusters of Docker containers. It transforms a group of Docker engines into a single, virtual Docker engine capable of deploying scalable applications across multiple hosts. Docker Swarm integrates with Docker and allows the use of the same Docker CLI for arrangement. The transition from Docker Compose to Docker Swarm is not mandatory when moving development to production. For simpler and smaller-scale applications Docker Compose is more sufficient, but for larger and more complex deployments requiring high availability, scalability, and automatic load balancing Docker Swarm is more beneficial. [18]

2.3.3 Kubernetes and Docker

Kubernetes is another popular choice for orchestration, for managing large and complex applications across multiple containers and hosts. Kubernetes and Docker complement each other, while Docker provides a container runtime environment, Kubernetes offers advanced orchestration capabilities. Kubernetes simplifies container operations with features like automatic bin packing, automatically replacing and rescheduling containers from failed nodes, service discovery, and load balancing. [19]

3 Security Considerations

Deploying applications using Docker securely is crucial. This section outlines essential security considerations based on findings and experience in deploying and managing Docker environments. These include the security of Docker images, containers, networking, and the Docker daemon, with a particular focus on deploying Node.js and MongoDB applications. Additionally, this section discusses security recommendations for running Docker on various operating systems, highlighting the importance of understanding how different hosting operating systems handle security.

3.1 Docker Images

Deployment of a secure Docker environment begins with the choice of base images. Using trusted, official, or verified Docker images is important, as these are more likely to be free from malware and vulnerabilities. Docker Hub hosts over 8.3 million repositories, with official images and verified publishers. [20] These images are regularly updated and security checked and provide a reliable base for building Docker applications. Moreover, using multi-stage builds minimizes image sizes and reduces the potential for security attacks. Multi-stage builds involve constructing Docker images in stages, where each stage can start from a clean base and only include the necessary components from previous layers. This process makes the image smaller and safer as it excludes unnecessary dependencies that could cause security risks. An example of unnecessary dependencies could be build tools and temporary data necessary to create the application not needed in the final application.

It is also important to regularly update Docker images and use automated tools for vulnerability scanning to ensure that the most suitable image is selected. This helps ensure that the chosen base image is as secure as possible.

Additionally, if an image is updated in the Dockerfile is it important to use the `'--no-cache'` option, to build the image with the new instructions otherwise a cached image version will be created.

3.2 Docker Containers

When setting up Docker containers is it crucial to adopt several security considerations while following the principle of least privilege ensuring that each container of the system only operates with the necessary permissions.

Running containers as non-root is an important practice that minimizes the risk of privileged attacks. Despite WSL2 providing a high level of isolation between the Windows host and Linux containers, it is still essential to ensure that containers do not operate with root privileges.

To further apply security, configuring containers with read-only filesystems can prevent unauthorized data manipulation and ensure that files within the container cannot be changed. Treating containers as immutable by replacing them instead of updating them can also minimize the risk of runtime attacks.

Implementing resource limits on containers can mitigate problems. By setting limits on each container, only the affected container is impacted and not the entire system. This limitation could help prevent the exhaustion of system resources and maintain general system availability. Setting precise CPU and memory limits for the containers would prevent them from using more resources than necessary and help in maintaining system stability.

Additionally, minimizing the number of packages installed in containers most probably reduces the probability of attacks by making it harder to exploit unused or unnecessary software.

3.3 Operating System Specific Security

Operating systems differ in their approach deploying Docker environments. Linux uses namespaces and control groups to isolate containers and manage resources. This setup limits containers running as root within their containers, reducing the risk of privilege escalation attacks.

In contrast, Windows does not support Linux-style namespaces and has a different approach, as Windows shares the host's kernel. However, Windows provides Hyper-V containers, which isolate containers at the hypervisor level by running each container in a lightweight virtual machine.

Understanding the differences between operating systems is crucial for deploying Docker securely. For instance, when using WSL 2 which is a Linux subsystem there is no need for namespaces because it runs a full Linux kernel in a lightweight virtual machine. This setup isolates the Linux environment from Windows and provides a security layer that reduces the need for using namespace configurations.

3.4 Docker Networking

Isolating containers from one another is a key to network security. This is done by, implementing network policies that restrict traffic between containers using the principle of least privilege at the network level. These policies can limit container-to-container communication both within the same host and across other hosts.

By default, Docker Compose establishes a single network for the containers, allowing them to communicate. For instance, using the 'expose' directive instead of 'ports' in Docker Compose is a recommended security practice when a container needs to expose its ports only to other containers in the same network.

Careful management of ports is essential to ensure that only necessary ports are exposed, thereby reducing the vulnerabilities. When deploying applications like a Node.js application or databases such as MongoDB in Docker containers it is important to only expose the necessary ports and firewall rules to further limit access.

For secure communication is encrypting data transmitted between services using TLS or SSL a good security practice, especially for any exposed APIs or web interfaces. Whenever possible is it a good idea to run services in separate networks. When services are placed on separate networks it creates an isolation limiting unauthorized access.

3.5 Volume Management

Securing volumes in Docker is essential if they contain sensitive data, that if exposed could lead to security breaches and data loss. A best practice for achieving a secure volume setup in Docker is to utilize Docker managed volumes instead of bind mounts. This strengthens isolation from the host system and enables Docker to manage data access more securely. For applications that do not require write access, it is recommended for volumes to be mounted as read-only to prevent unauthorized modifications.

Furthermore, is it important to restrict access to the directory on the host where the Docker volumes are stored, to ensure that only authorized users have access. Implementing encryption with volumes is also recommended for securing sensitive information. Regularly monitoring volumes for any unauthorized access or changes is crucial for maintaining a secure volume configuration. Additionally, is it a good practice to regularly back up data to handle recovery in cases of data corruption or data loss.

3.6 General Security Practices

- **.dockerignore File:** Use a .dockerignore file to exclude sensitive information from Docker images during the build process. This file will allow you to specify files and directories to be excluded from the image.
- **Docker Daemon Security:** Ensure that the Docker daemon is not publicly accessible and implement TLS authentication to prevent unauthorized access. For instance, activate TLS on a MongoDB database running in a Docker container.
- **Dockerfile:** Ensure that Dockerfiles are securely written and managed to prevent security vulnerabilities. This involves using trusted base images.
- **Secrets Management:** Use Docker Secrets or other external tools to handle sensitive data such as passwords and API keys.
- **Minimal Access Rights:** Run containers with the minimal set of permissions necessary.
- **Security Updates:** Regularly update host system and Docker components to protect against known vulnerabilities.
- **Continuous Integration/Continuous Deployment (CI/CD) Pipeline:** Maintain a secure CI/CD pipeline where all components in the environment are verified to ensure security in the deployment process with scanning tools.

4 Related work

In the area of virtualization technology, Docker has become a popular tool that changes how applications are deployed and managed. But as it has become more widely used, Docker's vulnerabilities, particularly in the context of containerized environments and web applications have become a concern.

The research "Docker Security: A Threat Model, Attack Taxonomy and Real-Time Attack Scenario of DoS" delves into these vulnerabilities presents a threat model, and demonstrates a Denial of Service (DoS) attack in Docker settings. Docker's containerization offers a lot of advantages, including improved scalability, speed, and resource efficiency making it more scalable compared to traditional virtual machines. However, its architecture allows containers to directly interact with the host kernel, which introduces security vulnerabilities. And unlike VMs, is Docker more directly vulnerable to attack. [21]

The researchers propose a layered threat model for Docker, that illustrates the different potential security attacks within a Docker environment.

