A STUDY ON ANALYSIS AND DETECTION OF CONTAINER ESCAPE VULNERABILITIES IN DOCKER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İLTER TAHA AKTOLGA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2024

Approval of the thesis:

A STUDY ON ANALYSIS AND DETECTION OF CONTAINER ESCAPE VULNERABILITIES IN DOCKER

submitted by **İLTER TAHA AKTOLGA** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Naci Emre Altun
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Mehmet Halit S. Oğuztüzün
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Pelin Angın Ülküer
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. İbrahim Körpeoğlu
Computer Engineering, Bilkent University _____

Assoc. Prof. Dr. Pelin Angın Ülküer
Computer Engineering, METU _____

Prof. Dr. Ertan Onur
Computer Engineering, METU _____

Date:22.07.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: İlter Taha Aktolga

Signature :

ABSTRACT

A STUDY ON ANALYSIS AND DETECTION OF CONTAINER ESCAPE VULNERABILITIES IN DOCKER

Aktolga, İlter Taha
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Pelin Angın Ülküer

July 2024, 88 pages

In the rapidly evolving landscape of cloud computing, containerization technologies like Docker have become essential for efficient application deployment and scalability. However, this widespread adoption has also introduced significant security challenges, particularly the threat of container escape vulnerabilities. These vulnerabilities enable malicious actors to breach the isolation of a container, potentially gaining unauthorized access to the host system or other containers, thus posing severe risks to cloud infrastructure security.

This thesis provides a comprehensive analysis of container escape vulnerabilities within Docker environments, focusing on critical Common Vulnerabilities and Exposures (CVEs). The research emphasizes the implementation of Proof of Concepts (PoCs) on ARM-based architectures to demonstrate the feasibility and implications of these vulnerabilities on alternative hardware platforms. Utilizing system call logging with Auditd and a rule-based log analysis methodology, the study offers a structured approach to detect and understand the nature of malicious activities.

Key contributions of this research include the successful adaptation of PoCs for ARM

devices, detailed system call analysis during vulnerability exploitation, and the development of rule-based detection mechanisms for identifying anomalous patterns indicative of container escapes. These findings significantly advance the field of container security by providing empirical evidence and methodological advancements aimed at enhancing defenses against container escape attacks.

The thesis concludes with a discussion on the implications of the findings, highlighting the necessity for robust security measures in containerized systems and proposing directions for future research to address the evolving threat landscape.

Keywords: Container Security, Docker, Container Escape, System Call Logging, Rule-based Anomaly Detection

ÖZ

DOCKER'DA KONTEYNER KAÇIŞ ZAFİYETLERİNİN ANALİZİ VE TESPİTİ ÜZERİNE BİR ÇALIŞMA

Aktolga, İlter Taha
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Pelin Angın Ülküer

Temmuz 2024 , 88 sayfa

Bulut bilişimin hızla gelişen dünyasında, Docker gibi konteynerleştirme teknolojileri, uygulamaların verimli dağıtımı ve ölçeklenmesi için vazgeçilmez hale gelmiştir. Ancak, bu yaygın kullanım, özellikle konteyner kaçış açıkları tehdidi ile önemli güvenlik zorluklarını da beraberinde getirmiştir. Bu açıklar, kötü niyetli aktörlerin bir konteynerin izolasyonunu ihlal ederek ana sisteme veya diğer konteynerlere yetkisiz erişim sağlamasına yol açarak bulut servislerinin güvenliği için ciddi riskler oluşturur.

Bu tez, Docker ortamlarında konteyner kaçış açıklarının kapsamlı bir analizini sunmakta ve kritik konteyner kaçış açıklarının kontrollü bir ortamda gerçeklenmesine odaklanmaktadır. Bu açıkların alternatif donanım platformlarında uygulanabilirliğini göstermek için ARM tabanlı mimari seçilmiştir. Auditd ile sistem çağrılarının kayıtları tutularak, bu kayıtlar analiz edilmiş ve seçilen güvenlik açıkları için kurallar çıkarılmıştır.

Araştırmanın ana katkıları arasında ARM cihazlar için konteyner kaçış senaryolarının başarılı bir şekilde uyarlanması, bu kaçış deneyleri sırasında detaylı sistem çağrısı

analizi ve konteyner kaçışlarını belirten şüpheli aktiviteleri tanımlamak için kural tabanlı tespit mekanizmalarının geliştirilmesi yer almaktadır. Bu bulgular, konteyner güvenliği alanında önemli ilerlemeler sağlayarak konteyner kaçış saldırılarına karşı savunmaları güçlendirmeye yönelik ampirik kanıtlar ve metodolojik gelişmeler sunmaktadır.

Tez, bulguların etkilerinin tartışılması ile sonlanmakta olup, konteynerleştirilmiş sistemlerde sağlam güvenlik önlemlerinin gerekliliğini vurgulamakta ve gelişen tehdit ortamına yönelik gelecekteki araştırmalar için öneriler sunmaktadır. Bu araştırma, konteyner güvenliği alanında hem teorik hem de pratik katkılar sağlamayı hedeflemekte ve bulut altyapılarının daha güvenli hale getirilmesinin gerekliliğini ortaya koymaktadır.

Anahtar Kelimeler: Konteyner Güvenliği, Docker, Konteyner Kaçışı, Sistem Çağrısı Kaydı, Kural Tabanlı Anomali Tespiti

To my parents, whose unwavering love and boundless support
have been the foundation of all my achievements.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Assoc. Prof. Dr. Pelin Angın Ülküer, for her invaluable guidance and support throughout this research. Her expertise and feedback were crucial in shaping this thesis.

I am deeply thankful to my family for their unwavering support and encouragement. Their love and patience have been a constant source of motivation.

Lastly, I extend my thanks to my friends and colleagues for their encouragement and to Middle East Technical University for providing the necessary resources.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xv

LIST OF FIGURES xvi

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

1.1 Containers and Virtual Machines 1

1.2 Security Challenges of Containerization 2

1.3 Contributions 3

1.4 Thesis Outline 4

2 PRELIMINARIES 5

2.1 Containerization and Docker 5

2.1.1 Introduction to Docker 5

2.1.2 Key Concepts in Docker 5

2.2 Container Escape 6

2.2.1 Platform Vulnerabilities 7
2.2.2 Kernel Vulnerabilities 7
2.2.3 Misconfigurations 8
2.3 Vagrant 8
2.4 Kernel-Level Security and Isolation 9
2.4.1 Resource Isolation 9
2.4.1.1 Kernel Namespaces 9
2.4.1.2 Control Groups (cgroups) 10
2.4.2 Protecting the Kernel 11
2.4.2.1 Capabilities and LSM's 11
2.4.2.2 AppArmor 11
2.4.2.3 SELinux 12
2.4.2.4 seccomp 12
2.5 Common Vulnerabilities and Exposures(CVEs) 13
2.6 System Call Monitoring 13
2.6.1 Techniques and Tools 13
2.6.1.1 Traditional Tracing 14
2.6.1.2 eBPF Tracing 15
2.7 Rule-Based Anomaly Detection 15
3 LITERATURE REVIEW 19
3.1 Container Security 19
3.2 Container Escape 21
4 METHODOLOGY 27

4.1 Collecting Vulnerabilities 27
4.1.1 Docker Vulnerabilities 28
4.1.2 gVisor Vulnerabilities 29
4.1.3 Kata Container and Runtime Vulnerabilities 30
4.1.4 CRI-O Vulnerabilities 31
4.1.5 LXC Vulnerabilities 32
4.1.6 containerd Vulnerabilities 33
4.1.7 runc Vulnerabilities 34
4.1.8 rkt Vulnerabilities 34
4.1.9 Singularity Vulnerabilities 35
4.2 Rationale for Focusing on Docker 36
4.3 Finding Proof of Concepts for CVE's 38
4.4 Experiment Setup 38
4.4.1 Configuring Vagrant for Apple Silicon 38
4.4.2 Auditing Tool Selection 40
4.4.3 Chosen PoCs 41
4.4.3.1 CVE-2019-5736 42
4.4.3.2 CVE-2020-15257 44
4.4.3.3 CVE-2019-14271 46
4.4.3.4 CVE-2022-0847 47
4.4.3.5 CVE-2022-0492 48
4.4.3.6 Capabilities and Namespaces 49
4.5 Container Escape Detection Rules 49

4.5.1 Analysis and Rule Crafting for CVE-2019-5736 53
4.5.2 Analysis and Rule Crafting for CVE-2020-15257 56
4.5.3 Analysis and Rule Crafting for CVE-2019-14271 62
4.5.4 Analysis and Rule Crafting for CVE-2022-0847 65
4.5.5 Analysis and Rule Crafting for CVE-2022-0492 67
4.5.6 Analysis and Rule Crafting for Capabilities 71
4.6 Proposed Design Model for Container Escape Detection System . . . 73
4.6.1 System Overview 74
4.6.2 Components of the System 75
4.6.2.1 System Call Monitoring 75
4.6.2.2 Centralized Log Management 75
4.6.2.3 Rule-Based Detection Mechanism 76
5 CONCLUSION 79
5.1 Future Work and Limitations 79
REFERENCES 81

