National College of Ireland

Enhancing Docker Container Security

MSc Research Project
MSCCYB1

YELLAMMAGARI SRIKAR PUTTA
Student ID: X21184054

School of Computing
National College of Ireland

Supervisor: EVGENIIA JAYASEKERA

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: YELLAMMAGARI SRIKAR PUTTA

Student ID: X21184054………………………………………………………………………………

Programme: MSCCYB1…………………………………………………………Year: 2022

Module: MSc Research Project/Internship…………………………………………

Supervisor: EVGENIIA JAYASEKERA………………………………………………………………………
Submission Due Date: 14-Aug-2023 ……

Project Title: Enhancing Docker Container Security…………………………………………………

Word Count: …………………………………… **Page Count**…………………………………………………..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: YELLAMMAGARI SRIKAR PUTTA ………

Date: 12-Aug-2023 ………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Docker Container Security

YELLAMMAGARI SRIKAR PUTTA
X21184054

Abstract

With the widespread adoption of Docker for deploying applications comes novel security challenges that attackers are actively exploiting. Common attack vectors include container escapes, image tampering, and privilege escalation. This research conducts an in-depth investigation into Docker container vulnerabilities, attack methods, and security strategies. As developers increasingly utilize Docker often with misconfigured images, attackers can compromise containers and gain unauthorized host access. This work demonstrates experimental attacks exploiting container misconfigurations to breach the underlying host system. It proposes concrete mitigation techniques focused on restricting privileges, limiting dangerous system calls, and reducing Linux capabilities granted to containers. These hardening techniques prevent attackers from reaching the host even in the event of container compromise. The analysis provides developers and devops teams actionable guidance on building more secure Docker images to avoid handing attackers the keys to the host system. By identifying risks, attack vectors, and tailored mitigations, this research enables organizations to benefit from the efficiencies of Docker while protecting against emerging security threats in this landscape. The insights equip developers with the knowledge to strengthen Docker container security as adoption continues to accelerate.

1 Introduction

Docker containers have become very popular as a virtualization technology to deploy applications on different platforms. Docker containers encapsulate the software code along with the necessary runtime environment, making the applications portable so they can run independently of the host environment. Docker, which is built on top of the Linux operating system, uses core features of Linux to achieve containerization. Although Docker containers offer many benefits, such as efficient use of resources and quick deployment, they also introduce security issues, especially when they are not configured properly.

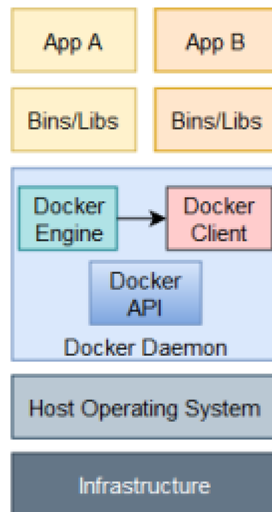

Figure 1: Docker Container

Developers run Docker containers with some misconfigurations like granting elevated privileges to accomplish their application development or to achieve some functions in their deployment process. But they are unaware of how serious the security implications could be. They don't realize how these misconfigurations can assist attackers in gaining access to the host system. There are multiple Docker architecture vulnerabilities that compromises the host system. The Docker daemon acts as an important intermediary between Docker clients and the containers they manage. Users typically interact through the Docker client rather than direct access to the daemon, which operates with root privileges. Only trusted users should be allowed control of the daemon due to this level of access.

Docker enables directory sharing between the host and containers without imposing file access limitations on containers. However, a major security concern arises - a container can be created where its /host directory corresponds to the root / directory of the host itself. This grants the container unrestricted access to modify the host filesystem. This vulnerability poses severe risks. For example, an attacker could exploit it to create arbitrary containers on a Docker host running a web server. By manipulating parameters, they could gain unauthorized access to manipulate or disrupt the host filesystem and critical resources Chelladhurai, Chelliah and Kumar, 2016.

Linux relies on a binary separation of root and non-root access. But for containers, this can introduce risks. Traditionally, a process like a web server binding to a port requires root privileges. This exposes the whole system if the web server is compromised. Linux capabilities provide more granular access control that transforms the rigid root/non-root model. Capabilities mitigate the need for full root container privileges, except for the daemon/manager. Containers can be assigned specific capabilities matching their required tasks. The 38 capabilities cover diverse functions, enabling containers to operate with restricted privileges for improved security. For example, a web server container can have the CAP_NET_BIND_SERVICE capability, allowing it to bind to ports without full root access. If the web server is exploited, this capability confines the attacker’s access to just port binding rather than everything. Additionally, capabilities play a key role in protecting the host from container threats. Containers with inadvertent elevated capabilities could enable an attacker to breach the host system, directly jeopardizing host security. By assigning capabilities to processes, administrators can achieve a balance between security and functionality, ensuring that processes only have the necessary privileges to carry out their intended tasks. However, misconfigurations in the privileges assigned to Docker containers can expose serious security risks. Administrators may inadvertently assign excessive privileges to containers, granting them capabilities beyond what is required for their intended functions. Running Docker containers with elevated privileges, such as assigning more Linux capabilities than necessary, can potentially lead to privilege escalation, unauthorized access to the host system, and compromise of critical resources.

In this research, we will thoroughly examine the interplay between Docker containerization, Linux capabilities, and security vulnerabilities. We will explore real-world scenarios where Docker containers, due to misconfigurations or elevated privileges, can become entry points for attackers to improperly

gain unauthorized access to the underlying host system. By performing real-world experimental analysis, we aim to demonstrate the significant dangers of running Docker containers with unnecessary privileges and highlight the specific methods through which attackers exploit these misconfigurations. This research will serve as a reference for small startups and IT companies who don't have a proper security team. Our study will go beyond theoretical considerations by simulating practical attack scenarios and evaluating their impact. We will also propose effective mitigation techniques to prevent unauthorized access and privilege escalation. By scrutinizing the risks associated with the improper management of Docker container privileges, we intend to provide practical insights that will help administrators, developers, and security experts implement secure containerization practices.
Through this research, we aim to strongly emphasize the importance of properly configuring and managing Docker container privileges to ensure the overall security and integrity of host systems. As a result, our work will contribute to improving the understanding of Docker security best practices and offer actionable recommendations for reducing the risks posed by granting containers elevated Linux capabilities and misconfigured containers.