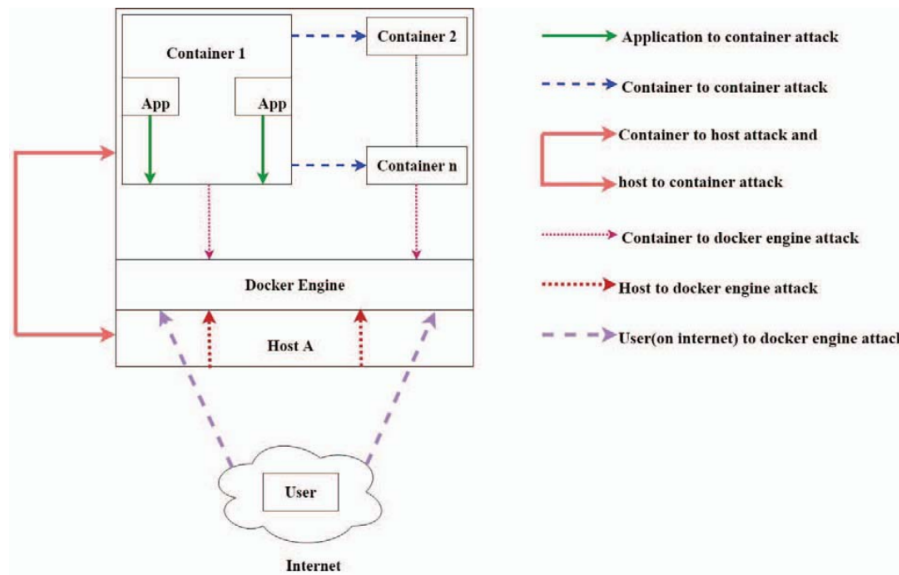


Figure 5: Layered Threat Model [21]

- **Application to container attack:** This occurs when a program running inside a container manages to take control over the container itself.
- **Container to container attack:** If one container gets infected, it could potentially infect other containers if they are not properly isolated.
- **Container to host attack and host to container attack:** If a container affects the host, it can have a big impact due to the shared kernel resources. The container can gain privileges or access sensitive parts of the host.
- **Container to Docker engine attack:** If a container interferes with the Docker engine, it could lead to unauthorized modifications.
- **Host to Docker engine attack:** Host systems have Docker engine installed and have privileges. Misuse of this access can harm the Docker engine, potentially affecting all containers managed by the engine.

- **User (on the internet) to Docker engine attack:** If an external attacker through the internet targets the Docker engine.

The study then continues to categorize potential Docker attacks across different layers including, container, application, Docker engine, and the host. [21]

At the container, potential threats include malware, which involves injecting harmful software, and Denial of Service (DoS) attacks that overwhelm system resources. Here can also, privilege escalation occur, where unauthorized access to higher privilege levels within the container is gained. Escape attacks, involve breaking out of the container to access the host system. And network threats, such as ARP spoofing, MAC flooding, and Man-in-the-middle attacks.

At the application level within Docker, potential threats include DoS attacks targeting specific applications and malware-infected applications. Poisoned images, which could compromise container images containing vulnerabilities or malicious code, including outdated software with known vulnerabilities are also a potential threat[21].

The Docker engine faces potential attacks by using the Cloud Container Attack Tool (CCAT) to exploit cloud environments. Code injection, by inserting harmful code that could affect Docker's operations, while kernel exploit could take advantage of vulnerabilities in the shared operating system kernel[21].

At the host level, potential threats include escape attacks, where an attacker breaks out from Docker to the host system, and DoS attacks targeting the host's resources directly. And tampering, involving unauthorized modifications to Docker's configurations or files. Unwanted service attack exploits, because of unnecessary services running on the host, and kernel exploits involve taking control of the host through vulnerabilities in the shared kernel[21].

The article also includes a real-world example where a Denial of Service (DoS) attack is carried out on a Docker system. It uses network scanning tools like 'nmap' to find targets and 'hping3' to overwhelm the system with traffic. The attack led to a noticeable increase in the workload of the CPU and demonstrated how such attacks could load and potentially overwhelm system resources. [21]

5 Implementation

This chapter outlines tools and methodologies employed to demonstrate and analyze potential security risks associated with different parts of the Docker architecture. By utilizing simplified test setups, this approach was aimed to illustrate Docker's responses to both potential and more straightforward security vulnerabilities. The environment setups were designed to evaluate how Docker manages sensitive data, responds to unauthorized access attempts, and upholds strong network isolation.

5.1 Vulnerability Scanning

Ensuring the security of base images is crucial because vulnerabilities within these images could threaten the entire Docker environment. Different base images were explored to investigate the vulnerabilities. Scanning tools used for this purpose were Snyk and Docker Scout, employed to compare the results from two different providers, analyzing their simplicity of usage and effectiveness in scanning vulnerabilities.

For demonstration purposes, a 'node_container' in the Docker environment was set up using the base image 'node:14'.

```
#Code snippet of Dockerfile for 'node_container'  
...  
FROM node:14  
...
```

5.1.1 Snyk

Snyk was integrated with the GitHub repository containing the Docker setup to enable automatic scanning of Dockerfiles, including the associated Docker images each time changes were pushed to the repository. This setup enabled continuous scanning of the Docker images, ensuring that changes were consistently monitored for security vulnerabilities.

5.1.2 Docker Scout

Docker Scout included in Docker Desktop was another tool used for scanning Docker images. Docker Scout is enabled by default in Docker Desktop settings and scanned images as they were created and provided real-time updates on security issues related to the base images.

5.2 Privileges

The Principle of Least Privilege is an important security strategy that ensures users and programs only have minimum access needed to perform their tasks. In Docker environments, applying this principle helps reduce the risks of security issues like unauthorized access and data breaches. This section will explain how to use the Principle of Least Privilege with Docker containers to improve security.

To apply the Principle of Least Privilege in Docker, several strategies were used to ensure that containers only have the necessary permissions and resources they require to function.

5.2.1 Non-root User in Windows

A test was conducted to investigate whether Docker containers in a Windows environment would follow the Principle of Least Privilege without specific restrictive settings to look into the default access rights granted to the containers.

To demonstrate this, a Docker container was linked to the host environment through a volume mount. The following is a code snippet from a `docker-compose.yml` file.

```
#Docker Compose for a container
...
volumes:
  - 'C:/Windows:/windows'
...
```

Next, a folder named 'Test' was created on the host with system standard root credentials under the 'c:\Windows' folder. The container containing the mounted volume was then, accessed using the following command, 'docker exec <container-id> -it /bin/bash', to access the volume from a terminal within the container. The 'Test' folder was the target of the test to see if it could be tampered with, such as by renaming or deleting it.

5.2.2 Read-only Volumes

To further apply the Principle of Least Privilege read-only volumes can be used. Specifying a volume as read-only ensures that it cannot be modified after deployment. When a volume is mounted with the ':ro' in a Docker Compose file or a Docker run command, it sets the volume to read-only. This means that the data within this volume cannot be modified by processes running inside the container. The following code snippet is a volume configuration of the Node.js container in Docker Compose.

```
#Docker Compose for Node.js application
...
volumes:
  - node-data:/data/config:ro
...
```

Tests were conducted to verify the functionality of read-only volumes to ensure that data within the specified volume could not be modified by processes running inside the container. Once the environment was set up, access to the Node.js container was done using the 'docker exec -it <node_container-id> /bin/bash' command. Within the container, attempts were made to '/data/db' to create, modify, or delete files within the read-only volume.

5.2.3 Volumes For Data Persistence

Managing data efficiently within Docker involves defining volumes that allow data to persist beyond the life cycle of a container. It is crucial to be cautious with the mount point used for these volumes. Incorrectly mounting them against sensitive parts of the host OS could lead to security risks or system disruptions. The following code snippet is a volume configuration for a MongoDB container to ensure data persistence.

```
#Docker Compose for MongoDB
...
volumes:
  - './data:/data/db'
...
```

To verify the functionality of data persistence using Docker volumes, tests were conducted to ensure that data within the specified volume would remain intact even after the container was stopped and removed and then started again. Once the environment was set up, data was inserted into the MongoDB database. This was done by accessing the MongoDB container using the 'docker exec -it <mongo_container-id> /bin/bash' command and then using the MongoDB CLI to insert sample data into the database.