LIST OF TABLES

TABLES

Table 4.1 Docker container escape vulnerabilities 29

Table 4.2 gVisor container escape vulnerabilities 30

Table 4.3 Kata container escape vulnerabilities 31

Table 4.4 CRI-O container escape vulnerabilities 32

Table 4.5 LXC container escape vulnerabilities 33

Table 4.6 rkt container escape vulnerabilities 35

Table 4.7 Singularity container escape vulnerabilities 36

Table 4.8 Syscalls indicative of potential container escapes 51

LIST OF FIGURES

FIGURES

Figure 1.1 Comparison of virtual machines and containers 1

Figure 2.1 Diagram illustrating types of container escape vulnerabilities . . 7

Figure 2.2 Anomaly detection architecture 16

Figure 4.1 Histogram of syscall frequencies extracted from CVE-2019-5736 audit logs 54

Figure 4.2 Histogram of syscall frequencies extracted from CVE-2020-15257 audit logs 58

Figure 4.3 Scatter Plot of syscall frequencies extracted from CVE-2020-15257 audit logs 62

Figure 4.4 Histogram of syscall frequencies extracted from CVE-2019-14271 audit logs 63

Figure 4.5 Histogram of syscall frequencies extracted from CVE-2022-0847 audit logs 66

Figure 4.6 Histogram of syscall frequencies extracted from CVE-2022-0492 audit logs 68

Figure 4.7 Proposed system architecture 74

Figure 4.8 Proposed system graph diagram 74

LIST OF ABBREVIATIONS

ABBREVIATIONS	
API	Application Programming Interface
ARM	Advanced RISC Machine
BCC	BPF Compiler Collection
BPF	Berkeley Packet Filter
CA	Certificate Authority
CAPSET	Capability Set
CNCF	Cloud Native Computing Foundation
CPE	Common Platform Enumeration
CPU	Central Processing Unit
CRI	Container Runtime Interface
CVE	Common Vulnerabilities and Exposures
ELK	Elasticsearch, Logstash, and Kibana
EUID	Effective User ID
EXECVE	Execute Program (a system call)
GID	Group ID
JSON	JavaScript Object Notation
LKM	Loadable Kernel Module
LSM	Linux Security Modules
LXC	Linux Containers
NVD	National Vulnerability Database
OCI	Open Container Initiative
OS	Operating System
PID	Process ID

SYSCALL	System Call
TPM	Trusted Platform Module
UID	User ID
UNIX	Uniplexed Information and Computing System
URL	Uniform Resource Locator
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

As cloud computing continues to expand, the reliance on containerization technologies has become more pronounced. Containers, exemplified by Docker [1] and Kubernetes [2], offer a lightweight and flexible alternative to conventional virtual machine architectures, facilitating rapid application deployment and scaling. However, this shift also introduces significant security challenges, particularly concerning container escape vulnerabilities. These vulnerabilities represent a severe threat to cloud security, as they allow malicious actors to break the isolation of a container, potentially gaining unauthorized access to the host system or other containers on the same host.

1.1 Containers and Virtual Machines

Containers provide a lightweight form of virtualization by running multiple isolated user-space instances on a shared kernel.

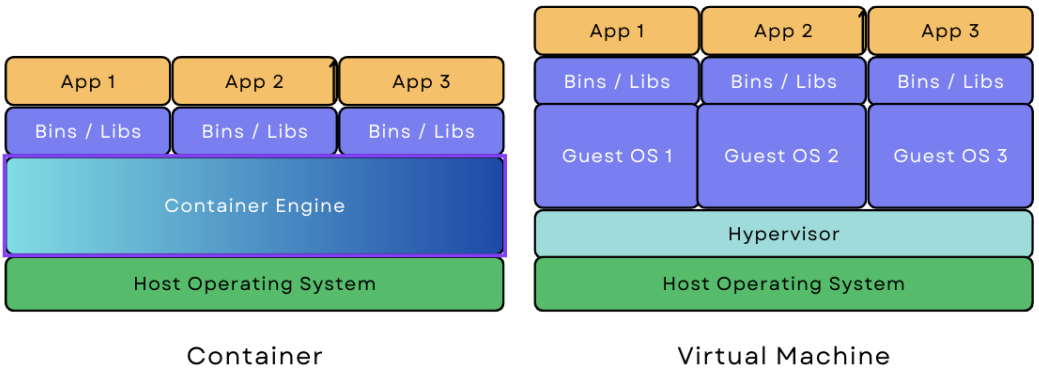


Figure 1.1: Comparison of virtual machines and containers

Unlike virtual machines, which encapsulate an entire operating system along with the application and its dependencies, containers share the host OS kernel and use features like namespaces and control groups for isolation. This results in reduced overhead, quicker startup times, and lower resource usage compared to VMs.

Figure 1.1 illustrates the architectural differences between virtual machines and containers. On the left, the container architecture shows each application running with its dependencies (Bins/Libs) on a container engine, which operates on a shared host operating system. On the right, the virtual machine architecture depicts each application running with its dependencies and a full guest operating system, managed by a hypervisor on the host operating system. This structural difference results in containers having lower overhead and faster startup times compared to virtual machines, which provide greater isolation.

Use Cases for Containers:

- **Microservices Architecture:** Containers are ideal for microservices, where applications are divided into small, loosely coupled services. Each microservice runs in its own container, enabling independent deployment and scaling.
- **DevOps and CI/CD Pipelines:** Containers streamline development and deployment by providing consistent environments across development, testing, and production stages, reducing environment-related issues.
- **Hybrid and Multi-Cloud Deployments:** Containers enhance application portability, simplifying the process of moving workloads between different cloud providers or between on-premises and cloud environments.
- **Big Data and Machine Learning Workflows:** Containers are used to package and distribute complex data processing and machine learning workflows, ensuring that all dependencies are met and the environment is reproducible.

1.2 Security Challenges of Containerization

Container escapes exploit vulnerabilities in container configurations, the underlying host kernel, or the container runtime itself. Studies indicate that attacks often lever-

age privilege escalation and memory corruption vulnerabilities [3] [4]. The shared kernel model of containers exacerbates security issues, making kernel vulnerabilities particularly dangerous [5].

To address these threats, it is essential to understand the nature and mechanics of container escape techniques, examine the existing defensive measures, and explore innovative strategies to fortify container environments against such vulnerabilities. This thesis aims to delve into these aspects, offering insights into the current state of container security and proposing methodologies to enhance defenses against container escapes in cloud computing.

1.3 Contributions

This thesis presents a comprehensive analysis of container escape vulnerabilities within Docker environments, focusing on a selection of five CVEs identified for their critical impact on security. Additionally, a proof of concept involving the testing of user namespaces and capabilities has been conducted, bringing the total number of experiments to six. The principal contributions of this work are as follows:

1. **Implementation of proof of concepts on ARM-based Machines:** We successfully implemented the PoCs for the identified CVEs on ARM-based architectures. This adaptation not only demonstrates the feasibility of container escape techniques on ARM devices but also broadens the understanding of security implications in a less commonly studied hardware environment.

2. **System Call Logging with Auditd:** During the execution of each PoC, system calls were meticulously logged using Auditd. This process was crucial for capturing the sequence and nature of system interactions triggered by each exploit, providing a detailed audit trail for subsequent analysis.

3. **Rule-Based Log Analysis for Anomaly Detection:** We developed and applied a rule-based analysis methodology to the system call logs to identify anomalous patterns associated with malicious activities. This approach allowed for a structured analysis of the logged data, facilitating the detection of potential

security breaches stemming from the exploits.

These contributions significantly advance the field of container security, particularly in ARM-based environments, by providing empirical evidence and methodological advancements for detecting and understanding container escape techniques.

1.4 Thesis Outline

The thesis is structured into five chapters, each exploring different aspects of container escape vulnerabilities in Docker environments.

Chapter 1 introduces the study, outlining the objectives, key contributions, and the overall structure of the thesis.

Chapter 2, titled 'Preliminaries', establishes the fundamental concepts and technical background necessary to understand the specifics of Docker and its security mechanisms, illustrated with relevant figures, equations, and tables.

In Chapter 3, 'Literature Review', the existing research on container security with a special focus on escape vulnerabilities is thoroughly reviewed to lay the groundwork for the analysis.

Chapter 4, 'Methodology', is the core of the thesis, describing the methods used to collect and analyze vulnerabilities specific to Docker. This chapter details the experimental setup, the selection of auditing tools, and the execution of chosen proof of concepts for specific CVEs that demonstrate escape techniques.

Finally, Chapter 5 concludes the thesis by summarizing the findings, discussing their implications, and suggesting potential areas for future research.

CHAPTER 2

PRELIMINARIES

2.1 Containerization and Docker

Containerization is a lightweight form of virtualization that allows applications to run in isolated environments called containers. Unlike traditional virtual machines, which include a full operating system instance, containers share the host system's kernel while isolating the application's processes, file systems, and network stacks. This makes containers more efficient and faster to deploy.

2.1.1 Introduction to Docker

Docker [1] is a prominent platform that simplifies the creation, deployment, and management of containers. It was introduced in 2013 by Docker Inc. and has since become the industry standard for containerization due to its ease of use and powerful features. Docker uses a client-server architecture, where the Docker client communicates with the Docker daemon (server) to build, run, and manage containers.