Research Questions:

How the dockers running with misconfigurations helps the attacker to get access to host system?
What are the defensive mechanisms that we should enforce using AppArmor, Seccomp and other techniques to protect the underlying docker host even the docker container is compromised?

Objectives:

The research aims to thoroughly understand cyberattacks on Docker containers in order to explore the techniques attackers use to gain access to the Docker host and improve defensive approaches by analyzing these techniques. Additionally, the study seeks to empower developers and small IT teams without dedicated security teams to enhance Docker container security using accessible and effective tools. Addressing runtime vulnerabilities and enforcing organization-specific compliance policies are critical goals to ensure Docker systems stay robust and meet stringent security standards. The research further strives to safeguard data integrity and confidentiality in real-time, contributing to the broader advancement of Docker security best practices.

2 Related Work

This literature review explores attack tactics targeting Docker images and containers as well as vulnerabilities in the Docker architecture itself, particularly related to cgroups and containerd. Additionally, various security strategies recommended by different authors for hardening Docker are evaluated. The review looks at attacker techniques to compromise Docker environments and components as well as inherent weaknesses that can be exploited. Proposed defensive measures from multiple sources are also examined to identify effective ways to secure Docker deployments. The goal is to synthesize knowledge on Docker threats, vulnerabilities, and recommended safeguards based on current literature.

Static and Dynamic Code Analysis on Docker Containers:

The author discusses a follow-up analysis conducted after six months, using tools like Clair, Trivy, and Snyk to assess changes in container image vulnerabilities. Notably, some images, particularly those related to stable operating systems, remained relatively unchanged. However, the Log4j incident prompted patching in official images, resulting in decreased vulnerabilities. Snyk detected more low-severity vulnerabilities, while Clair detected fewer vulnerabilities overall, except for the "Unknown" and "None" categories. Trivy showed an increase in "Unknown" and "Critical" vulnerabilities. The author also highlights dynamic exploitation experiments on Raspberry Pi and Kubernetes environments, employing Proof-of-Concept codes for known vulnerabilities Haq, Tosun and Korkmaz, 2022.

The author outlines Docker-sec's methodology for creating container security profiles. Initial static analysis of configurations provides baseline rules. Dynamic testing by monitoring live container actions and extracting required privileges. The combined static and dynamic approaches restrict unauthorized access while permitting normal behavior. Iterative tuning during runtime builds accurate, secure profiles safeguarding containers Alyas et al., 2022.

The author outlines techniques to secure container images - minimizing attack surfaces, signing for integrity, scanning for vulnerabilities, using trusted repositories, and regular patching. A multi-layered approach compensates for limitations in individual practices. Combined, these strategies harden containers despite persisting challenges Wong et al., 2021.

The author performed an analysis focused on evaluating the security of Docker containers using static code analysis techniques. The core goal was to identify vulnerabilities in container implementations and understand their potential impacts. Through the use of regression testing, robustness testing, and other behavioral software analysis methods, the author was able to detect security flaws in Docker container code. A key finding highlighted in the paper is the existence of specific Common Vulnerabilities and Exposures (CVEs) that can significantly affect container environments, as detailed in the below table. This table outlined the effects, causes, and consequences of known CVE vulnerabilities with regards to container security. By uncovering container vulnerabilities through static and behavioral analysis, quantifying their impacts, and applying security upgrades to address them, the author provides crucial insights for hardening Docker containers against risks. This study emphasizes the importance of analyzing container code for vulnerabilities, determining their implications, and remediating them through security fixes to enable more secure container adoption Duarte and Antunes, 2018.

How CVE-2020-15257 allowed attackers to exploit containerd unix socket

The author highlights a serious container security vulnerability caused by lack of access controls on a critical runtime component situated on the host system. If compromised by an attacker, this pivotal component, which lacked proper access restrictions, could enable unauthorized access and container creation privileges. As an example, the author cites CVE-2020-15257 which allowed attackers to exploit the containerd abstract UNIX socket. This socket enabled communication between containers and containerd but failed to impose stringent access controls. By connecting to this vulnerable socket, an attacker could leverage the containerd API to dispatch commands for spinning up new containers within the host namespace that inherited host privileges.Equipped with an unrestricted root container, the attacker could gain complete access to the host system, sidestepping the container isolation boundaries. This example underscores the critical need for access controls on intrinsic container runtime components to prevent compromise enabling unauthorized container creation and host system access s3-us-west-2.amazonaws.com, n.d.

How elevated user permissions cause Privilege Escalation Attacks:

The author emphasizes the important role of user permissions, specifically the Linux user ID (UID), in governing access to containers. The kernel oversees UIDs which determine a user's privileges to interact with containers through key system calls. Users can be granted customized container access by creating them within designated groups. However, the author stresses the need to promptly remove any unnecessary permissions associated with users. This guards against security risks like escalation attacks and unauthorized container access arising from excessive privileges. Proactively trimming unneeded permissions is advocated to reduce potential attack surface. In summary, the author highlights vigilantly managing user IDs and pruning unnecessary authorizations as critical to reinforcing container security defenses. Effectively controlling user permissions via the kernel-managed Linux UID is vital for securing container environments Shameem Ahamed, Zavarsky and Swar, 2021.

The author advises avoiding insecure secret passing in containers through environment variables prone to leaks. Instead, secrets should be passed securely via protected volumes. Read-only filesystems and avoiding unnecessary privileges also help safeguard sensitive data. These techniques limit secret

exposure across debug sessions and processes, harden containers, and mitigate risks Shetty, Jyoti. (2017).

Docker Architecture Vulnerabilities

The author focuses on vulnerabilities discovered in Docker and libcontainer pertaining to file-system isolation, a critical security aspect. Various concerns are highlighted including chroot escapes, path traversals, unauthorized filesystem access, container escalation, and privilege escalation. Importantly, patches have addressed these, with Docker 1.6.1 being a milestone. Later versions like 1.11.0 and 1.12.3 continue hardening efforts. The core issue is container processes breaching boundaries and executing as PID 0 wield substantial control over the host filesystem, enabling overwriting of binaries, code execution, and ultimately root privilege acquisition. Beyond fundamental namespaces and cgroups, the author examines Mandatory Access Control (MAC) mechanisms like the Apparmor "docker-default" policy. This whitelist-based profile restricts processes to authorized resources. However, existing policies seem to grant containers extensive access including to networks and filesystems, acting more as blacklists with limited deny rules Martin et al., 2018.