After the data was successfully inserted, the MongoDB container was stopped and removed. To verify data persistence, the Docker container was then started again and checked for data

persistence.

5.3 Resource Management

Implementing strict resource limits on Docker containers can serve as a defense mechanism against scenarios introducing host failures caused by erroneous container services. A controlled stress simulation was conducted on a Node.js container to demonstrate Docker's capability to enforce these limits under simulated stress tests. Tests were performed to verify if setting strict CPU (see section 5.3.1) and memory limits (see section 5.3.2) would allow the containers to handle resource exhaustion.

Resource limits were defined in the Docker Compose file to control each container's maximum CPU and memory usage. Below is the configuration for the Node.js container.

```
#Docker Compose for Node.js application
...
node:
  ...
  deploy:
    resources:
      limits:
        cpus: '0.5' # Limit to half CPU core
        memory: 512M # Limit to 512 megabytes of memory
  ...
```

In this setup, the Node.js container was restricted to use half a CPU core and 512 MB of memory. These limits were applied to demonstrate a constrained resource environment.

5.3.1 Simulating CPU Stress

The 'stress-ng' tool was installed in the Node.js container to simulate what would happen when CPU loads were increasing[22]. The command, 'docker exec node_container stress-ng --cpu \$3 --timeout \$60s' initiated three worker processes in the 'node_container'. Each process attempted to maximize CPU utilization and simulate a scenario where multiple processes tried to consume more CPU resources than had been allocated.

The resource utilization was monitored using the 'docker stats node_container' command, which provided real-time data on CPU usage. Additionally, Docker Desktop was used to analyze the CPU usage during the simulated attack visually.

5.3.2 Simulating Memory Stress

To simulate a scenario where more than 512 MB of memory is allocated, a memory-intensive process was run within the container to trigger the memory limits. This test aimed to demonstrate the memory constraints by generating an error message indicating that the program could not allocate additional memory.

After the memory limits were set in the Docker Compose file, a route in the Node.js container was added, including the necessary code to simulate memory stress and ensure that the application threw an error once the memory limit was exceeded.

The following is a code snippet of the Node.js application.

```
#Node.js "app" application
...
app.get('/memory-stress', async (req, res) => {
  const allocateMemory = () => {
    const memoryArray = [];
    try{
      while(true){
```

```

        memoryArray.push(new Array(1024).fill("*"));
        console.log(os.totalmem());
    }
} catch(error){
    res.status(500).send("Memory limit is reached!");
}
};
allocateMemory();
});
...

```

The test was conducted by accessing the '/memory-stress' route through 'http://localhost:3000/memory-stress'.

To analyze and monitor the results Docker desktop was used to monitor the container's memory usage in real-time.

5.4 Network Security

Managing how containers interact with both internal and external components is crucial for maintaining a secure Docker environment. This section focuses on how Docker containers were configured to handle internal communications with other containers, as well as external access to and from the internet. Configurations concerning port traffic, handling ports, and network setups were demonstrated to investigate security strategies to minimize risks associated with unauthorized access and data breaches.

5.4.1 Injection Vulnerability Test

Addressing security vulnerabilities associated with Docker container volumes is crucial for protecting systems from unauthorized access and manipulation of information. The purpose of this experiment was to investigate how Docker container volumes might be exploited through injection attacks, specifically targeting unauthorized file manipulation within a single container's volume. The Docker environment was configured to create a scenario in a single container that could potentially expose the system to security risks through its volume. The setup included a Node.js container named 'node_container' with an attached volume to simulate this concern. The experiment aimed to test the container against unauthorized file manipulations through command injection attacks, the kinds of vulnerabilities that could be exploited in production environments.

Figure 6 illustrates the injection vulnerability test setup. The 'node_container' was deployed on the host with a mounted volume located at './data:/data/db'. This setup was done to test potential injection vulnerabilities by simulating external attacks targeting the container's volume.

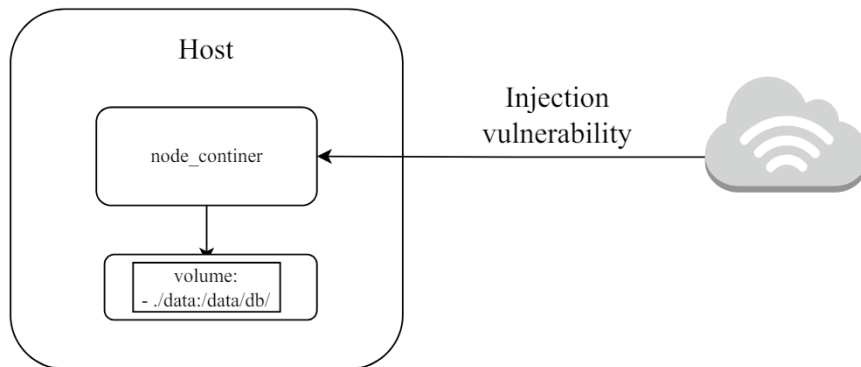


Figure 6: Illustration of HTTP Injection Test

The Node.js application within the 'node_container' was configured to include a route that executed user input commands, particularly to respond to the 'cmd' command. The following is a code snippet of the Node.js application.

```

#Node.js "app" application
...
app.get('/cmd', (req, res) => {
  const command = req.query.cmd; // Get command from URL query

  console.log(`Executing command: ${command}`); // Log the command for monitoring

  // Execute the command received from the query
  exec(command, { cwd: '/app' }, (error, stdout, stderr) => {
    if (error) {
      console.error(`Execution error: ${error}`); // Log execution errors
      // Respond with error details
      return res.status(500).send(`Command failed: ${error.message}`);
    }
    if (stderr) {
      console.log(`Standard Error: ${stderr}`); // Log errors reported to stderr
      // Respond with stderr content
      return res.status(500).send(`Error output: ${stderr}`);
    }
    // Respond with success message and command output
    res.send({ message: 'Command executed successfully', output: stdout });
  });
});
...
  
```

To demonstrate the test a file named 'critical.txt' was created within the 'node_container' volume to act as the target for the security test. This file was intended to represent a critical configuration file in a business environment. The following commands were used to create and insert content into the file. For accessing the container to create a file, 'docker exec -it <node_container-id> touch /data/db/critical.txt' was used. Then the text was inserted in the file by

```
'echo "This is some critical textfile" > /data/db/critical.txt'.
```

To evaluate the container's security against file manipulation, an HTTP request was made to attempt to delete 'critical.txt' through a command injection. The following HTTP request was made, 'curl "http://localhost:8097/cmd=rm critical.txt"'. This test aimed to demonstrate whether the Node.js application in the container could be manipulated through an injection attack to perform unauthorized file deletions.

5.4.2 Node.js Application Configuration and Network Testing

The purpose of this test was to assess container network behavior and its exposure to potential vulnerabilities, especially focusing on the ability to interact to/from internal and external networks under different configurations. Therefore, blocking container applications to do external access can be a good practice.

The experiment aimed to test the 'node_container' against unauthorized network access and manipulation. Two parts of the test were conducted: one without a proxy (detailed in section 5.4.3) and one with a reverse proxy setup (detailed in section 5.4.4). Both parts involved the same Express.js server configuration to maintain consistency and reliability in the results.

A simple Express.js server was configured in 'node_container' with endpoints to test the injection example.

- '/internal-access' : A basic route that responded with "Access to the local application!" to confirm the availability of the container's web server.
- '/external-access' : A route for fetching data from an external source where 'https://mirakel.nu/' was used for demonstration purposes to explore the container's capability of accessing internet resources.