2.1.2 Key Concepts in Docker

- **Images:** Docker images are read-only templates that contain the necessary instructions to create a container. They are built from a set of layers, each representing a filesystem change (e.g., an installation of a package). Images are created using Dockerfiles, which specify the steps to assemble the image.
- **Containers:** Containers are runnable instances of Docker images. They encap-

sulate the application and its dependencies, ensuring consistent environments across different stages of development and deployment.

- **Dockerfile:** A Dockerfile is a script containing a series of instructions on how to build a Docker image. It defines the base image, application code, dependencies, and any necessary commands to run the application.

- **Docker Hub:** Docker Hub is a cloud-based repository where Docker users can share and distribute container images. It hosts a vast library of official images for popular software, as well as community-contributed images.

- **Docker Compose:** Docker Compose is a tool for defining and running multi-container Docker applications. Using a YAML file, users can specify the services, networks, and volumes required for the application, simplifying the orchestration of complex deployments.

2.2 Container Escape

Container escape is a security breach where an attacker exploits a vulnerability to break out of the isolated environment of a container, gaining access to the host system or other containers. This poses significant security risks, as the attacker can potentially control the host system or other applications running on it. There are several methods attackers might use to achieve container escape, broadly categorized into platform vulnerabilities, kernel vulnerabilities, and misconfigurations:

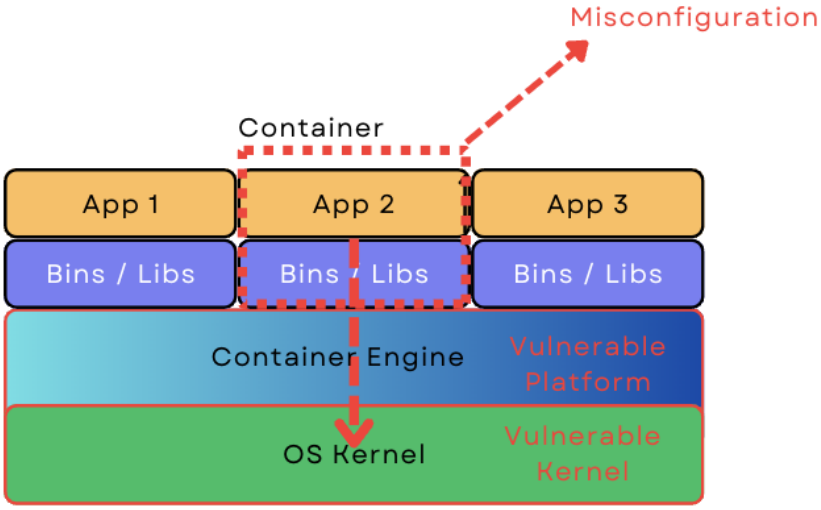


Figure 2.1: Diagram illustrating types of container escape vulnerabilities

2.2.1 Platform Vulnerabilities

These are specific to the container platform and can arise from flaws in its implementation or its components.

- **Daemon Exploits:** Exploiting vulnerabilities in the container platform's daemon (e.g., Docker daemon), which runs with root privileges and can give an attacker control over the host system.
- **API Misuse:** Using the platform's API improperly or exploiting its vulnerabilities to execute commands on the host.
- **Container Image Vulnerabilities:** Utilizing vulnerabilities in the base images or layers of a container, which can lead to an escape if the image runs with higher privileges.

2.2.2 Kernel Vulnerabilities

These vulnerabilities are present in the host system's kernel, which is shared by all containers. Exploiting such vulnerabilities can allow an attacker to break out of a

container and gain control over the host.

- **Kernel Exploits:** Exploiting vulnerabilities in the Linux kernel, such as privilege escalation bugs, which can give an attacker root access to the host system from within a container.
- **Namespace Bypass:** Exploiting flaws in the kernel's namespace implementation to escape the container's isolated environment.

2.2.3 Misconfigurations

These arise from incorrect or overly permissive configurations of containers, making it easier for an attacker to escape.

- **Excessive Privileges:** Running containers with root access or other unnecessary privileges, which can be exploited to gain control over the host.
- **Insecure Volume Mounts:** Mounting sensitive directories from the host file system into the container, exposing the host to potential manipulation.
- **Network Misconfigurations:** Configuring the container's network settings in a way that allows unauthorized access to the host or other containers.

2.3 Vagrant

Vagrant [6], a product of HashiCorp, serves as a pivotal tool in the realm of virtual environment management, offering rapid creation and seamless maintenance of portable virtual software development environments. It aims to streamline the process of configuring and launching consistent environments for developers.

At its core, Vagrant operates through the utilization of a "Vagrantfile," a Ruby-based configuration file. This file serves as a blueprint, encapsulating essential parameters such as the choice of base image, network settings, shared folders, and provisioning scripts. By standardizing these configurations, Vagrant mitigates the common chal-

lenge of disparate development environments among team members, thus enhancing collaboration.

One of Vagrant's notable features lies in its compatibility with various virtualization providers, including VirtualBox [7], VMware [8], and Hyper-V [9]. This compatibility enables developers to simulate environments closely resembling those hosting containerized applications like Docker [1] or Kubernetes [2]. Consequently, Vagrant emerges as a valuable asset for security researchers and practitioners exploring vulnerabilities and escape techniques within containerized environments.

2.4 Kernel-Level Security and Isolation

2.4.1 Resource Isolation

Resource isolation in Linux is implemented through kernel namespaces and cgroups, targeting different aspects of system management. Kernel namespaces isolate system resources, like network stacks and filesystems, creating independent operating environments for processes. In contrast, cgroups manage the allocation of physical resources, such as CPU and memory among these environments.

2.4.1.1 Kernel Namespaces

Kernel namespaces in Linux are pivotal for isolating system resources among various processes, facilitating environments that appear as separate systems but share the same kernel. This functionality is crucial for containerization, a lighter alternative to full-scale virtual machines.

Here's an overview of the types of namespaces in Linux, each designated for specific system aspects:

- **PID Namespace:** Isolates process identification numbers, allowing identical PIDs across different namespaces.
- **Network Namespace:** Creates isolated network environments including IP ad-

dresses, routing tables, and ports, enabling private networks for processes.

- **Mount Namespace:** Separates filesystem mount points among process groups, thus providing different filesystem views.

- **UTS Namespace:** Isolates system identifiers like the system name and domain name, enabling unique hostnames and domain names within each namespace.

- **IPC Namespace:** Segregates interprocess communication resources to prevent interactions between processes in different namespaces.

- **User Namespace:** Provides distinct views of system's user and group IDs to processes, allowing a process to have non-zero UID outside but appear as root inside the namespace.

These namespaces are instantiated via system calls such as `clone()` or `unshare()`, and managed using tools like `nsenter` for entering namespaces or `ip netns` for managing network namespaces.

This isolation is essential for securing and encapsulating multi-tenant environments like cloud services, ensuring that applications or services are securely and stably provided in a shared yet isolated manner.

2.4.1.2 Control Groups (cgroups)

Cgroups, short for control groups, is a Linux kernel feature that allows you to allocate, prioritize, deny, manage, and monitor system resources such as CPU time, system memory, disk and network bandwidth, or combinations of these resources among user-defined groups of tasks (processes) running on a system. This feature is essential for efficiently managing shared resources on a multi-user or multi-tenant system, such as those commonly found in cloud computing environments.

2.4.2 Protecting the Kernel

2.4.2.1 Capabilities and LSM's

In the realm of kernel security, capabilities represent a fine-grained, powerful mechanism that defines and limits the privileges of processes running on Linux systems. Unlike traditional UNIX permissions that provide a coarse level of control based on user and group IDs, capabilities break down the privileges of the superuser into smaller, manageable units. This allows system administrators to grant specific privileges to applications without exposing the entire system to potential security risks.

Linux Security Modules (LSMs) [10], such as SELinux [11] and AppArmor [12], utilize capabilities to enforce comprehensive security policies across the kernel. These modules are integral to protecting the kernel by ensuring that only authorized operations are permitted.

2.4.2.2 AppArmor

AppArmor [12] (Application Armor) is a Linux Security Module (LSM) that enhances the security of a system by enforcing mandatory access control policies. Unlike traditional discretionary access control systems where the protection mechanisms rely on the object's owner setting the permissions, AppArmor binds access control policies to programs rather than users. This means that it controls what resources each program can access, such as files, network ports, and directories, based on predefined security profiles. Each profile is tailored to restrict a program's capabilities to only those necessary for its function, thus minimizing the potential damage from exploits or software flaws. AppArmor is particularly useful in environments where multiple applications with varying security requirements are deployed, as it provides a straightforward way to compartmentalize security policies, thereby reducing the risk of a security breach spreading from one application to another

2.4.2.3 SELinux

SELinux (Security-Enhanced Linux) [11] is a powerful Linux Security Module (LSM) that enforces strict and fine-grained access control policies in the Linux operating system. Developed initially by the United States National Security Agency (NSA) to enhance the security capabilities of Linux, SELinux adds an additional layer of security by implementing mandatory access controls. This mechanism uses policies that dictate how processes and users interact with system resources, including files, applications, and network interfaces. These policies are enforced by the kernel, independent of traditional Unix permissions, thereby providing a robust defense against unauthorized access and system tampering. SELinux operates on the principle of least privilege, ensuring that both system processes and user programs operate with only the minimal set of permissions they need to function. This reduces the risk of exploitation from external attacks and limits the damage that can be caused by malicious or compromised applications.