The author highlights CGroups, essential Linux features that manage and restrict resource usage for containers. Unlike namespaces, which control visibility, CGroups limit resource consumption, preventing containers from monopolizing resources. They safeguard against DoS attacks among containers and shield the host from container-induced DoS risks. CGroups also enable resource accounting, supporting usage quotas vital for emerging cloud computing models like CaaS. In essence, CGroups ensure equitable resource allocation, enhancing both inter-container protection and host security. The authors' experiments reveal limitations in using Control Groups (CGroups) for memory denial of service mitigation, as CGroups struggle to constrain malicious applications designed to intensively consume memory within allocated limits Sultan, Ahmad and Dimitriou, 2019.

The author demonstrates attack tactics in Docker clusters involving privilege escalation and deception. By hijacking the cluster leader role, the attacker gains control of communications. Leveraging this, they deploy malicious WebShell containers on each host, achieving root access. Deception techniques conceal the malicious containers and activities. The maneuvers highlight risks of privilege manipulation, stealthy persistence, and host compromise in Docker clusters through multi-stage attacks exploiting leadership roles Farshteindiker, Adi & Puzis, Rami. (2021).

Summary:

The authors reviewed explain various attack tactics exploiting Docker, along with static and dynamic analysis tools to secure containers. Some highlight how misconfigured cgroups can enable denial-of-service attacks. However, much prior research is theoretical, making it difficult to grasp for those less familiar with Docker internals and Linux concepts. Brady et al. (2020) explain the limitations of static vulnerability analysis and compare different add-on dynamic security scanning tools. They demonstrate the capabilities by testing tools like Anchore Engine and Clair on AWS infrastructure. The researchers describe the full CI/CD process to identify Docker security issues across the software development lifecycle (SDLC). Rangnau et al. (2020) present a Docker security framework that can scan images for static vulnerabilities. They develop a malicious threat prediction module using machine learning to predict and mitigate unknown threats. Clair, a static vulnerability scanning tool, serves as the foundation.

There are some drawbacks to the proposed approach, notably the reliance on static testing over dynamic analysis. Our research aims to provide more accessible insights through experimental attack simulations. Some of the researchers explained about container escapes by using the old attacker techniques and it was very theoretical. The proposed mitigation techniques by the researchers are not very efficient. By demonstrating scenarios hands-on, the concepts and tactics become more tangible. This benefits developers and smaller IT teams lacking security expertise in particular. The simulations offer practical exposure to attack techniques and key security principles to apply when building and running Docker images. Rather than just theoretical discussion, this research provides first-hand

experience with container vulnerabilities. The experiential approach makes the Docker and Linux internals concepts more understandable

3 Research Methodology

Containers provide efficient resource isolation similar to VMs, but do not automatically secure the applications running within them. Poor container configuration and hardening during development can easily introduce security risks that persist into production, like retaining debug shells. Running containers with overly privileged access is also dangerous, as a compromise of that container could exploit its privileges to access the underlying host. A common pitfall is developers adding vulnerabilities like interactive shells to help debug apps, which become glaring risks when deployed. While containers enable portability and scalability, they also expand the potential attack surface through additional components like the Docker daemon. Containers still require rigorous security practices like proper configuration, scanning, and hardening - they do not provide inherent application security out of the box. Special care must be taken when transitioning containerized workloads from development to production to remove unneeded vulnerabilities. Overall, containers provide major benefits but also risks if not properly secured across the entire lifecycle.

The research assumes an attacker has already compromised a Docker container through an insecure service. To analyze how misconfigurations enable further compromise of the Docker host, test environments will be created with containers exhibiting common misconfigs like privileged access, mounting unix sockets, running containers in host network mode and abusing the capabilities that the docker is running with. Custom exploits will be developed to target each misconfiguration for container escape and host access. Detailed evidence will be gathered on exploit outcomes. Once attack vectors for each misconfig are proven, mitigation techniques will be proposed using security tools like AppArmor, seccomp to restrict container actions and access. Additional system hardening methods like read-only containers, whitelisting, and runtime user mappings will be evaluated to further impede attacks from within misconfigured containers. Monitoring controls will also be assessed to detect early signs of compromise. The goal is to rigorously validate the risk posed by real-world Docker misconfigurations and demonstrate more robust technical controls to protect hosts from container escape, even in the presence of vulnerabilities. The research will provide evidence-based recommendations for securing both containers as well as the host systems on which they run.

Setting Up the Test Environment

- A dedicated Docker host will be created on a physical machine or VM(Kali linux or Ubuntu) to serve as the platform for testing.
- Docker container images will be built using Dockerfiles and configured with the necessary packages to create a Docker network.

Injecting Vulnerabilities

- Well-understood Docker vulnerabilities will be intentionally introduced into components like mounting unix socket, exposing docker daemon tcp socket on to the internet, abusing linux capabilities
- This mimics real-world misconfigurations and flaws that lead to unauthorized access, privilege escalation, and denial of service.

Executing Vulnerable Containers

- The vulnerable container images will be deployed and executed on the Docker host using Docker tools.

- Exploits will be developed to take advantage of the injected flaws and compromise the host system from within containers.

Security Mitigations

- Mitigation techniques using Seccomp, AppArmor, read-only containers, Linux capabilities and system calls restrictions etc. will be implemented to secure the vulnerabilities.

4 Design Specification

In this research we will be working on below two attack models

Attacker Model - I

- **Container Escape**
  The attacker has already gained access to run code inside a container. From this position, the attacker will attempt to "escape" the isolation provided by the container and gain access to the underlying host system. Some common techniques include exploiting kernel vulnerabilities, misconfigurations, or abusing container orchestration features.

Attacker Model - II

- **Docker Architecture Attacks**
  The attacker has unprivileged access to the host system that is running Docker containers. Since they cannot directly execute commands on the host, they target the Docker daemon process. By compromising the Docker daemon, they can manipulate containers and access resources on the host. This often involves exploiting vulnerabilities or misconfigurations of the Docker daemon itself.