The following is a code snippet of the Node.js application.

```
#Node.js "app" application
...
app.get('/internal-access', (req, vres) => {
  vres.send('Access to the local node application!');
});
app.get('/external-access', (req, res) => {
  const url = https://mirakel.nu/;

  https.get(url, (externalRes) => {

    //Array to collect chunks of data
    let data = [];

    ...

...
}).on('error', (err) => {
  console.error('Error:', err.message);
  res.status(500).send({
    message: 'Failed to Fetch External Data!',
    error: err.message
  });
});
...

```

The test was then conducted in two parts.

- **Local Access Test:** This part involved accessing 'http://localhost:3000' and 'http://localhost:3000/internal-access' from the host to verify internal communication. The aim was to ensure that the internal routes of the container were working correctly and could be accessed from the host.
- **External Access Test:** This part involved accessing 'http://localhost:3000/external-access' from the host, to validate that external internet resources were blocked from the 'node_container'. This test aimed to ensure that the container was isolated from the internet and could not fetch external data.

5.4.3 Docker Network Configuration direct exposing Node.js Container

The Docker Compose setup defined a 'backend' network with 'internal: true' while hopefully blocking all external internet access within the 'backend_network'. This setting ensured that containers could still interact with each other while blocking all external access. This configuration was important for the application's functionality where the Node.js application needed to interact with MongoDB.

Fig 7 illustrates the isolated network setup. Where the 'node_container' is connected to the 'frontend_network', allowing it to interact with external systems through mapped ports, while the 'mongo_container' and 'node_container' communicate over the 'backend_network', which is isolated from external access.

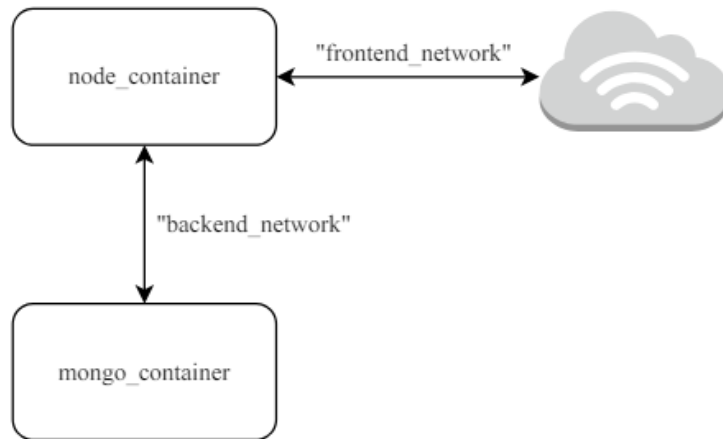


Figure 7: Isolated network setup with external access via mapped ports

5.4.4 Docker Network Configuration with reverse proxy

The network setup was constructed similarly to the initial configuration but with an added reverse proxy to isolate the Node.js application and prevent direct external access from the 'node_container'. This setup would allow monitored external internet access by directing outgoing traffic through the 'proxy_container'. A simple Nginx proxy was implemented for demonstration. The Nginx reverse proxy configuration was constructed to route requests while securing the 'backend_network' containers from external interactions. The following is a code snippet from the setup of the nginx.conf settings.

```
#nginx.conf file
...
location / {
    proxy_pass http://node_container:3000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}

location /internal-access {
    proxy_pass http://node_container:3000/internal-access;
```

```

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}

location /external-access {
    proxy_pass http://node_container:3000/external-access;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}
...

```

Figure 8 illustrates the isolated network setup with the 'proxy_container' added for external access. The 'node_container' and 'mongo_container' communicates over the 'backend_network' which is isolated from direct external access. External traffic is routed through the 'proxy_container', which was connected to both the 'backend_network' and the 'frontend_network'. To ensure that all outgoing traffic from the 'node_container' was monitored and controlled, improving security by preventing direct exposure of the Node.js application to external networks.

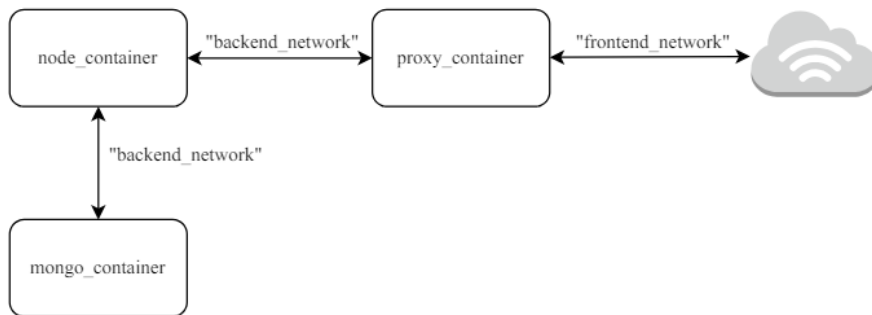


Figure 8: Isolated network setup with 'proxy_container' for external access

6 Results and Discussion

6.1 Vulnerability Scanning

6.1.1 Snyk

The integration of Snyk helped with continuous findings of vulnerabilities during development, ensuring more secure and updated images. It provided recommendations for safer images when vulnerabilities were found. During the tests, the versions of the base images were continually updated based on these recommendations to identify the most suitable image with the fewest vulnerabilities. This section analyses the outcomes of the Snyk security scans on Docker images to demonstrate how updating Docker images reduces vulnerabilities.

The initial scanning results of the 'node:14' Docker base image, as seen in Figure 9, revealed a high number of vulnerabilities. This image was built on 'debian:10', as shown in Figure 15. These vulnerabilities were categorized into 'Critical (C)', 'High (H)', 'Medium (M)', and 'Low (L)' severity issues, also seen in Figure 9. Each vulnerability had an attached score, see example in Figure 11 and Figure 12.

Recommendations for upgrading the base image

	BASE IMAGE	VULNERABILITIES	SEVERITY
Current image	node:14	498	6 C 58 H 66 M 368 L
Mejor upgrades	node:18.20.1	180	1 C 3 H 0 M 176 L

[Show more upgrade types](#)

[View docs](#)

Figure 9: Vulnerability Scanning of 'node:14'

PROJECT OWNER Add a project owner	SOURCE GitHub	TARGET OS debian:10
BASE IMAGE node:14	REPOSITORY Exjobb	MANIFEST project-root1/NodeApp1/Dockerfile
BUSINESS CRITICALITY Add a value	LIFECYCLE Add a value	LINKED IMAGES Add a value

Figure 10: Included Information of Vulnerability Scanning of 'node:14'

Snyk uses severity levels to estimate the risk of vulnerabilities in the applications, categorizing them as 'Critical (C)', 'High (H)', 'Medium (M)', and 'Low (L)'. These levels help to prioritize issues, with 'Critical (C)' indicating a risk of attackers running code or accessing sensitive data, and 'Low (L)' indicating minor data exposure. The severity levels also affect the Snyk Priority Score. According to Snyk documentation, the Snyk Priority Score helps to prioritize issues on a scale from 0 to 1000, where a higher score indicates a more critical issue that needs to be addressed first. This score is calculated based on various factors, including the exploit's maturity, the vulnerability's exposure, its ability to be fixed, the age of the vulnerability, and whether the issue comes from a malicious package. These factors ensure that each issue is evaluated thoroughly and assigned a unique priority score.

Detailed analysis was included in each discovered vulnerability, and provided insights into their root cause and potential impacts. The vulnerabilities were further categorized by their exploit

status. 'MATURE' vulnerabilities for exploits that have been developed and are known to the public and 'NO KNOWN EXPLOIT' for those who haven't been exploited, as seen in Figure 11 and 12.

Figure 11 shows a critical vulnerability (CVE-2023-38408) in the OpenSSH client, identified as "Unquoted Search Path or Element". This vulnerability was assigned a high priority score of 714 out of 1000. The vulnerability was introduced in 'node:14' Docker image, specifically with an outdated OpenSSH client version 'openssh/openssh-client@1:7.9p1-10+deb10u2'. This outdated version came from the image being built on 'debian:10'. Updating to 'openssh/openssh-client@1:7.9p1-10+deb10u3' fixed this issue.