2.4.2.4 seccomp

seccomp [13] (secure computing mode) is a security mechanism in the Linux kernel that allows a process to make a one-way transition into a restricted state where only a subset of the kernel's system calls are available. By filtering system calls and limiting the available system call interface, seccomp significantly reduces the surface area available to potential attackers. This is particularly important as many security vulnerabilities arise from the misuse or abuse of system calls. Once enabled, seccomp enforces these restrictions without the need for continuous monitoring, providing a lightweight, yet effective means of isolating applications. It is commonly used in sandboxing environments and to secure applications that handle untrusted data, such as web browsers and server daemons. This strict limitation of system call availability helps prevent malicious code execution, ensuring a higher level of system integrity and security.

2.5 Common Vulnerabilities and Exposures(CVEs)

Common Vulnerabilities and Exposures [14] is a standardized list of common identifiers for publicly known cybersecurity vulnerabilities. Each CVE entry represents a unique vulnerability, along with pertinent information such as its description, potential impact, and mitigation strategies. CVE identifiers provide a common language for communication among security professionals, facilitating the exchange of information and coordination of efforts to address vulnerabilities across various systems and platforms. By assigning CVE identifiers, security researchers, vendors, and organizations can effectively track and manage vulnerabilities, enhancing the overall security posture of digital infrastructures.

2.6 System Call Monitoring

System call monitoring is a critical component of operating system security and performance analysis. It involves the tracking and logging of system calls made by applications. System calls are the fundamental interface through which an application interacts with the operating system, and monitoring these calls can provide invaluable insights into the behavior of applications and potential security threats.

2.6.1 Techniques and Tools

Kernel hooks, tracing tools, and custom agents are three primary techniques employed in system call monitoring. Kernel hooks involve implementing hooks directly within the kernel to capture system calls at their source, providing detailed data while requiring careful implementation to avoid impacting system stability and performance. Tracing tools, like Strace [15], offer user-friendly interfaces for monitoring system calls by attaching to running processes or initiating programs under their supervision to log calls made during execution. Lastly, developing custom agents that run on the operating system to monitor system calls allows for tailored solutions to specific monitoring needs, with optimizations for minimal performance overhead and enhanced security features.

This thesis concentrates on tracing tools, with a particular focus on two distinct types: traditional tracing and eBPF [16] (extended Berkeley Packet Filter) based tracing.

2.6.1.1 Traditional Tracing

Traditional tracing is a diagnostic technique used in software development and system administration to monitor and log system calls made by a program. This type of tracing provides essential insights into how applications interact with the operating system, which is critical for both troubleshooting and optimizing software performance.

Traditional tracing tools operate by intercepting system calls at the kernel level. When a system call is made, the tracing tool stops the process, records the details of the call, and then resumes the process. This method allows developers to see the sequence of system calls in real-time, including the arguments passed and the values returned by these calls. This interception is typically achieved through a software interrupt or a similar mechanism that signals the operating system to halt the ongoing process and handle the trace.

Traditional tracing tools such as Strace [15], Truss [17], and Auditd [18] are critical in system call monitoring. Strace is primarily used on Linux and captures logs of system calls executed by processes, providing a granular view of system interactions. Truss serves a similar function on UNIX-like systems, specifically tailored for environments like Solaris [19] and FreeBSD [20]. Auditd, while predominantly an auditing tool, also logs detailed system call information as part of its security auditing capabilities.

The primary advantages of traditional tracing tools include their simplicity and ability to provide immediate, detailed insights without requiring modifications to the application's code. However, these tools come with significant drawbacks. The process of intercepting and logging every system call can considerably slow down the application, introducing a notable performance overhead. Additionally, traditional tracing methods have a limited scope, focusing solely on system calls and not providing insights into the internal logic of applications or deeper kernel operations, which can be crucial for comprehensive system analysis and debugging.

2.6.1.2 eBPF Tracing

eBPF-based tracers leverage the extended Berkeley Packet Filter framework within the Linux kernel. Unlike traditional tracers, eBPF [16] does not require stopping the process to intercept system calls. Instead, eBPF programs are loaded directly into the kernel, where they run in a sandboxed environment to monitor system calls with minimal performance impact. These programs can dynamically trace system calls, network events, and more, providing a rich data set for analysis.

eBPF-based tracing tools such as Tracee [21], Tetragon [22], and the BPF Compiler Collection (BCC) [23] represent advancements in system monitoring technology. Tracee, developed by Aqua Security, leverages eBPF to monitor system and security events with high efficiency, while Tetragon focuses on security observability and enforcement, using eBPF to implement real-time security policies based on the data it monitors. BCC provides a versatile toolkit for creating efficient kernel tracing and monitoring programs, including a range of ready-to-use tracing tools that target various aspects of system performance and security.

The advantages of eBPF-based tracers include their low overhead, which allows them to operate with minimal impact on system performance, making them ideal for production environments. Additionally, these tools are capable of collecting rich data not only on system calls but also on network traffic and internal kernel operations, offering a comprehensive view of system behavior.

However, the disadvantages include the complexity of setup and development, which requires a deep understanding of system internals and eBPF programming. This complexity can pose a barrier to entry for some users. Moreover, eBPF's reliance on newer kernel versions may restrict its use in environments that operate on older versions of the Linux kernel.

2.7 Rule-Based Anomaly Detection

Rule-based anomaly detection is a crucial technique employed in data analysis to identify deviations from expected patterns. These deviations, or anomalies, are de-

tected using a set of predefined and explicitly stated rules, which makes this approach particularly useful in environments where anomalies are predictable and can be clearly defined.

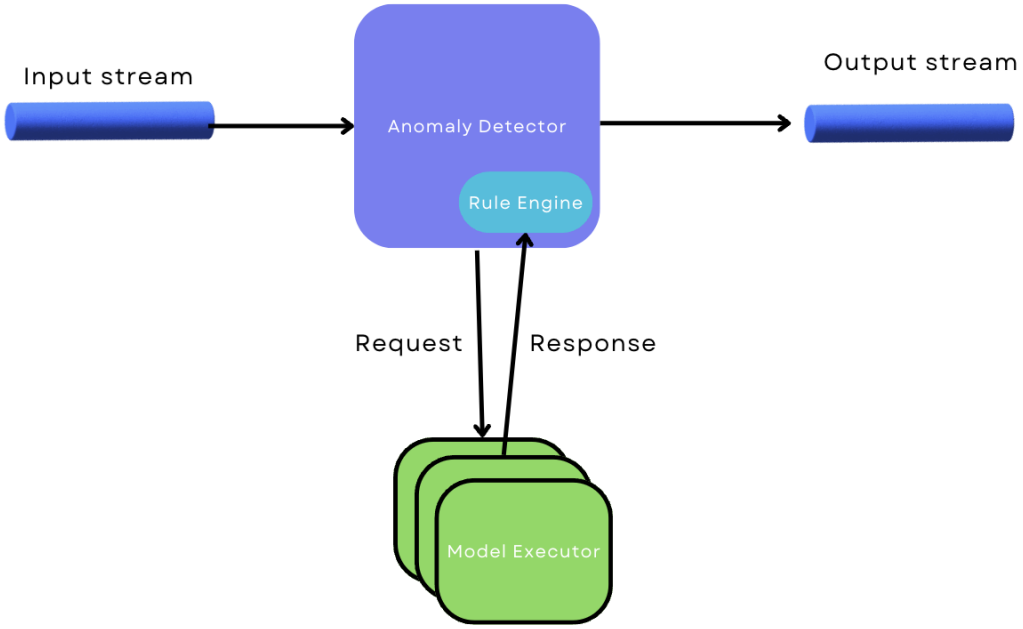


Figure 2.2: Anomaly detection architecture

The core principle of rule-based anomaly detection revolves around the application of user-defined rules that specify the boundaries of normal behavior. These rules can be binary (conditions either met or not met), or they can involve more complex logical constructs that include multiple variables and thresholds. Data points that violate these rules are flagged as anomalies. The rules are typically derived from historical data analysis, domain knowledge, or a combination of both, providing a straightforward method to spot irregularities [24].

The architecture of a rule-based anomaly detection system typically comprises several key components: the input stream, anomaly detector, rule engine, model executor, and output stream (Figure 2.2). Data enters the system through the input stream and is processed by the anomaly detector, which uses a predefined set of rules to identify deviations from normal patterns. Within the anomaly detector, the rule engine applies these rules to the incoming data and communicates with the model executor to vali-

date potential anomalies. The model executor, which can consist of multiple models working in parallel, processes the data further and provides responses back to the rule engine. These interactions, represented as requests and responses, ensure accurate anomaly detection. After processing, the data, along with any identified anomalies, is sent to the output stream for further analysis or to trigger alerts. The alert system, although not explicitly shown in the diagram, plays a critical role in notifying stakeholders or initiating automated processes to address detected anomalies. This architecture supports various application scenarios such as financial fraud detection, where it identifies unusual transaction patterns [25]; network security, by pinpointing irregular traffic that could indicate a cyber attack [26]; and health monitoring, by ensuring transparency, real-time processing, and ease of implementation while addressing challenges related to flexibility, maintenance, and the risk of false positives or negatives.