5 Implementation

We set up a virtual machine and installed Kali Linux, followed by the installation of Docker. By pulling images from DockerHub, we intentionally introduced misconfigurations and deployed these images. This hands-on approach allows us to understand how attackers uses such misconfigurations to gain unauthorized access to the host system. We will be proposing mitigation techniques using AppArmor and Seccomp. We will be following below two attack models for exploiting docker and getting access to the host.

Attacker Model 1 (Container Escape):

- Mounting Docker sockets into containers provides access to the Docker API and ability to manipulate other containers/hosts.
- Running containers in privileged mode grants almost full host access. Could be abused to access files, processes, network, kernels, etc.
- Abusing SYS_MODULE capability allows loading arbitrary kernel modules. Can be used to disable security modules.
- Process injection techniques allow escaping into the host PID namespace to access processes.

Attacker Model 2 (Docker Daemon Attack):

- Abusing SYS_DAC_READ_SEARCH capability could allow container to read/write arbitrary files on host.
- DAC_OVERRIDE capability can be used to bypass discretionary access control checks.
- Exposing Docker daemon TCP socket allows remote access to control Docker engine.
- Targeting Portainer interfaces - often run with weak credentials or vulnerable versions.

Mitigations:

- AppArmor profiles are developed to restrict container capabilities and block specific exploits like socket mounts.
- Seccomp policies whitelist only approved syscalls, blocking things like module loading.
- Privilege dropping done by starting containers as non-root and limiting capabilities.

- Read-only volumes and filesystems used to prevent privilege escalation within containers.
- Docker daemon configured to only listen on localhost, secured with TLS auth.

6 Evaluation

In this part, we will evaluate the two attacker attacker models we discussed by running the docker container with the misconfigurations and we analyse how attackers take advantage of this misconfiguration to get access to the host and we will evaluate the mitigations by creating our own customized rules.

6.1 Experiment / Case Study 1

The Docker Unix Socket: Unveiling Potential Hazards in Containerized Environments

The Unix socket, for Docker, which can be found at /var/run/docker.sock acts as a communication link between Docker containers and the Docker daemon. This daemon manages operations related to Docker on the host system. The socket allows containers to send commands to the daemon and perform actions such as creating, starting, stopping and managing containers and images. Tools like Portainer, Sysdig or GitLab Runner often mount the Docker socket from the host into their containers when they run inside them. This setup enables these tools to seamlessly interact with the Docker daemon and carry out their intended functions.

However there are security considerations associated with this convenience. If a container that houses one of these tools becomes compromised due to vulnerabilities like command injection or remote code execution an attacker could exploit the fact that the Docker socket is mounted. By gaining access to this socket they would have control over the host system. They could execute Docker commands that directly impact both the Docker daemon and the resources of the host.

In terms an attacker with access, to the Docker socket could launch malicious containers, manipulating existing ones to expose sensitive directories on the host system or even tamper with its network settings. For example an attacker could potentially run a command like the one you shared to attach a containers volume to the file system of the host**;**

Command:
```
docker run -d -v /:/host root <image>
```

If the container's filesystem is connected to the host's root filesystem, then a potential attacker would gain direct entry to the host's files and could possibly enhance their privileges.

6.2 Experiment / Case Study 2

Security Risks and Escapes in Privileged Docker Containers

Why do Developers run Docker Containers in Privileged mode:

Running Docker containers in privileged mode is every now and then necessary for specific use cases in software development, but it should be approached cautiously because of its ability security dangers. In certain eventualities, developers might require elevated privileges to get right of entry to host devices, manage kernel settings, or perform low-stage machine operations inside the field. This is probably critical for debugging, diagnosing hardware-precise issues, or checking out positive packages that depend on direct hardware interaction. For instance, a developer working on a network tracking device would possibly need privileged mode to get entry to community interfaces for packet shooting

Investigation:

In the context of Docker containerization operating in privileged mode, a higher level of vigilance becomes essential due to the increased potential for security vulnerabilities. This mode grants extensive operational privileges to the container, effectively giving it quasi-root access to the host system's kernel.

We focus on a specific scenario where the container is equipped with the cap_sys_admin capability, enabling it to perform partition mounting activities.

The following is the command to run the docker in previleged mode

Command: sudo docker run –privileged –device=/dev/sda:/dev/sda -it ubuntu

I tested a container granted the cap_sys_admin capability, permitting partition mounting on the host. First, I assessed the container's capabilities, revealing the host's /dev/sda partition. I then formulated the precise mount command needed to leverage cap_sys_admin to mount /dev/sda onto a directory within the container.Successfully executing this mount command demonstrated the power granted by privileged capabilities to directly access and manipulate host resources.To further test the breakout potential, I gained an expanded environment by using chroot to switch the container's root directory to the mounted host partition.This firsthand privileged container breakout simulation provided tangible evidence of how capabilities like cap_sys_admin can be abused to escape isolation.

6.3 Experiment / Case Study 3

Injecting Kernel Modules from Docker Containers with cap_sys_module

Why Do Developers run Docker Containers with SYS_MODULE Capability?
The sys module capability enables Docker containers to directly interact with the host system's kernel, allowing the integration of custom kernel modules and access to low-level settings. This grants containers greater control for scenarios like testing new drivers, fine-tuning performance, and advanced debugging of software issues. For example, developers can leverage the capability to seamlessly incorporate and validate kernel modules for filesystems or networking without disrupting the host. They can also tweak kernel parameters like resource allocation and memory management to optimize application performance. Additionally, deep kernel access facilitates diagnosing complex bugs using system-level tracing and debugging tools within the isolated container. While these use cases highlight the advantages, running containers with sys module capability also introduces security risks from exposing privileged kernel-level functionality.

Investigation:
In this, we will see how break container isolation and access the Docker host system by exploiting the SYS_MODULE capability. My strategy involved several stages. First, I reviewed the container's enabled capabilities and identified SYS_MODULE, permitting kernel module insertion on the host. Next, I obtained the container's IP address for further interaction. I then developed a program to leverage the usermode helper API and open a reverse shell back to my server. This reverse shell program was compiled into a kernel module and loaded onto the Docker host using SYS_MODULE, establishing the connect-back shell. This provided me with a bash session on the host machine, from which I could access processes and directories, effectively escaping the container. In summary, by abusing the SYS_MODULE capability I was able to load an exploitative kernel module that allowed me to bypass container isolation controls and gain full access on the Docker host. This exercise demonstrated the considerable security risks introduced by SYS_MODULE against which access controls and kernel hardening is critical. It highlighted the importance of securing both the container itself as well as the underlying host kernel, since container escape depends on the integrity of kernel protections.