The vulnerability was categorized under 'NO KNOWN EXPLOIT' suggesting that while the vulnerability is critical there are no known active exploits targeting this vulnerability.

C openssh/openssh-client - Unquoted Search Path or Element [🔗](#) SCORE
714

VULNERABILITY | CWE-428 [🔗](#) | CVE-2023-38408 [🔗](#) | CVSS 9.8 [🔗](#) | **CRITICAL** | SNYK-DEBIAN10-OPENSSSH-5788320 [🔗](#)

Introduced through	openssh/openssh-client@1:7.9p1-10+deb10u2	Exploit maturity	NO KNOWN EXPLOIT
Fixed in	openssh/openssh-client@1:7.9p1-10+deb10u3, @1:7.9p1-10+deb10u3		

Show less detail [^](#)

Detailed paths

- Introduced through: node@14 · openssh/openssh-client@1:7.9p1-10+deb10u2
- Fix: Upgrade to openssh/openssh-client@1:7.9p1-10+deb10u3 [🔗](#)

Figure 11: Vulnerability 'Critical (C)'

Figure 12 shows a medium severity vulnerability in the 'openssh/openssh-client' package, specifically version '1:7.9p1-10+deb10u2'. To handle this issue the recommendation was to upgrade to a newer version 'openssh/openssh-client@1:7.9p1-10+deb10u4'. The vulnerability had a Priority Score of 621 considered as a medium priority, marked with a 'PROOF OF CONCEPT' status, suggesting that while an exploit exists, it may not be widely implemented or actively used in attacks yet. This underscores the importance of regular updates and continuous scanning for vulnerabilities, as these might change over time.

M openssh/openssh-client - Improper Validation of Integrity Check Value [🔗](#) SCORE
621

VULNERABILITY | CWE-354 [🔗](#) | CVE-2023-48795 [🔗](#) | CVSS 5.9 [🔗](#) | **MEDIUM** | SNYK-DEBIAN10-OPENSSSH-6130526 [🔗](#)

Introduced through	openssh/openssh-client@1:7.9p1-10+deb10u2	Exploit maturity	PROOF OF CONCEPT
Fixed in	openssh/openssh-client@1:7.9p1-10+deb10u4, @1:7.9p1-10+deb10u4		

Figure 12: Vulnerability 'Medium (M)'

Figure 13 shows the final update of the base image to 'node:20.12.1-bookworm-slim', with a recommendation for further upgrading to 'node:21.7.3-bookworm-slim'. This upgrade significantly reduced the number of vulnerabilities in the beginning from 498 to 38, demonstrating the importance of maintaining images regularly to improve security.

Recommendations for upgrading the base image

	BASE IMAGE	VULNERABILITIES	SEVERITY
Current image	node:20.12.1-bookworm-slim	38	1 C 1 H 0 M 36 L
Major upgrades	node:21.7.3-bookworm-slim	37	1 C 1 H 0 M 35 L

[Show more upgrade types](#)

[View docs](#)

Figure 13: Vulnerability Scanning of 'node:20.12.1-bookworm-slim'

Snyk was found user-friendly, displaying detailed vulnerability information and recommendations for fixes, such as updating to newer image versions to address critical vulnerabilities and reduce exploit risks. For example, critical vulnerabilities often came with a "Fix" recommendation, which could easily be accessed by clicking on the vulnerability for more detailed information. This feature made it simpler for me as a user to manage and investigate the risks. The integration of automated security tools like Snyk into the CI/CD pipeline is crucial, as it enables regular suggestions for updates and scanning. This process not only helps with minimizing potential security threats but also highlights the importance of regular updates in maintaining a secure Docker environment. Overall, the detailed analysis provided by Snyk underscores the critical impact of regular updates and the value of integrating these automated tools into development processes, as this approach helps identify potential security threats before deployment.

6.1.2 Docker Scout

Docker helped analyze and find vulnerabilities during the development process. Figure 14 shows the Docker Scout analysis of the base image 'node:14'. It provided a detailed analysis of the image hierarchy, vulnerabilities, and the layered structure of the Docker image. Each image in the hierarchy underwent vulnerability scanning, as seen on the right side of Figure 14. This scanning process provided insightful security status for every layer. The vulnerabilities were categorized based on their severity

In figure 14, it was possible to see that the Docker image consisted of 20 layers. These layers included various commands like adding files, setting environment variables, and installing necessary packages. Each layer not only contributed to the functionality of the final image but also inherited its own set of vulnerabilities from the base images.

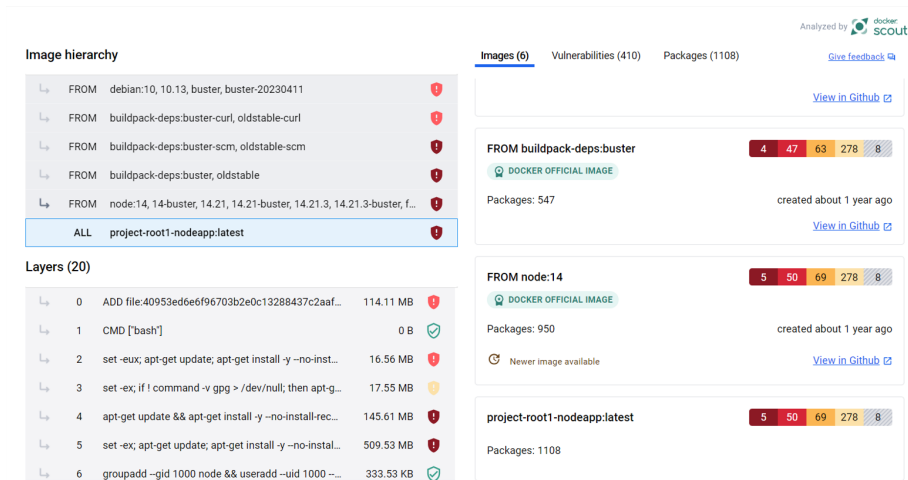


Figure 14: Vulnerability Scanning of 'node:14'

The image hierarchy seen in figure 15 showed the base images and the layers that the final Docker image was built upon, tagged as 'project-root1-nodeapp:latest'. The hierarchy revealed that the 'node:14' image was built on top of several layers, starting from the 'debian:10' image, followed by various of buildpack dependencies such as 'buildpack-deps:buster' and finally the 'node:14' image itself. Where each layer adds specific functionalities and dependencies required by the Node.js runtime.

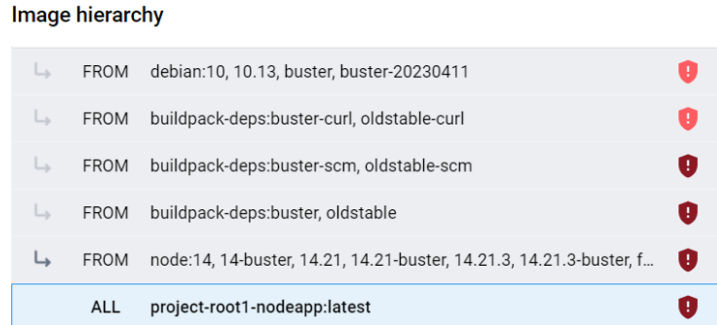


Figure 15: Image hierarchy of 'node:14'

Figure 16 shows the recommended fixes for the base image 'node:14'. The preferred tag recommended was '22-slim' which was smaller in size by 261 MB, with fewer packages, and introduced no new vulnerabilities while removing 370 existing ones. Additionally, the '22-slim' tag was a more recent push, indicating it included the latest updates and security patches. Other recommended updates were also seen, such as '18:slim' and '21-slim', each offering significant reductions in vulnerabilities and packages.

Recommended fixes for base image node



[Refresh base image](#)

[Change base image](#)

[Learn about Docker Scout](#)

Select the tag you would like to see recommendations for. The list displays new recommended tags in descending order, where the top results are rated as most suitable.