Rule-based anomaly detection offers significant advantages, including transparency, which allows users to easily understand why an event was flagged; real-time processing, providing immediate alerts and feedback; and ease of implementation, as these systems are simpler to set up and maintain compared to complex machine learning models. However, the approach also faces challenges such as limited flexibility, as it may not adapt well to new, undefined types of anomalies; maintenance overhead, due to the need for frequent updates of rules as the environment evolves; and the risk of false positives or negatives from imperfect rules, which can undermine the system's effectiveness.

Despite these challenges, rule-based anomaly detection is invaluable in contexts where anomaly conditions are clearly definable and critical, such as in regulated industries or scenarios where the cost of not catching an anomaly is particularly high. The ongoing effectiveness of these systems hinges on continuous rule evaluation and updates to adapt to new insights and changes in the environment.

CHAPTER 3

LITERATURE REVIEW

In the literature review section of the thesis, divided into "Container Security" and "Container Escape," we explore key security challenges in containerization. The first part addresses general security mechanisms and vulnerabilities within containers. The second part focuses on container escape techniques, examining how vulnerabilities can be exploited to breach container isolation. This structure helps us understand both the foundational security features and the specific threats that containers face.

3.1 Container Security

Container security remains a paramount concern in the realm of cloud computing and microservices, where the agility and scalability of container technologies are often accompanied by unique security challenges. The breadth of research in this domain addresses various aspects, from security enhancements to vulnerability assessments.

In their recent work Aktolga et al. [27] delve into the intersection of AI-driven container security and the evolving landscape of 5G networks. By analyzing the impact of machine learning on container security, they shed light on novel approaches to safeguarding containerized applications. The survey provides valuable insights for practitioners, researchers, and policymakers as they navigate the complex terrain of cloud-native security in the context of 5G and beyond.

In the realm of container security, Azab et al. [28] introduce an innovative moving target defense strategy for Linux containers. This method dynamically modifies system resources and configurations, significantly reducing the attack surface by con-

tinuously altering the environment visible to potential attackers. Such strategies are crucial for adapting to the evolving tactics of cyber threats.

Further exploring container security, Lu and Chen [29] delve into penetration testing techniques. Their work showcases methods to detect and exploit vulnerabilities within containerized environments, emphasizing the critical need for ongoing assessment protocols. This research highlights the persistent challenges in securing containers against the backdrop of rapidly advancing cyber threats.

In their examination detailed by Martin et al. [30], a comprehensive exploration of security vulnerabilities within Docker is presented. The study meticulously scrutinizes critical concerns such as container isolation failures, image security flaws, and configuration vulnerabilities. Throughout their analysis, they shed light on the susceptibility of various Docker components, ranging from Docker Engine to Docker Hub, to potential breaches. Additionally, the importance of utilizing automated tools for vulnerability assessment to effectively mitigate these security risks is emphasized.

In their analysis of container security using a dataset of 223 exploits, Lin et al. [31] develop a taxonomy categorizing threats and focus on 88 significant exploits. The study highlights the critical risk of privilege escalation exploits, despite existing security measures, which can breach container isolation and compromise the host system. Identifying a common attack pattern in 11 exploits, the team proposes a novel defense mechanism to neutralize them. The research underscores the dependency of container security on kernel robustness and stresses the need for meticulous configurations to mitigate privilege escalation attacks effectively.

Focusing on one of the most widely used container platforms, Jiang and Li [32] analyze the security aspects of Docker [1]. Their study addresses the inherent vulnerabilities in Docker containers, shedding light on both the potential security risks and the effectiveness of current mitigation strategies. This research is particularly vital for developers and security experts who utilize Docker in cloud computing environments, where security is paramount.

In addition to research papers, there is a comprehensive book on the topic of container security by Rice [33] that dives deep into the security challenges and best practices

for containerized environments. This book thoroughly covers the implementation of security measures at various stages of container deployment and management, providing strategies for robust isolation, effective management of container images, and tailored network security measures to safeguard against vulnerabilities. This resource is invaluable for those looking to enhance their understanding of security within containerized infrastructures.

Together, these studies form a comprehensive picture of the current state of container security, illustrating both the challenges and the innovative strategies being employed to enhance security in cloud-based systems.

3.2 Container Escape

Container escape vulnerabilities represent a significant security concern in the realm of containerized environments, threatening the integrity of host systems and other containers running on the same host. This literature review explores recent research findings on the sources, detection, and mitigation of these vulnerabilities.

Research by Reeves et al. [5], [34] underscores that container escape vulnerabilities often originate from three main sources: incorrect container configuration settings, inherent bugs within the Linux kernel, and deficiencies in container runtime systems. Their study specifically addresses the under-explored area of runtime vulnerabilities, demonstrating how these can significantly compromise container security.

Recent advancements in the identification of container escape incidents have led to significant developments such as PACED, a novel system that leverages a provenance-based methodology for the real-time detection of escape attempts with remarkable precision. This framework, developed by Abbas et al. [35], is particularly effective in Docker and Kubernetes environments, offering a solid basis for continuous security surveillance. Furthermore, Wu et al. [36] have implemented monitoring strategies based on process observation to regulate access to host resources, thereby preventing unauthorized actions that could facilitate container escapes.

To mitigate the threats posed by container escapes, robust security protocols are es-

sential. Guo et al. [37] propose comprehensive security practices that incorporate mechanisms such as AppArmor and Seccomp to establish stringent controls within container operations. These controls are crucial in curbing illicit activities and bolstering the overall security framework of the container landscape. Additionally, proactive measures advocated by Yang et al. [38] involve the enhancement of Docker container security through the preemptive identification and remediation of vulnerabilities in container images before their activation.

In addition to established security measures, emerging research by Song et al. [39] explores the efficacy of sequence-based system call filtering for enhancing container security. Their study, conducted at Kyungpook National University, proposes a method that analyzes patterns of system call sequences to prevent security breaches more effectively. By focusing on the sequences rather than individual calls, this approach allows for a more nuanced detection of potentially malicious activities, offering a sophisticated layer of security that adapits dynamically to the evolving landscape of container threats.

The research paper by Hassan et al. [40] addresses the security vulnerabilities associated with Docker containers, particularly focusing on reducing the attack surface through debloating techniques. The authors develop a benchmarking framework named DEBLOATBENCHC to evaluate different debloating tools—CONFINE, SPEAKER, and SLIMTOOLKIT—which utilize static, dynamic, and hybrid analysis methods, respectively. The paper highlights how these tools reduce unnecessary system calls and slim down container images to mitigate risks such as container escape, thereby enhancing container security. By comparing these tools under various metrics, the research reveals significant insights into optimizing container performance and security, establishing a foundational approach for future studies and improvements in container debloating.

Yasrab's study [41], "Mitigating Docker Security Issues," addresses key security vulnerabilities in Docker, particularly the risks tied to its shared Linux kernel architecture. The research outlines significant threats such as privileged escalation and provides targeted solutions to bolster Docker's security.

The study by Reti and Becker [42] explores the development of simulated container

escape scenarios within honeypots to enhance the detection and mitigation of such attacks in virtual environments. This research, conducted by the Intelligent Networks Research Group at the German Research Center for Artificial Intelligence in Kaiserslautern, aims to create deceptive systems that can lure attackers into revealing their methodologies without compromising actual operational containers. By integrating these simulations into honeypot architectures, the study contributes to the evolving field of cybersecurity defenses, offering a novel approach to training and strengthening security frameworks against container-based threats.

In their research, Pope et al. [43] delineates a groundbreaking approach to security in edge computing by constructing a dedicated dataset tailored for detecting container escape incidents. This dataset is designed to train machine learning models that can identify anomalous behaviors indicative of security breaches within containerized applications on edge devices. The significance of this dataset lies in its ability to provide real-time, context-aware analytics which is crucial for the dynamically changing environments of edge computing where conventional security systems often fail. The research collaboration between academic institutions and industry partners like Toshiba Europe Limited ensures that the dataset not only addresses theoretical security concerns but is also applicable in practical, operational settings. This comprehensive dataset thus serves as a vital resource for developing advanced detection systems that enhance the resilience of edge devices against sophisticated cyber-attacks.

The study by Kim et al. [4] discusses vulnerabilities such as CVE-2017-5123, CVE-2016-0728, and CVE-2014-3519, which target the kernel, noting that these issues can compromise the kernel's integrity. However, it clarifies that their research does not address mitigations for specific container runtime vulnerabilities like CVE-2019-5736, associated with the runc flaw, indicating that these types of vulnerabilities fall outside the scope of their analysis. Additionally, the paper introduces "Prof-gen," a sophisticated tool for generating system call whitelists. This tool utilizes a combination of dynamic and static binary analysis tailored to different phases of a container's lifecycle. Prof-gen tracks system calls during the initial boot-up phase of a container using dynamic analysis. Following this, it reconstructs local symbols and undertakes call graph analysis of the application binaries. Importantly, Prof-gen operates efficiently, requiring only the container image and a run command to function, streamlining the

process of securing containers against unauthorized system calls.