6.4 Experiment / Case Study 4

Process Injection:

I have been exploring the fascinating concept of process injection. This allows injecting code into the memory space of another running process to execute arbitrary shellcode using that process's context. It enables transcending traditional process boundaries in intriguing but potentially concerning ways.

In my investigation so far, I have identified two key prerequisites for successful process injection. First, the injecting process must have root privileges. Additionally, it requires the CAP_SYS_PTRACE capability to trace and debug other processes, which enables accessing their memory. I spent some time examining the PID namespace mapping between the container and the host. I found the container's PID namespace aligns closely with the host's, allowing visibility into both container and host processes. A key test case I explored was a container running with the cap_sys_ptrace capability enabled. This granted any process in that container the ability to trace other processes and read/write their memory - a very intriguing capability from a security perspective. I took care to choose stable target processes like Python and Nginx to avoid disrupting critical services during my tests. I then obtained an exploit and compatible shellcode, compiled them, and injected the shellcode into the target process. This opened up a bind shell on a chosen port, providing me access to container and host files by connecting through the host IP. Overall, this process injection journey has shown me how it is possible to break intended process boundaries through memory manipulation.

6.5 Experiment / Case Study 5

In this we can see how CAP_DAC_READ_SEARCH capability is used, to escape from a Docker container and access files on the host system that I was not supposed to. These capabilities let me bypass the normal permissions and security checks that prevent me from reading and writing files outside the container.

The first thing I did was to look for files in the container that had links to files on the host system. I found three such files and chose one of them, the "/etc/hostname" file, to test if I could change the name of the host system. However, I ran into a problem: the container did not have the "/.dockerinit" file that the exploit code needed. I solved this problem by changing the code so that it could take any file path as an argument (argv[1]) and use it as the target file. This way, I could try different files without having to change and recompile the code every time.

I executed the exploit with the command "./shocker /etc/shadow" and saw something amazing: I could see the password hash of the root user on the host system. I used a tool called "john the ripper " with a wordlist called " rockyou.txt" to crack the password and get the root user's login credentials. This meant that I could log in as the root user on the host system and do anything I wanted.

Suppose if ssh service is running on the host system, we can use the stolen credentials to log in as the root user via SSH. This was the final step of my exploit, showing how powerful and dangerous these capabilities were if they were not properly secured.

6.6 Experiment / Case Study 6

Exploiting Docker Daemon TCP Misconfiguration

In this, I am assumming attacker got access docker host, and he doesn't has root privileges. I began my exploration by carefully enumerating all open TCP ports using netstat, looking for potential points of entry. I soon identified port 2375 with the telltale Docker daemon listening behind it, wide open and unprotected. After confirming direct access to the daemon through this exposed socket, I meticulously configured the Docker client to communicate over this port by pointing its environment to tcp://<public-ip>:2375. With the vector now open, I validated the vulnerability's impact by

querying the local Docker images using my altered client, verifying unfettered container access on the host. Not yet satisfied, I further escalated my attack by launching an Ubuntu container and mounting the entire host root file system to a directory inside it. Navigating this directory exposed the host's most sensitive files now accessible from within the container. Finally, I demonstrated total compromise by chrooting into the mounted host, granting me root privilege to manipulate any file on the system from the container shell. Through careful examination, exploitation, and escalation, I systematically uncovered a severe vulnerability arising from Docker's unprotected daemon socket exposure.

6.7 Experiment / Case Study 7

What is Portainer?

Portainer serves as a user-friendly management platform designed to streamline and simplify container operations across diverse environments. Its standout feature is a centralized, intuitive dashboard that provides users a single pane of glass for creating, deploying, and managing containers. This unified control panel enables seamless container administration while abstracting away the underlying complexity of container orchestration and infrastructure. By handling these details behind the scenes, Portainer offers administrators important visibility into containers and infrastructure without the burden of directly interfacing with intricate container platforms and APIs.

Accessing host by using Portainer:

In this, I am assumming attacker got access docker host, and he doesn't has root privileges and found that Portainer which was running inside the container using weak credentials. Lets assume somehow he is able to crack the Portainer weak credentials. From there, Examining the user interface of portainer, it revealed Portainer's ability to deploy new containers with custom mounts. I decided to exploit this by deploying my own privileged container and mounting the host root file system to a directory within it. This bridged access from the container to the host OS. To maximize exposure, I selected privileged mode to permit filesystem read/write operations. After deploying the container, I accessed its shell terminal which allowed full interaction with the bound host root directory. This successfully demonstrated escalated privileges, reading/modifying any host file from within the container. My phased and thorough research methodology identified a major Portainer vulnerability - weak credentials allowed container creation that, when combined with strategic mounts, permitted complete host system access, privilege escalation, and unconstrained remote control. These findings spotlight the urgent need to implement strong Portainer authentication and carefully configure container deployments to avoid exposing risks that can lead to container escape and total host compromise.

6.8 Experiment / Case Study 8

Running Docker containers with an unprivileged user

To enhance container security, it is best practice to avoid running as the root user which can maximize damage if compromised. Docker defaults to root inside containers, allowing privilege escalation exploits. A better approach is launching containers with an unprivileged user, which reduces blast radius from vulnerabilities. First, create a non-root user on the Docker host that will be utilized inside containers. When defining containers, use the docker run -u option to specify this user rather than root. Additionally, set the USER instruction in the Dockerfile to the unprivileged account. Running as a non-root user inside containers restricts the impact of potential privilege escalation attacks if an attacker gains control. Avoiding root privileges shrinks the attack surface and limits possible damage from vulnerabilities. Specifying a least privilege unprivileged user within containers is a key technique to minimize risks and follow the principle of least privilege.