Image options	Tag	Age	Vulnerabilities
Current image	14	about 1 year	5 49 64 275 7
Tag is preferred tag	22-slim Also known as: 22.2.0-slim, 22.2-slim, curre...	9 days	0 0 0 23 0 -5 -49 -64 -252 -7
Major runtime version update	18-slim Also known as: 18.20.3-slim, 18.20-slim, hy...	4 days	0 0 0 23 0 -5 -49 -64 -252 -7
Major runtime version update	21-slim Also known as: 21.7.3-slim, 21.7-slim, 21-b...	11 days	0 0 1 23 0 -5 -49 -63 -252 -7

Benefits:

- Image is smaller by 261 MB
- Tag is preferred tag
- Tag was pushed more recently
- Tag is using slim variant
- Image contains 621 fewer packages
- Major runtime version update
- Image introduces no new vulnerability but removes 370

Replace your old tag with this new one in your Dockerfile

```
FROM node:22-slim
```

Figure 16: Recommendations for updating

For demonstration purposes, the 'node:20.12.1-bookworm-slim' was chosen to compare it to the other scanning tool Snyk. Figure 17 shows the vulnerability scanning of the updated base image. The image hierarchy seen in figure 17, indicates that the updated base image is built upon 'debian:12-slim' and 'node:20-bookworm-slim'. The updated image consists of 16 layers, compared to the 20 layers in the previous 'node:14' base image.

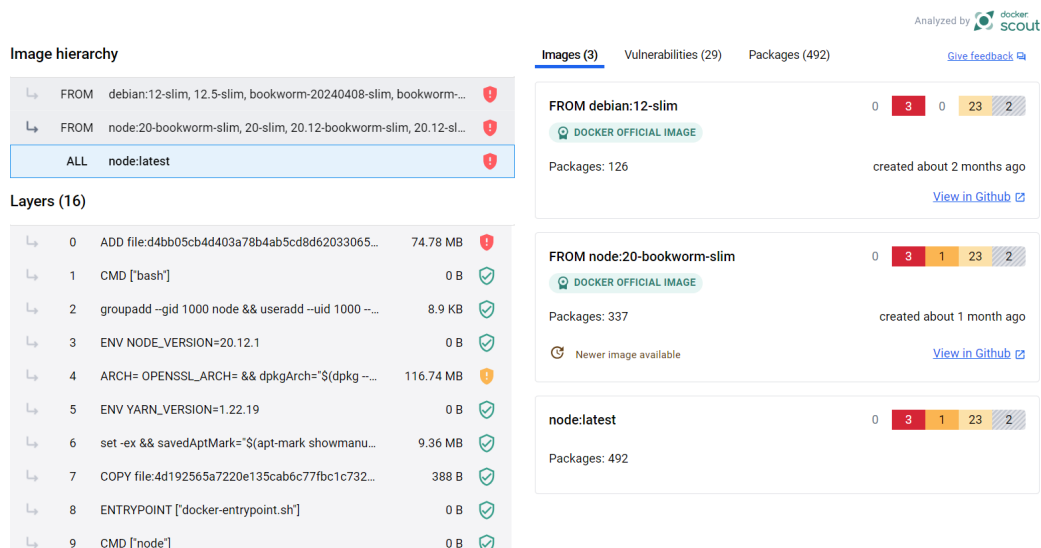


Figure 17: Vulnerability Scanning of 'node:20.12.1-bookworm-slim'

In the vulnerability analysis provided by Docker Scout, the 'debian:12-slim' base image had 3 high-severity vulnerabilities but no critical vulnerabilities, indicating a relatively secure base. The 'node:20.12.1-bookworm-slim' image builds on this base image with an additional one medium-severity vulnerability but also introduces some improvements providing a newer version of the Node.js runtime '20.12.1'.

The overall vulnerabilities for the 'node:latest' image, based on the 'node:20.12.1-bookworm-slim', showed a significant reduction in vulnerabilities compared to the 'node:14' image. The new image has 3 high-severity and 1 medium-severity vulnerability which is a clear improvement from the 5 critical, 50 high, 96 medium, and 278 low vulnerabilities found in the node:14 image.

This visualization was straightforward and could be particularly useful when employing multi-stage builds, to keep the images as minimal as possible. For example, one could use a base of Debian for some stages and other minimal bases for different parts of the build to maintain a smaller final image.

The Docker Scout tool provided a clear and detailed visualization of the image hierarchy, showing how different images are built and linked together. This helped in understanding the complexity of the build and identifying where improvements could be made to increase the security. The ability to see the image hierarchy easily can help in building smaller images, which can be beneficial for improving performance and reducing resource usage.

Using tools like Docker Scout offers several benefits for maintaining and improving the security of Docker images. The detailed analysis of image hierarchy and layered structure helps developers understand the build of their images and identify potential security risks at each layer. The categorization of vulnerabilities by severity and the CVSS scores enable the priority of security fixes, ensuring that critical issues are addressed faster.

Moreover, Docker Scout's recommendations for updating base images to more secure and efficient versions highlight the importance of using tools like this. This approach not only enhances security but also optimizes image performance by reducing size and dependencies.

Overall, the use of Docker Scout in the development process provided a comprehensive and

user-friendly method for managing Docker image security, by offering clear visualization and important insights, supporting the creation of secure, efficient, and well-maintained Docker environments.

6.1.3 Evaluating Docker Scout and Snyk

While Docker Scout and Snyk both offer real-time scanning capabilities, integrating both tools into the security setup can significantly enhance vulnerability management due to their unique detection methodologies and database differences. Each tool's approach to scanning and analysis can lead to different vulnerabilities being detected, which can be crucial for a comprehensive security review. Using Docker Scout and Snyk together provides an effective method for cross-validation. If one tool identifies a vulnerability that the other ignores, these differences can help with the overall security.

Moreover, the varied fixed solutions offered by each tool give developers wider options to address identified vulnerabilities. This includes everything from immediate patches or bigger changes. Despite some functional overlap, employing both Docker Scout and Snyk can strengthen security practices by ensuring a more layered and thorough approach to vulnerability detection and management.

6.2 Privileges

6.2.1 Non-root User in Windows

The test conducted in the Docker environment running on Windows with the WSL2 Linux subsystem showed insightful results regarding user privileges within containers. The outcome was that the user within the container could not delete the 'Test' folder located in the host's Windows directory. This confirmed that the container's user did not have root privileges on the host system without requiring specific non-root configurations. This finding is interesting as Docker containers are configured to run as root by default, which sets an increased risk of privilege escalation attacks.

In environments like Linux, it is often necessary to configure Docker containers to run as non-root users to minimize these risks. However, the results of this test indicate that in the Windows environment with WSL2, such specific configurations are not necessary to prevent container users from accessing critical host system files.

By demonstrating that the container user was unable to modify system-level files in the host environment, the test investigated the practice of using non-root configurations in the Docker environment to strengthen security. However, it was possible to see the content in the folder and read files. Because of this, it is important to be careful where the volumes are mounted, to maintain the security of the host system. This configuration ensures that even if the container is hacked, the potential damage to the host system would be minimized.

6.2.2 Read-only Volumes

The tests to verify the functionality of read-only volumes in the Node.js container demonstrated the effectiveness of this security measure. Any attempts to create, modify, or delete files in the read-only volumes resulted in permission-denied errors, indicating that the volume was indeed read-only. These results confirm that the read-only volume configuration successfully prevented unauthorized modifications to the data within the volume. This approach is crucial for enhancing security by ensuring that critical data remains immutable and protected from potential malicious actions by processes running inside the container. By implementing read-only volumes, it is

possible to reduce the risk of data tampering and unauthorized modifications. This configuration helps in maintaining the security of sensitive data, which can be especially important in environments where containers handle critical applications and data.