In their paper, Wenhao and Zheng [44] investigate the security vulnerabilities of Docker containers, outlining the types of risks involved and suggesting mitigation strategies. It provides an analysis of Docker's security architecture, identifies common security flaws, and recommends practices to enhance the security of Docker environments.

The investigation conducted by Semjonov et al. [45] examines the security benefits and challenges of using user namespaces in rootless container configurations. It highlights how running containers without root privileges through user namespaces can reduce security risks by limiting the attack surface. The paper also assesses the effectiveness of these namespaces in isolating processes and addresses potential vulnerabilities in this security model.

Flauzac et al. [46] thoroughly examine the essential security mechanisms within container technologies, focusing on thwarting container escapes. They emphasize the reinforcement of container boundaries through tools like namespace isolation, control groups, and SELinux policies. The study evaluates the effectiveness of these security measures within operational settings, underscoring the importance of diligent management and updates to counter emerging security threats effectively. It advocates for a layered security approach to uphold robust protection in containerized systems.

Examining specific vulnerabilities, the analysis by Farshteindiker and Puzis [47] sheds light on how unauthorized code execution on the host system can lead to the usurpation of leadership roles within Docker Swarm. Such a security breach poses a significant threat to the architecture's high availability and fault tolerance. In response, the authors delve into methodologies aimed at mitigating these risks, seeking to bolster Docker Swarm's security framework against invasive attacks.

A significant aspect of Abdelmassih's study [48] is addressing container escape vulnerabilities, which pose a risk in multi-tenant environments by potentially allowing attackers to gain unauthorized access to host systems. The thesis investigates various security measures and configurations that can prevent such breaches, ensuring that the deployment of container technologies aligns with the security protocols necessary

for law enforcement operations.

CHAPTER 4

METHODOLOGY

4.1 Collecting Vulnerabilities

In the initial phase of this research, a comprehensive analysis was undertaken to scrutinize the security landscape of container technologies. This endeavor embarked with an extensive survey of a diverse array of containerization platforms, including but not limited to Docker [1], containerd [49], runc [50], gVisor [51], Kata Containers [52], CRI-O [53], LXC [54], rkt [55], and Singularity [56]. Such a broad spectrum of platforms was chosen to furnish a holistic view of the prevalent security vulnerabilities, thereby setting the stage for a more concentrated investigation.

As the study progressed, a strategic decision was made to refine the scope of the investigation to focus primarily on Docker [1], containerd [49], and runc [50]. This decision was informed by the recognition of these platforms as pivotal components within the container ecosystem, underscored by their extensive adoption and the substantive body of reported vulnerabilities associated with them. The primary intent of this focused approach was to identify and analyze vulnerabilities that could potentially precipitate container escapes — situations where malicious entities circumvent isolation to execute unauthorized actions on the host system.

To systematically catalog and analyze vulnerabilities pertaining to these selected platforms, this study employed nvdtools [57], a sophisticated suite of tools designed for interacting with the National Vulnerability Database (NVD) [58]. This tooling facilitated the mapping of Common Platform Enumerations (CPEs) to their respective Common Vulnerabilities and Exposures (CVEs), thus enabling a targeted identification of vulnerabilities relevant to the container technologies under examination. The

selection of nvdtools [57] was further justified by its proven effectiveness and ongoing relevance as demonstrated in Reeves' work [34], underscoring its capability to provide up-to-date and comprehensive vulnerability analysis critical for the security assessment of these technologies. Following the graphical overview, vulnerabilities related to each platform are examined in the corresponding subsections.

4.1.1 Docker Vulnerabilities

For Docker [1], a total of twenty CVEs were initially identified, spanning a wide range of vulnerabilities. However, upon closer examination, only ten of these were found to be relevant to container escape scenarios, underscoring specific weaknesses that could potentially allow an attacker to break out of a Docker container and gain unauthorized access to the host system or other containers. The relevant CVEs are tabulated below.

Table 4.1: Docker container escape vulnerabilities

CVE ID	**Description**
CVE-2016-9962	Allows an attacker to escalate privileges within a container and potentially execute unauthorized commands on the host system.
CVE-2018-10892	Exposes a flaw in the handling of certain file descriptors, leading to a breach in container isolation.
CVE-2018-15514	Involves a misconfiguration that permits an attacker to circumvent container boundaries.
CVE-2018-15664	Enables directory traversal, allowing an attacker to access restricted areas beyond the container.
CVE-2019-13139	Highlights a problem with Docker's API, through which malicious commands could be executed, breaching container isolation.
CVE-2019-14271	Associates with the loading of a malicious Docker image that could lead to container escape.
CVE-2019-16884	Involves an incorrect handling of file paths, leading to privilege escalation and potential container escape.
CVE-2019-5736	Allows an attacker to overwrite the host Docker binary, facilitating a container to host escape.
CVE-2020-14298	Relates to flaws in container management and configuration that could be exploited to escape container boundaries.
CVE-2020-14300	Is similar to CVE-2020-14298, it relates to vulnerabilities in container management and configuration that could be exploited for container escape.

4.1.2 gVisor Vulnerabilities

For the gVisor [51], two CVEs were identified: CVE-2018-16359 and CVE-2018-19333. These vulnerabilities pertain to Google's gVisor, a sandboxing tool that provides an additional layer of isolation for running containers. gVisor is designed to

offer a more secure environment by intercepting and managing calls to the host kernel from the container. The identified CVEs highlight specific areas within gVisor's implementation that could potentially compromise its ability to securely isolate containerized applications.

Table 4.2: gVisor container escape vulnerabilities

CVE ID	**Description**
CVE-2018-16359	Describes a specific vulnerability in gVisor's handling of certain system calls, which could potentially allow an attacker to execute unauthorized actions outside the sandboxed environment.
CVE-2018-19333	Highlights a flaw in gVisor's network stack that could be exploited to bypass the sandboxing mechanism, potentially leading to unauthorized network access or data exfiltration.

4.1.3 Kata Container and Runtime Vulnerabilities

In the context of container security, CVE-2020-27151 emerges as a critical vulnerability associated with Kata Containers [52], a lightweight virtual machine that provides an isolated environment for container workloads. The vulnerability stems from inadequate validation and enforcement of guest VM exit conditions, which are crucial for maintaining the integrity of the isolation provided by the virtual machine. In a typical setup, Kata Containers operate by creating a lightweight VM for each container, thus adding an extra layer of isolation compared to traditional container runtimes. CVE-2020-27151 exploits a gap in this isolation mechanism, where an attacker, by executing specific sequences of operations within the container, could manipulate the exit conditions of the guest VM.

Table 4.3: Kata container escape vulnerabilities

CVE ID	**Description**
CVE-2020-2023	Allows network isolation mechanisms to be bypassed, potentially enabling attackers to intercept or manipulate container network traffic.
CVE-2020-2025	Permits unauthorized access to sensitive data by exploiting vulnerabilities in the storage handling mechanisms, leading to potential data leakage or compromise.
CVE-2020-2026	Pertains to inadequate input validation and sanitation in the management interface, highlighting the risk of arbitrary code execution within the Kata runtime environment.
CVE-2020-27151	Stems from a flaw in the handling of VM escape mechanisms within Kata Containers, allowing attackers to bypass security controls, escape from a container's VM, and gain unauthorized access to the host environment, compromising container isolation and host system security.

4.1.4 CRI-O Vulnerabilities

CRI-O [53] is an open-source, community-driven project under the umbrella of the Cloud Native Computing Foundation (CNCF) [59] that serves as a lightweight container runtime specifically designed for Kubernetes [2]. Its primary goal is to provide an integration pathway between OCI (Open Container Initiative) [60] compatible runtimes and the kubelet on a Kubernetes cluster, facilitating the orchestration of containers on a wide scale without additional overhead.

Table 4.4: CRI-O container escape vulnerabilities

CVE ID	**Description**
CVE-2019-14891	Identifies a security vulnerability within Docker, enabling an attacker to escape from a container and gain root privileges on the host machine.
CVE-2022-0811	Highlights a vulnerability in containerd, which allows an attacker to escape from a container and obtain root privileges on the host system.
CVE-2022-27652	Grants an attacker the ability to escape from a container and elevate privileges within the Kubernetes cluster due to a security flaw in Kubernetes.

4.1.5 LXC Vulnerabilities

LXC (Linux Containers) [54] is a lightweight virtualization technology that allows for running multiple isolated Linux systems (containers) on a single host, using shared kernel features. It's efficient and fast, leveraging Linux kernel's cgroups and namespaces for resource allocation and process isolation. This technology supports the deployment of microservices by enabling scalable and portable containerized applications. While LXC underpins the shift towards cloud-native architectures, emphasizing the significance of security in containerized environments is crucial, as vulnerabilities could lead to potential security breaches.

Table 4.5: LXC container escape vulnerabilities

CVE ID	Description
CVE-2019-5736	Affects runc, a container runtime used by Docker before version 18.09.2. It allows attackers with write access to a container to overwrite the host runc binary and potentially gain root privileges on the host system.
CVE-2017-5985	Allows local users to create network interfaces on the host. It doesn't grant container escape but could be leveraged during an attack.
CVE-2018-6556	Allows users to write to certain files outside the container. Similar to CVE-2017-5985, it doesn't directly enable escape but could be a part of a larger exploit.