Command: sudo useradd -m <myuser>

Add the user to the "docker" group:

To enable the non-root user to communicate with the Docker daemon and run containers, add the user to the "docker" group. This grants the required permissions to execute Docker commands and manage containers without needing full root privileges. Adding unprivileged users to the docker group allows interacting with the Docker API and runtime in a restricted manner according to least privilege principles.

Command: sudo usermod -aG docker **<myuser>**

Create a Docker image with a non-root user:

If you're building your Docker image, you should create it with a non-root user inside the container. This user should have minimal privileges required for the application to run.

In your Dockerfile, add the following lines to create and switch to the non-root user:

```
# Create a non-root user with a specific user ID and group ID (optiona
RUN groupadd -r mygroup && useradd -r -g mygroup myuser

# Set the working directory and change ownership to the non-root user
WORKDIR /app
RUN chown -R myuser:mygroup /app

# Switch to the non-root user
USER myuser
```

Ensure the application runs without requiring root privileges:

When configuring the application, ensure it does not require root privileges to function. The application should be designed to operate within the limited permissions available to the non-root user defined for the container. Following least privilege principles for applications is key.

There are a few different ways you can prevent the root user from being used in your Dockerfile

```
# Disable the root user by changing the default shell to /usr/sbin/nologin
RUN usermod -s /usr/sbin/nologin root
```

Preventing Privilege Escalation Attacks

To mitigate privilege escalation risks from attackers misusing SETUID binaries, use the **--security-opt=no-new-privileges flag when** starting Docker containers. This prevents processes inside the

container from gaining more privileges than the container itself has been granted. The no-new-privileges option protects against exploits targeting SETUID binaries that could otherwise enable unauthorized privilege gains beyond the container's intended scope. It confines processes to the container's predefined privileges, limiting the blast radius from potential escalation attacks leveraging SETUID weaknesses.

Here's how you can use the flag when running a container:

```
docker run --security-opt=no-new-privileges <IMAGE-ID>
```

The --security-opt=no-new-privileges flag explicitly disables processes in the container from acquiring privileges beyond the container's predefined scope, confining them even if the host allows privilege escalation. However, this may impact applications requiring dynamic privileges at runtime, so containers should be thoroughly tested with this flag to evaluate impacts on functionality.

Limiting Docker container kernel capabilities:

Restricting kernel capabilities in Docker containers is crucial to prevent unnecessary privileges and potential exploits. By default, containers inherit some host kernel capabilities. To enhance security, explicitly specify required capabilities rather than relying on defaults. Carefully managing kernel capabilities limits container access and minimizes attack surface from privilege escalation vulnerabilities.

Here are the steps to drop and add kernel capabilities in Docker containers:

Drop all kernel capabilities:

Use the --cap-drop all option when starting a container to remove all kernel capabilities, making it more secure. This prevents the container from inheriting any privileged host capabilities. However, it may limit functionality needing elevated privileges. Explicitly dropping all capabilities enhances security by eliminating unchecked kernel access inherited from the host. But capabilities required for legitimate functionality must then be granted selectively.

```
docker run --cap-drop all <IMAGE-ID>
```

If the application in a container needs specific capabilities, use --cap-add <CAPABILITY> to grant only those required while still dropping all others for security. For example, add NET_ADMIN or SYS_PTRACE as needed by the application. --cap-add allows selectively enabling individual capabilities instead of blanket inheriting from the host. Only essential capabilities should be added to adhere to least privilege principles. Explicitly specifying required capabilities hardens containers while still permitting necessary functionality.

```
docker run --cap-drop all --cap-add <CAPABILITY> <IMAGE-ID>
```

Explicitly controlling kernel capabilities in Docker containers minimizes attack surface and reduces privilege escalation risks. Assessing the application's required capabilities and finding the right balance between security and functionality is key when configuring capability options. Enabling only essential capabilities provides security while permitting needed functions. Carefully managing capabilities rather than relying on defaults allows securing containers against potential exploits targeting excessive privileges

Launch a Docker container configured with a file system that is read-only instead of read-write.

Use the --read-only option when starting a container to make the filesystem read-only at runtime. This prevents any modifications to the container's files and folders. Enabling read-only protects the container

from malicious or unintended changes during execution. It is useful for situations where ensuring the immutable state of the container's files is critical.

```
docker run --read-only <IMAGE-ID>
```

Keep in mind that using a read-only file system may restrict certain functionalities that require write access, so make sure your application is compatible with this mode.

Specify a temporary file system for your container:

In certain situations your containerized software application might require a temporary file system for storing transient data like caches or other writable content that does not need persisting across container runs. You can configure a temporary file system utilizing the --tmpfs option followed by the mount point and maximum size settings for the temporary file system.The --tmpfs option allows mounting a temporary, memory-based file system inside a container. It can be useful for things such as caches or temp files that do not require persistence between runs.

```
docker run --read-only --tmpfs /tmp <IMAGE-ID>
```

In this provided example, the /tmp directory inside the container will be a temporary file system, which means it can be utilized for storing temporary files, however any changes made will not persist between runs of that container. By employing these options, you can determine **whether your Docker containers can store data** persistently or temporarily and prevent unintentional modifications to the file system that may result in security issues. Always take into account your application's requirements and security needs when configuring file system permissions and access for Docker containers.

Disabling inter-container communication

It is a useful technique to isolate Docker containers from each other, enhancing security and preventing unintended interactions. By default, Docker allows containers to communicate freely within the same network. Here's how you can **disable inter- container communication:**

Create a new Docker network with ICC (Inter-Container Communication) disabled:

To **create a new network with** ICC disabled, use the **--driver bridge option** along with the - o "com.docker.network.bridge.enable_icc"="false " option. Replace <NETWORK-NAME> with any desired name for the new network.

```
docker network create \
  --driver bridge \
  -o "com.docker.network.bridge.enable_icc"="false" \
  <NETWORK-NAME>
```

Run containers on the isolated network:

Now that you have the isolated network, you can run containers on it by using the --network flag. Containers running on this network will not be able to communicate with other containers on different networks or the default bridge network.

```
docker run --network <NETWORK-NAME> <IMAGE-ID>
```

By specifying the <NETWORK-NAME> with the --network flag, you ensure that the container is part of the isolated network with inter-container communication disabled.

This approach provides a higher level of isolation and security between Docker containers, ensuring that they cannot directly communicate with each other unless explicitly allowed through other means. It is particularly useful when you want to create a more isolated environment for specific containers in your Docker ecosystem.