The results of this test underline the importance of read-only volumes in securing containerized environments, by preventing any changes to the data after deployment. This security measure is aligned with the Principle of Least Privilege, ensuring that containers only have the necessary permissions to function without exposing critical data to medication risks. In conclusion, read-only volume configuration proved to be an effective method for protecting data integrity within Docker containers. The permission-denied errors observed during the tests confirm that this configuration can effectively protect sensitive data.

6.2.3 Volumes For Data Persistence

The tests confirmed the functionality of data persistence using Docker volumes. After restarting the container, the previously inserted data was still present in the MongoDB database. This verified that volume configuration allowed the data to persist beyond the life cycle of the container.

The outcome of this test demonstrated the effectiveness of Docker volumes in ensuring data persistence. By mounting a volume to the host directory, the data remained intact even after the container was stopped and removed. This approach is essential for applications that require reliable data storage, ensuring that data is not lost between container restarts.

Additionally, the test highlights the importance of careful volume configuration to avoid mounting volumes against sensitive parts of the host OS, which could pose security risks or lead to system disruptions. In conclusion, the successful verification of data persistence underscored the importance of Docker volumes in managing data efficiently within containerized environments.

The result of these tests demonstrated that the volume configuration for data persistence worked as expected. The data inserted into the MongoDB database remained intact even after the container was stopped and removed, confirming that the data was correctly persisted on the host machine. This approach is crucial for applications that require persistent data storage, as it ensures that critical data remains available across container restarts and deployment. It also highlights the importance of carefully managing volume mount points to avoid potential security risks or system disruptions.

6.3 Resource Management

6.3.1 Simulating CPU Stress

As seen in Figure 18, the CPU usage was minimal at the beginning of the test indicating that the container was inactive before the stress conditions were applied. Once 'stress-ng' was initiated with three worker processes, there was an increase in CPU usage until it stabilized around 45% and did not exceed the 50% limit set for the container.

The stabilization of around 45% confirms the effectiveness of Docker's resource constraints, clearly demonstrating that the CPU usage remained constant once it reached the predefined limit. Such controlled CPU usage underscores Docker's ability to effectively manage the resources allocated to each container, ensuring that no container exceeds its resource allocation. This is important to stop any container from using up too much of the system's host resources.

The results aligned with the expectations of the test, demonstrating that Docker's resource limitations can serve as a defense mechanism against scenarios with services handling resources

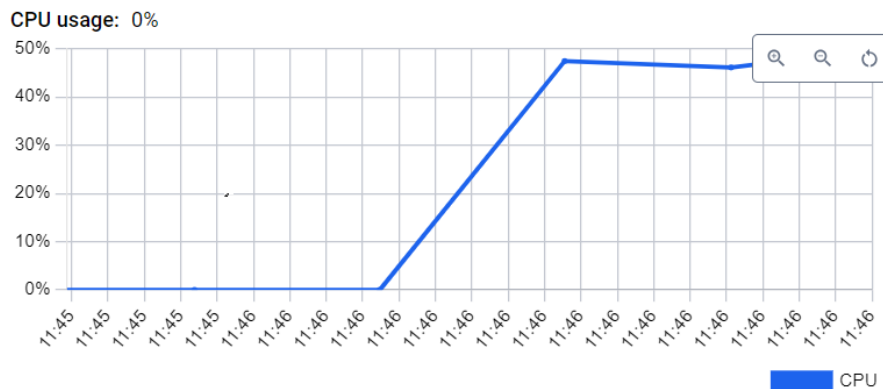


Figure 18: CPU Utilization During "stress-ng" Test

poorly by exploits or bad code that could lead to resource exhaustion. This test not only demonstrates Docker's ability to enforce resource limitations but also underscores the importance of managing resource use effectively in multi-container environments.

6.3.2 Simulating Memory Stress

During the test, it was observed that when the memory usage within the container reached its limit of 260 MB, the container was terminated. This termination is a result of Docker's default security settings, which are designed to prevent the container from negatively impacting the host system by exceeding its allocated memory.

This test effectively demonstrated how Docker's resource limits can be used to manage container memory usage, ensuring that applications do not exceed their allocated resources and preventing potential system instability. Constraining memory in Docker containers is crucial for several reasons, particularly from a security perspective.

Limiting the memory usage of a container prevents a single container from consuming all available memory resources, which could lead to a choke on the host and affect all services running on it. An attacker could intentionally run a memory-intensive process to exhaust the system's memory, causing other services to fail. Memory limits help maintain the overall stability of the host system. Without these constraints, a memory leak in one container could exhaust the host's memory, negatively affecting other containers and the host's performance. Setting memory limits ensures predictable resource allocation, which is important in multi-tenant environments where different applications or services share the same host resources. By preventing a container from exceeding its allocated memory, these constraints enhance the security and stability of the system.

In conclusion, the successful simulation of memory stress demonstrated the effectiveness of Docker's memory constraints. Implementing memory limits is a crucial aspect of container management, as it ensures that no single container can disrupt the performance and reliability of the host system.

6.4 Network Security

6.4.1 Injection Vulnerability Test

The command injection test in Docker volumes highlighted a significant security vulnerability, where the successful execution of a command to delete the 'critical.txt' file underscored

the risk of unauthorized file tampering through command injections. It was found that with a default setting, mimicking a very simple type of injection scenario it was possible to delete 'critical.txt' directly through HTTP requests.

Setting the Docker volume to read-only effectively stopped the file deletion attempt. This approach blocked write operations at the filesystem level and showed a practical security measure for volumes containing static configuration files that would not require updates. Moreover, configuring the container to operate with non-root user privileges also prevented the deletion of 'critical.txt'. This result aligns with the principle of least privileges, which ensures that processes should only operate with necessary permissions.

In scenarios where volumes are used by multiple containers, the security risks increases. If multiple containers would access the same volume, it would only take one container to affect the entire system. Therefore, isolating sensitive data into separate volumes minimizes the risk.

The effortless removal of 'critical.txt' through a simple HTTP request underscores the importance of security configurations in Docker volumes to prevent unauthorized access and modifications.

This result serves as a reminder that the vulnerabilities explored here, could apply to other sensitive data and scenarios. It highlights the importance of careful security strategies to protect Docker environments from command injection and other types of attacks.

6.4.2 Docker Network Configuration direct exposing Node.js Container

The setup of the 'backend_network' with the 'internal:true' setting successfully blocked all external internet access to the 'mongo_container'.

However, the 'frontend_network' enabled access from the host but also external access from the 'node_container' which was an undesirable result. Due to this, it could pose a security risk since malicious code could be downloaded within the 'node_container'.

This finding highlights a critical aspect of Docker network configurations, as there is a need to balance security with accessibility. The strict network isolation in the 'backend_network' effectively prevented unauthorized external communications, which is crucial for protecting against potential attacks such as data breaches. But since the existence of the 'frontend_network', the system was not fully secured.

This test demonstrated the importance of carefully testing Docker network settings in a controlled environment before deployment. Suggesting that network access needs to vary depending on the application, in some cases, restricting access to a private network can fulfill the specific need. These insights are vital for creating a secure Docker environment, demonstrating the need for carefully managed network configurations to achieve an optimal balance between security and functionality.

6.4.3 Docker Network Configuration with reverse proxy

The introduction of the 'proxy_container' as a reverse proxy allowed the 'node_container' accessing internal network resources while maintaining strict network isolation policies and not having complete external access. This setup demonstrated how a reverse proxy could handle controlled access within a secure isolated network environment.

The 'proxy_container' enabled external access to the service in 'node_container' without compromising network security within the 'backend_network' blocking external access from containers. It was possible to call 'localhost:3000/internal-access', aligning with the expected outcome demonstrating the proxy's role in efficiently managing internal requests.

However, attempts to reach `'localhost:3000/external-access'` was unsuccessful, which was expected and confirmed the efficiency of the isolation strategy.