4.1.6 containerd Vulnerabilities

Containerd [49] is an industry-standard core container runtime, designed to be available as a daemon for Linux and Windows. It focuses on simplicity, robustness, and portability. Containerd is part of the Cloud Native Computing Foundation (CNCF) [59] and is used by Docker [1], Kubernetes [2], and other container-based systems to manage the full container lifecycle of its host system. From image transfer and storage to container execution and supervision, containerd provides the essential capabilities to run containers on a wide variety of platforms, emphasizing security and efficiency.

CVE-2020-15257, titled "Abstract Socket Madness," is a notable vulnerability that was discovered in containerd versions before 1.3.9 and 1.4.3. This vulnerability exposes a significant security flaw within the containerd runtime. Specifically, it affects the containerd-shim API, accessible over an abstract Unix domain socket, which does not properly enforce access control. Consequently, this oversight allows any user on the host system, regardless of privileges, to connect to this socket.

4.1.7 runc Vulnerabilities

runc is a lightweight, portable container runtime that sits at the heart of most containerization platforms, including Docker and Kubernetes. It is an implementation of the Open Container Initiative (OCI) specifications, which define standards for container formats and runtimes, ensuring interoperability and consistency across different container ecosystems. runc is designed to spawn and run containers according to the OCI container runtime specification. It directly interacts with the operating system's kernel to create the environments that isolate the containers from one another and from the host system.

CVE-2019-19921 is a vulnerability identified within runc that highlights a critical security risk, affecting versions before 1.0.0-rc9. The vulnerability stems from how runc handles file descriptors when executing the runc init command in a new namespace. Specifically, it allows an attacker with basic user privileges within a container to exploit this flaw by overwriting the runc binary and gaining root-level control over the host system.

4.1.8 rkt Vulnerabilities

rkt [55], initially developed by CoreOS, emerged as an innovative container runtime project, aligning with the principles of security, simplicity, and modularity within the dynamic sphere of container technologies. It introduced a novel architecture that encapsulated applications within pods—a concept that resonates with the Kubernetes [2] pod model—thereby facilitating a cohesive environment where multiple containers could operate in unison, sharing the same lifecycle and network space. With a pronounced emphasis on security, rkt was among the first to implement rigorous security mechanisms, including integration with TPM (Trusted Platform Module) for hardware-level security and the utilization of seccomp [13] for enhanced process isolation.

Table 4.6: rkt container escape vulnerabilities

CVE ID	**Description**
CVE-2019-10144	Authorizes an attacker to gain root privileges on the host system through an authentication bypass vulnerability in the enter command of rkt.
CVE-2019-10145	Allows an attacker to escape the container and gain root privileges on the host system due to a lack of seccomp filtering in the enter command of rkt, a container runtime.
CVE-2019-10147	Enables an attacker to take over all resources of the container and affect other containers or the host system because of a lack of cgroups limits.

The analytical lens then narrowed further, centering specifically on Docker. This was motivated by Docker's prominence within the containerization landscape and the extensive documentation of its vulnerabilities. A meticulous examination of these vulnerabilities was undertaken, with a particular emphasis on those that could enable container escapes.

4.1.9 Singularity Vulnerabilities

Singularity[56] is a container platform tailored for high-performance computing (HPC) environments, making it ideal for scientific research and data analytics. It packages containers as single files, ensuring portability and consistent operation across different systems. Unlike Docker [1], Singularity allows users to run containers without root privileges, enhancing security. It integrates seamlessly with existing HPC workloads, enabling researchers to use container technology without reconfiguring their systems. Singularity encapsulates applications and their dependencies, facilitating easy transport and execution across various environments, ensuring reproducibility and ease of deployment crucial for scientific research.

Table 4.7: Singularity container escape vulnerabilities

CVE ID	Description
CVE-2021-29136	Allows attackers to overwrite arbitrary host paths via a crafted image causing symlink traversal when “umoci unpack” or “umoci raw unpack” is used. Related to Open Container Initiative umoci before 0.4.7.
CVE-2019-11328	Allow editing of files within /run/singularity/instances/sing/<user>/<instance> due to insecure permissions in Singularity 3.1.0 to 3.2.0-rc2, potentially resulting in privilege escalation on the host through manipulation of the starter-suid program when instances are joined.

4.2 Rationale for Focusing on Docker

In our thesis, we concentrated our investigation on security vulnerabilities specific to Docker [1], deliberately choosing to narrow our focus to this platform amidst the myriad of container technologies available. This decision was guided by several key considerations, distinguishing Docker not only as a pioneer in the containerization movement but also as a critical component of the broader container ecosystem. Below, we elucidate the rationale behind our targeted approach, particularly emphasizing why Docker was selected over other container platforms.

Pioneering Technology: Docker’s role as the pioneering container technology has been instrumental in shaping the landscape of containerization. By focusing on Docker, our research taps into the core principles of container security, laying a foundational understanding that is applicable across container technologies.

Widespread Adoption and Impact: Docker’s widespread adoption across industries makes it a significant subject of study. Its impact on the development and deployment practices of organizations worldwide underscores the importance of addressing its security vulnerabilities.

Simplicity and Educational Value: Docker provides a straightforward and accessible approach to containerization, making it an ideal platform for dissecting and understanding container security. This simplicity is invaluable for educational purposes, allowing for a clearer presentation of security concepts and practices.

Security Model and Implications: Docker's security model, encompassing aspects from image security to container runtime isolation, offers a comprehensive framework for studying container security. This model serves as a basis for understanding security in more complex systems.

Direct Influence on Application Security: The direct influence of Docker on application security, through its handling of container images and execution environments, presents a focused area for studying how container vulnerabilities can affect application integrity and security.

Foundation for Advanced Technologies: Understanding Docker's security vulnerabilities provides a foundation for exploring security in more complex container orchestration platforms. Docker serves as a building block for these technologies, and insights gained from Docker security directly apply to broader contexts.

Industry Relevance: Compared to other container platforms, Docker is often the first container solution adopted by organizations. This prevalence makes Docker a critical subject for understanding and mitigating real-world security issues that organizations face.

Community and Ecosystem Support: Docker has a vast and active community that continuously contributes to its development and security enhancements. This strong ecosystem support provides a wealth of resources, tools, and best practices that can be leveraged in our research.

Relevance to Security Best Practices: Focusing on Docker enables a detailed exploration of security best practices at the container level. These practices are crucial for securing Docker environments and offer foundational knowledge for securing containers in any environment.

In sum, our decision to concentrate on Docker stems from its foundational role in

the container ecosystem, its widespread adoption, and its direct relevance to core security principles. This focus allows us to delve deeply into the nuances of container security, offering insights that are both specific to Docker and broadly applicable to the containerization domain.

4.3 Finding Proof of Concepts for CVE's

In our research focused on identifying proof of concept (PoC) for vulnerabilities specific to Docker, we utilized a specialized tool named trickest/cve [61] found on GitHub [62]. While exploring other resources, we also examined the Metarget [63] project but were unable to leverage it due to our requirement for ARM-based Docker versions. To supplement our search, we conducted extensive searches on GitHub and various search engines using the pattern "<CVE name> poc" to locate accessible PoCs. This approach was critical in gathering the necessary data for our thesis, emphasizing our proactive and comprehensive strategy in identifying and analyzing Docker vulnerabilities.

Within our investigative framework for Docker vulnerabilities, we meticulously selected three PoCs for our experimental setup, specifically targeting CVE-2019-5736 [64], CVE-2020-15257 [65], and CVE-2019-14271 [66]. To broaden our research spectrum, we further incorporated two kernel-based container escape vulnerabilities, CVE-2022-0847 [67] and CVE-2022-0492 [68], into our analysis. The details of these selected Proof of Concepts (PoCs) will be elaborated in the experiment setup section.

4.4 Experiment Setup

4.4.1 Configuring Vagrant for Apple Silicon

At the outset of our experimental setup, it's crucial to note that all tests were conducted on a 2021 M1 MacBook Pro.

The very nature of the vulnerabilities we were testing—aimed at gaining unauthorized

access to the host system—underscored the critical importance of conducting these tests on a platform that was isolated from our main machine. Since the essence of container escape techniques involves breaching the boundaries set by containerization to reach the host operating system, performing such tests on our primary systems would inherently pose a significant risk. The potential for these vulnerabilities to not just interfere with Docker's functionality but also to compromise the host machine's security and integrity made it paramount to ensure a strict separation between the testing environments and our main infrastructure.

Using Vagrant [6] to manage VMs offered a robust solution to this challenge. Each VM served as a standalone testing ground, fully detached from the host system's critical components and data. This setup allowed us to simulate attack scenarios in a controlled and safe manner, without the risk of accidentally exposing our main system to the very threats we were investigating. The isolation provided by Vagrant's VMs ensured that any potential breach or malfunction induced by the PoCs remained confined within the virtual environment, thus preserving the integrity and security of our host system.

We opted to utilize Vagrant due to its capacity to streamline the creation and management of virtualized environments, a necessity given the distinct Docker version requirements for each CVE under investigation. Vagrant enables us to efficiently establish isolated environments tailored to the specific conditions each vulnerability demands, without inducing permanent modifications to the host system.