Strengthening Docker Container Security using AppArmor

AppArmor is a Linux security module that provides mandatory access control (MAC) enforcing path-based policies on programs. Unlike the standard discretionary access control (DAC) model which uses permissions, AppArmor confines programs according to defined profiles specifying file paths the program can access. For example, an AppArmor policy may specify the Python interpreter can only read/write files under /usr/bin/ and /tmp/. AppArmor profiles are configured per program and if a program does not have a profile, it runs unconfined with standard DAC permissions.

I have developed a customized AppArmor profile with finely tuned access control rules. The first set of rules I created deny read and write access to critical host filesystem directories like /tmp and /var/tmp. This prevents containers from accessing or modifying sensitive temporary files that may contain credentials or other artifacts. I also defined network restriction rules that forbid containers from using low-level network operations like raw sockets or crafting packets. This limits the ability to make unauthorized network connections or exfiltrate data. Additionally, I restricted a number of dangerous system capabilities like sys_module, sys_ptrace, sys_admin, and sys_chroot. By denying these capabilities, I can reduce the vectors for privilege escalation from within a container. For securing container resources, themselves, I allow read-only access to container data and config directories, while permitting controlled write access to designated log directories. This prevents tampering of container assets while still enabling logging. To guard against potential memory exploitation, I blocked the ability for containers to use executable stacks. This enhances protection against stack overflow or stack smashing attacks. I also wanted to limit the ability to tamper with privileged system directories, so I denied write access to locations like /sys and /proc/sys. This prevents changing host configurations. Finally, I implemented confinement rules on Docker binaries like dockerd and docker-runc to operate with minimal filesystem permissions. This reduces their exposure to potential attacks. Together these granular rules restrict filesystem, network, capability, and access abuses to limit post-compromise impacts.

The following command will run the container with the AppArmor profile enforcing the access controls and restrictions defined in your policy. The container's processes will now be confined by AppArmor according to the customized rules, limiting the container's access to only necessary resources and functions. The --security-opt parameter is key to linking your AppArmor profile to a particular container so the access limitations are put into effect

Command: docker run –security-opt appararmor=<profile-name> <docker-image>

```
root@d4bb644067ab:/# sh
bash: /bin/sh: Permission denied
root@d4bb644067ab:/# dash
bash: /bin/dash: Permission denied
root@d4bb644067ab:/# bash
root@d4bb644067ab:/# exit
exit
root@d4bb644067ab:/# exit
exit
```

```
[root@55ff44cbda2d /]#
[root@55ff44cbda2d /]# unshare
unshare: unshare failed: Operation not permitted
[root@55ff44cbda2d /]#
[root@55ff44cbda2d /]#
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      55ff44cbda2d
10.0.0.3  attacker-ip
"/etc/hosts"
"/etc/hosts" E212: Can't open file for writing
Press ENTER or type command to continue
```

```
bash-4.4#
bash-4.4#
bash-4.4# mount
bash: /usr/bin/mount: Permission denied
bash-4.4#
bash-4.4#
```

```
bash-5.1#
bash-5.1# apt
bash: /usr/bin/apt: Permission denied
bash-5.1#
bash-5.1#
```

```
┌──(srikar㉿kali)-[/etc/apparmor.d/containers]
└─$ sudo docker run --rm -it --cap-add=SYS_MODULE --security-opt apparmor=docker-nginx-sample ubuntu
root@9ff2da6aedb1:/#
```

```
root@a822b12f32ff:/# capsh --print | grep cap_sys_admin
Current IAB: !cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!cap_net_admin,!cap_ipc_lock,!cap_ipc_owner,!cap_sys_rawio,!cap_sys_ptrace,!cap_sys_pacct,!cap_sys_admin,!cap_sys_boot,!cap_sys_nice,!cap_sys_resource,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!cap_mac_override,!cap_mac_admin,!cap_syslog,!cap_wake_alarm,!cap_block_suspend,!cap_audit_read,!cap_perfmon,!cap_bpf,!cap_checkpoint_restore
root@a822b12f32ff:/#
```

As intended, the container running with the attached AppArmor profile is not permitted to spawn shells. In the above images, we can see there are many command that helps the attcker in getting access to the host are prohibited. Even though the Docker container was launched with the sys_module capability, a custom AppArmor profile was defined to deny access to that capability. As seen in the output, the sys_module capability was disabled within the container due to the restrictive AppArmor profile, despite being granted at launch. This demonstrates how a tightened AppArmor policy can override and restrict capabilities that may be imprudently granted to a container. This demonstrates how an AppArmor policy can be applied to a Docker container using the --security-opt flag to restrict access. The customized profile was configured during template editing to control the allowed executables like restricting shell. Additionally, Linux capabilities were tuned in the template to only enable what the container requires. Now with the profile attached, the container is confined to the defined rules, exemplifying how App Armor can selectively limit binaries run and capabilities used. Overall, this shows how --security-opt connects the customized AppArmor profile to enforce the access controls specified in the template, disallowing unneeded actions like arbitrary shells while still permitting the authorized executables and capabilities.

Restricting Docker Container System Calls using seccomp

SecComp or Secure Computing is a security mechanism in the Linux kernel that allows restricting the system calls a process can make. It sets rules enforced by the kernel on which syscalls are allowed or blocked for a process, defined using Berkeley Packet Filter syntax. This limits the amount of damage a compromised process can do.

I developed a customized Seccomp profile to limit access to potentially dangerous system calls.