The use of a reverse proxy in this setup provided several significant benefits. Firstly, it enhances security by acting as a channel between the client and the `'node_container'`, preventing direct access to the `'node_container'` thereby reducing the attack surface. This isolation is crucial for protecting sensitive backend services from potential external threats. Additionally, the `'proxy_container'` effectively managed incoming traffic, handling incoming requests by filtering them through predefined rules in the Nginx configuration. This ensured that only specified routes, such as `'localhost:3000/internal-access'`, could reach the `'node_container'`, enhancing control over the network traffic.

Moreover, other benefits not demonstrated in the test include that a reverse proxy can distribute incoming traffic across multiple backend servers, improving load balancing and scalability. This is particularly beneficial for handling high traffic volumes and ensuring consistent application performance. The reverse proxy can also manage SSL/TLS, simplifying the process of securing communication between clients and the application, certificate management, and reducing the complexity of configuring SSL/TLS on each backend server.

The results confirmed that the `'proxy_container'` functioned effectively as a reverse proxy by serving as a controlled gateway for incoming and outgoing traffic. This setup was crucial in environments requiring strict security against external threats, ensuring that external connections are not permitted from the backend network. This controlled approach ensured that external access was carefully blocked and provided a secure bridge between the internal applications and external clients while maintaining the security of the internal network.

This test demonstrated the importance of carefully configuring and testing Docker network settings in a controlled environment before deployment. It is essential to assess on a case-by-case basis whether an application requires external access or not. For applications requiring access in a controlled environment from the host controlling/preventing outgoing traffic, this reverse proxy approach proved to be effective and secure.

7 Conclusion

This thesis aimed to explore the security aspects of Docker, particularly in the context of deploying Node.js applications. Throughout the research and experiments, it became evident that while Docker offers a robust framework for containerization, it does not automatically guarantee security. Proper configuration and continuous management are essential to utilize its full potential for securing IT environments. The analysis included vulnerability scanning using Docker Scout and Snyk, highlighting the security improvements achieved by transitioning from the 'node:14' to 'node:20.12.1-bookworm-slim' image. The reduction of vulnerabilities underscored the importance of using up-to-date and minimal base images. Multi-stage builds were also found to be effective in creating smaller and more secure images by removing build dependencies from the final runtime environment.

A key observation from this study was that the foundation of the base image found 'debian:10', played a critical role in the overall security of Docker images. The base image often contains multiple layers, each potentially introducing vulnerabilities. The foundations of these images can present security risks, that may not be immediately obvious, underscoring the need to carefully select and manage base images. Ensuring the use of minimal and well-maintained base images can mitigate some of these risks.

The comparative evaluation of Docker Scout and Snyk demonstrated that using multiple scanning tools enhances vulnerability management by providing diverse detection methodologies and recommendations. This cross-validation ensures a more comprehensive security view. Experiments with non-root user configurations and read-only volumes further highlighted the importance of following the Principle of Least Privilege and protecting data integrity within containers. These measures are crucial in preventing unauthorized access and modifications, thereby enhancing container security. Resource management tests, including simulations of CPU and memory stress, validated Docker's ability to enforce resource constraints effectively. This is essential for maintaining the stability and performance of multi-container environments, particularly in preventing resource exhaustion attacks. Network security configurations, such as setting up private networks and using reverse proxies, showcased the ability to isolate containers and control network traffic efficiently. These configurations are pivotal in safeguarding sensitive backend services from external threats while maintaining necessary accessibility for development.

The findings from this thesis confirm that while Docker provides a flexible and efficient environment for application deployment, it requires careful configuration to ensure security. Tools like Docker Scout and Snyk play a vital role in identifying and mitigating vulnerabilities, but they must be used as a part of a broader security strategy. Additionally, the choice of operating system and the continuous monitoring and updating of Docker environments are critical factors in maintaining security. As Docker continuously evolves, staying up-to-date with updates and best practices is essential. Looking forward, the next step involves exploring Kubernetes for the orchestration and scaling of containerized applications. Kubernetes offers additional layers of management and security features that can further enhance the deployment and maintenance of secure Docker environments.

In conclusion, Docker does not inherently make a system more secure. However, when used correctly and combined with the best security practices, it can be a powerful tool for protecting and maintaining IT environments. This thesis provides a foundational understanding and practical guidelines for leveraging Docker to build secure, efficient, and well-maintained application deployments.

References

- [1] Docker Inc. Install docker desktop on windows. Docker Documentation, . URL <https://docs.docker.com/desktop/install/windows-install/>.
- [2] Tamanna Siddiqui, Shadab Alam Siddiqui, and Najeeb Ahmad Khan. Comprehensive analysis of container technology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 224–225. IEEE, Nov 2019. doi: 10.1109/ISCON47742.2019.9036238.
- [3] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018. doi: 10.1109/CLOUD.2018.00030.
- [4] P.P.W. Pathirathna, V.A.I. Ayesha, W.A.T Imihira, W.M.J.C. Wasala, Nuwan Kodagoda, and Tharindu Edirisinghe. Security testing as a service with docker containerization. In *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, Malabe, Sri Lanka, 2017. IEEE. doi: 10.1109/SKIMA.2017.8294109.
- [5] Docker Inc. Docker overview. Docker Documentation, . URL <https://docs.docker.com/get-started/overview/>.
- [6] Docker Inc. Docker build cache. Docker Documentation, . URL <https://docs.docker.com/build/cache/>.
- [7] Docker Inc. Docker official images. Docker Documentation, . URL <https://docs.docker.com/trusted-content/official-images/>.
- [8] David Dooling. From misconceptions to mastery: Enhancing security and transparency with docker official images. Docker Official Blog, April 4 2024. URL <https://www.docker.com/blog/enhancing-security-and-transparency-with-docker-official-images/>.
- [9] Docker Official Image Team. Node docker official image. Docker Hub, . URL https://hub.docker.com/_/node.
- [10] Docker Official Image Team. Mongodb docker official image. Docker Hub, . URL https://hub.docker.com/_/mongo.
- [11] Docker Inc. Dockerfile reference. Docker Documentation, . URL <https://docs.docker.com/reference/dockerfile/>.
- [12] Docker Inc. About storage drivers, . URL <https://docs.docker.com/storage/storagedriver/>.
- [13] Docker Inc. Use containers to build, share and run your applications, . URL <https://www.docker.com/resources/what-container/>.
- [14] Docker Inc. Networking overview, . URL <https://docs.docker.com/network/>.
- [15] Docker Inc. Manage data in docker, 2024. URL <https://docs.docker.com/storage/>.
- [16] Docker Inc. Why use compose?, . URL <https://docs.docker.com/compose/intro/features-uses/>.
- [17] Docker Inc. Use compose in production, . URL <https://docs.docker.com/compose/production/#running-compose-on-a-single-server>.

- [18] Docker Inc. Swarm mode overview, . URL <https://docs.docker.com/engine/swarm/>.
- [19] Docker Inc. Build kubernetes-ready applications on your desktop, . URL <https://www.docker.com/resources/kubernetes-and-docker/>.
- [20] Donnie Berkholz. Docker index shows continued massive developer adoption and activity to build and share apps with docker, February 10 2021. URL <https://www.docker.com/blog/docker-index-shows-continued-massive-developer-adoption-and-activity-to-build-and-share-apps-with-docker>.
- [21] Aparna Tomar, Diksha Jeena, Preeti Mishra, and Rahul Bisht. Docker security: A threat model, attack taxonomy and real-time attack scenario of dos. In *Proceedings of the 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 150–155, Dehradun, India, 2020. IEEE. doi: 10.1109/Confluence47617.2020.9058115.
- [22] Inc. Red Hat. Chapter 29. stress testing real-time systems with stress-ng. URL https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/assembly_stress-testing-real-time-systems-with-stress-ng_optimizing-rhel8-for-real-time-for-low-latency-operation.