First, I denied specific syscalls like execve, ptrace and kill. This prevents processes in the container from executing arbitrary programs, tracing other processes, or sending kill signals. It hinders an attacker from manipulating processes if they compromise a container. Additionally, I restricted network-related syscalls including socket, bind, listen and accept. This reduces a container's ability to make unauthorized network connections or expose ports. It limits the avenues for network-based attacks. I also wanted to contain processes within their intended namespaces and environments. So I blocked syscalls like unshare and setns that allow manipulating namespaces. This prevents escalating privileges through namespace escapes.
Furthermore, I isolated powerful syscalls that could potentially enable container breakouts, like chroot, mount and pivot_root. By denying these, I make it harder for attackers to change filesystem roots and escape the container's filesystem controls. Finally, I prevented write access to sensitive directories like /etc which contain critical system configuration files. This avoids malicious modification of host configurations from within containers. Together these rules reduce the Docker attack surface by selectively denying access to system calls that could enable privilege escalation, network exploits, and unauthorized access.

```
┌──(srikar㉿kali)-[~/seccomp]
└─$ sudo docker run --rm -it --security-opt seccomp=/home/srikar/seccomp/test1.json ubuntu bash
root@??host??:/#
root@??host??:/#
root@??host??:/# chroot /bin
chroot: cannot change root directory to '/bin': Operation not permitted
root@??host??:/# mkdir test
mkdir: cannot create directory 'test': Operation not permitted
root@??host??:/#
root@??host??:/#
root@??host??:/# uname -a
uname: cannot get system name: Operation not permitted
root@??host??:/#
```

```
bash-5.1# mount
bash: /usr/bin/mount: Permission denied
bash-5.1#
bash-5.1#
```

```
bash-5.1#
bash-5.1#
bash-5.1# unshare
bash: /usr/bin/unshare: Permission denied
bash-5.1#
```

```
bash-5.1#
bash-5.1# chown
bash: /usr/bin/chown: Permission denied
bash-5.1#
```

```
bash-5.1#
bash-5.1# apt
bash: /usr/bin/apt: Permission denied
bash-5.1#
```

```
bash-5.1#
bash-5.1# chmod
bash: /usr/bin/chmod: Permission denied
bash-5.1#
```

mkdir and chroot were likely blocked because the SecComp profile does not permit those syscalls either. seeing "Operation not permitted" errors when trying to use syscalls that were prohibited in the profile verifies that our SecComp configuration is active and effective at restricting those calls. In the above images, we can see how SecComp profile blocks risky syscalls like mkdir, chmod, moury, unshare, chown, chroot that could allow privilege escalation or host tampering if abused by a malicious container. In the above image we can see apt utility which is a package manager that install the packages and binaries is getting restricted. When a container process attempts a syscall, the kernel checks if it is allowed by the active SecComp profile. If a syscall is prohibited, the kernel terminates the process with an "Operation not permitted" error before executing the call. This filtering of system calls essentially firewalls containers from making potentially compromising calls. Custom SecComp profiles can further lock down containers by denying additional syscalls specific to an application. This proactive blocking of dangerous syscalls is essential for Docker to sandbox and isolate containers.

6.9 Discussion

The experiment evaluations in this research demonstrates how attacker takes advantage of docker container and docker daemon that are running with misconfiguration inorder to get access to the host. This research demonstrates how we can limit the access permissions, how we can restrict the read, write permissions of sensitive files, how we can restrict system calls, network calls and binaries that helps attacker to get access to the host. We also demonstrated how to run the docker as non root user. In my view, attackers are finding new ways and techniques to exploit and dockers are mainly used for deployment. Deployment process involves many steps, like pulling images from the dockerhub, creating our own docker images with the necessary packages and libraries that are required for web application development. There might be a chance of having vulnerabilities in every step. Attacker make use of every chance that he got to take advantage and compromise the docker container and docker host. This research won't solve every tactic that attacker uses but it helps to serve as reference for secure development and running of docker containers. This research act as reference for hardening their container security.

7 Conclusion and Future Work

In my view, developers should adopt the security mind in their every step of development. They always focus to achieve the functionalities of product they are developing. But they should realize security is also important to access their application by users seamlessly without any interruption. Through this real world scenarios, we have seen how attackers take advantage of misconfigured containers to get access to the host. This research demonstrates mitigation techniques that hardens the container security. As I said before, there are many vulnerabilities in each and every step of the deployment. So for the future work, one can investigate, how attackers are inserting malicious layers while constructing image, and we can also investigate runc and containerd vulnerabilities and we can work how attackers are bypassing restrictions put by cgroups to consume all the resources of docker host and one can also how attackers are namespace isolations.

References

Chelladhurai, J., Chelliah, P.R. and Kumar, S.A. (2016). Securing Docker Containers from Denial of Service (DoS) Attacks. *2016 IEEE International Conference on Services Computing (SCC)*. doi:https://doi.org/10.1109/scc.2016.123.

Duarte, A. and Antunes, N. (2018). *An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability*. [online] IEEE Xplore. doi:https://doi.org/10.1109/LADC.2018.00013.

s3-us-west-2.amazonaws.com. (n.d.). *IEEE Xplore - Temporarily Unavailable*. [online] Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9031195.

Shameem Ahamed, W.S., Zavarsky, P. and Swar, B. (2021). *Security Audit of Docker Container Images in Cloud Architecture*. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICSCCC51823.2021.9478100.

Martin, A., Raponi, S., Combe, T. and Di Pietro, R. (2018). Docker ecosystem – Vulnerability Analysis. *Computer Communications*, 122, pp.30–43. doi:https://doi.org/10.1016/j.comcom.2018.03.011.

Haq, M.S., Tosun, A.Ş. and Korkmaz, T. (2022). *Security Analysis of Docker Containers for ARM Architecture*. [online] IEEE Xplore. doi:https://doi.org/10.1109/SEC54971.2022.00025.

Sultan, S., Ahmad, I. and Dimitriou, T. (2019). Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access*, 7, pp.52976–52996. doi:https://doi.org/10.1109/access.2019.2911732.

Farshteindiker, Adi & Puzis, Rami. (2021). Leadership Hijacking in Docker Swarm and Its Consequences. Entropy (Basel, Switzerland). 23. 10.3390/e23070914.

Alyas, T., Ali, S., Khan, H.U., Samad, A., Alissa, K. and Saleem, M.A. (2022). Container Performance and Vulnerability Management for Container Security Using Docker Engine. *Security and Communication Networks*, 2022, pp.1–11. doi:https://doi.org/10.1155/2022/6819002.

Shetty, Jyoti. (2017). A State-of-Art Review of Docker Container Security Issues and Solutions. American International Journal of Research in Science, Technology, Engineering & Mathematics.

Wong, A.Y., Chekole, E.G., Ochoa, M. and Zhou, J. (2021). *Threat Modeling and Security Analysis of Containers: A Survey*. [online] arXiv.org. doi:https://doi.org/10.48550/arXiv.2111.11475.