Due to the ARM-based nature of the M1 MacBook Pro, it was essential for the operating systems we intended to install via Vagrant to also be ARM-compatible. This compatibility is crucial for ensuring that the virtual environments run efficiently and without the complexities associated with cross-architecture emulation. Consequently, we embarked on a search for ARM-compatible Vagrant boxes, which are pre-packaged development environments. These boxes are essentially virtual machine templates designed for use with Vagrant, offering a convenient and efficient method to deploy repeatable, predictable, and portable work environments.

We meticulously adhered to a detailed guide authored by Sébastien Bailliez [69], which delineated a comprehensive methodology for installing Vagrant in conjunction

with VMWare Fusion 13 Player [70], the latter of which includes a preview version specifically designed for Apple Silicon. This guide was instrumental in equipping us with the necessary tools and knowledge to set up a virtualized environment optimized for the ARM-based architecture of the M1 MacBook Pro. By integrating Vagrant with VMWare Fusion 13 Player, we were able to leverage the full potential of Apple Silicon, ensuring our experimental setup was both efficient and aligned with the latest technological advancements. This strategic approach facilitated a seamless installation process, enabling us to focus on the precise emulation of Docker vulnerability testing conditions in a controlled and reproducible manner.

4.4.2 Auditing Tool Selection

In the evaluation of proof of concepts for container vulnerabilities, the choice of an appropriate auditing tool is crucial. Our study considered various available tools, including Auditd [18], Tracee [21], Tetragon [22], strace [15], and other eBPF-based tracers. Each tool has its strengths and potential drawbacks depending on the context of use. Here, we provide a comparative analysis and explain our rationale for opting against the use of eBPF-based tracers, primarily due to concerns about backward compatibility.

Auditd [18] is part of the Linux Auditing System, renowned for its stability and deep integration with the Linux kernel. It excels in logging detailed system calls and can be configured to track a broad spectrum of system activities. Its longstanding presence in Linux systems makes it a reliable choice for environments with diverse kernel versions.

Tracee [21], developed by Aqua Security, uses eBPF technology to capture events at the kernel level, providing insights into the system's behavior with minimal performance overhead. Although Tracee is effective for real-time analysis and can efficiently trace system calls and security events, it requires a relatively recent Linux kernel that supports eBPF.

Tetragon [22], similar to Tracee, is another eBPF-based security observability tool. It offers advanced capabilities for observing and intercepting system calls and events,

crucial for security-focused analyses. Like Tracee, Tetragon demands a newer kernel version with full eBPF support.

strace [15] is a tracing utility used to monitor the system calls made by a process and the signals it receives. Strace is straightforward to use and does not require any specific kernel features beyond what is available in very old Linux versions. However, it can introduce significant performance overhead and is less suited for highly granular, security-focused logging.

eBPF-based Tracers (Tracee, Tetragon) provide advanced tracing capabilities that can be highly advantageous in a modern environment. They allow for dynamic tracing, which is less disruptive and can be targeted very precisely. However, the major limitation of eBPF-based solutions is their requirement for newer kernel versions (4.x and above), which support eBPF functionalities.

Our decision to use auditd over eBPF-based tracers was influenced by the need for backward compatibility. Many of the host machines available for our testing run on older versions of the Linux kernel that do not support eBPF. This compatibility issue is significant as it limits the generalizability of our findings across different systems, especially in environments where upgrading the kernel is not feasible due to stability or policy constraints.

By choosing Auditd, we ensured that our testing environment would be applicable and relevant to a wider array of real-world systems, including those not on the cutting edge of kernel developments. This choice supports the objective of our research to provide insights that are both practical and broadly applicable, particularly in enterprise settings where legacy systems are common.

4.4.3 Chosen PoCs

In this section, we will examine the PoCs that we have selected and prepared an appropriate environment to test. To facilitate a thorough and controlled exploration of these vulnerabilities, we established a virtualized testing environment using Vagrant. Within this virtual environment, while executing the PoCs, we simultaneously employed the Auditd tool, a powerful utility for monitoring system activities. Auditd

was configured to track and log various system calls as well as file read and write operations. This dual approach allowed us to not only execute the PoCs effectively but also to gain detailed insights into how each exploit interacts with the system at a granular level.

4.4.3.1 CVE-2019-5736

Discovered in February 2019, CVE-2019-5736 [64] targets a significant vulnerability in runc [50], the underlying container runtime beneath Docker and other platforms. This vulnerability allows malicious containers to overwrite the host system's runc binary with arbitrary code, which is then executed typically with root privileges. Such an attack is possible because runc may be invoked to execute commands within the context of an existing container. If an attacker successfully replaces the host's runc binary with a malicious version, any subsequent invocation of runc could lead to unauthorized privilege escalation and control over host system resources, thereby breaching the isolation boundary that containers are supposed to maintain.

The core of the exploit is a race condition; the runc process can modify itself between checking whether its binary on the host is write-protected and executing a command. By repeatedly attempting to overwrite the binary, an attacker can eventually succeed during this window of vulnerability.

runc operates by creating and managing namespaces and control groups (cgroups) that isolate the container's processes from the host system. This vulnerability is triggered when runc executes a command requiring access to a file descriptor linked to the runc binary itself. If a malicious container accesses the binary and executes a command concurrently with runc, it can alter the execution flow.

This vulnerability is inherently linked to how Linux manages file descriptors passed to running processes. runc did not adequately secure its own file descriptors, particularly those associated with its executable binary. When runc executes a container process, it opens its own binary to reference certain execution parameters. If a malicious process overwrites or modifies the binary at this critical moment, it can dictate the actions of runc.

This security flaw was particularly concerning because it affected all Docker versions prior to 18.09.2 and all versions of runc before 1.0-rc6, impacting a wide array of systems running Linux containers. Immediate mitigation measures included upgrading to patched versions of Docker and runc, and implementing stricter access controls on the host systems to limit the ability of containers to modify executable binaries.

To conduct a practical demonstration of CVE-2019-5736 [64], we established a testing environment using Vagrant [6] to simulate the conditions under which the vulnerability could be exploited. Our setup involved the following steps to create a controlled, reproducible environment for our PoC.

Firstly, we utilized the "uwbbi/bionic-arm64" Vagrant box as the foundation for our virtual environment. This choice provided a stable and well-supported base operating system, allowing us to closely mirror the configurations commonly found in production environments where Docker is deployed.

Next, we installed Docker on our Vagrant environment. We specifically downloaded and installed the docker−ce_18.06.0 arm package from Docker's stable releases [71]. This particular version was chosen to align with the pre-patched versions susceptible to CVE-2019-5736 [64], thereby ensuring the authenticity of our vulnerability testing.

Following the Docker installation, we proceeded to set up the vulnerability exploit. For this purpose, we referred to a detailed example provided by Palo Alto Networks on their blog [72], which included a link to a GitHub repository. [73]. We cloned this repository, which contained all the necessary scripts and configuration files to execute the exploit.

The repository included two key components critical for demonstrating the exploit: an Executable container and a Malicious container. Running these containers in our environment allowed us to effectively reproduce the conditions of CVE-2019-5736 [64]. When executed, the Malicious container successfully altered the runc binary as per the exploit's design, confirming the vulnerability's presence and demonstrating how an attacker could escalate privileges and control over the host system by manipulating the container runtime.

We configured auditctl to log every syscall for both major architectures.Comprehensive

syscall auditing enhances our ability to detect anomalies that could be missed with more selective logging. Anomalies in syscall patterns often indicate exploits or misconfigurations that are exploiting the system, making them important signals in security research.

4.4.3.2 CVE-2020-15257

CVE-2020-15257 [65], identified in November 2020, is a significant security vulnerability found within containerd [49], a widely used container runtime environment. This vulnerability exposes fundamental issues related to the handling of network namespaces and inter-process communication (IPC) mechanisms in containerized environments.

The core of CVE-2020-15257 lies in the containerd-shim API service, which is responsible for managing the lifecycle of containers. This vulnerability stems from an insecure handling of IPC mechanisms by the containerd-shim service, which operates as a daemon. In typical configurations, containerd-shim does not require authentication for connections, assuming that access to its API is sufficiently restricted by container and network namespaces.

However, the vulnerability arises when host processes or other containers can access the API endpoint of containerd-shim. Since containerd-shim exposes a Unix socket to facilitate communications, any process with access to this socket can send commands to containerd-shim, leading to unauthorized actions. The issue is particularly critical because it can be exploited from any container in the same network namespace, bypassing intended isolation mechanisms.

Exploiting CVE-2020-15257 involves a malicious container within the same network namespace as the containerd-shim communicating over the Unix socket to execute unauthorized commands. The exploit is based on sending crafted messages to the containerd-shim API, which then executes these commands with host-level privileges. This could allow attackers to manipulate other containers, access sensitive data, or disrupt system operations, effectively escaping the container and gaining control over the host.

To execute the exploit, an attacker first needs to understand the communication protocol used by containerd-shim. They then create malicious payloads that mimic legitimate API calls but are designed to perform unauthorized operations. By sending these payloads to the containerd-shim socket, the attacker can initiate commands that the shim executes on the host, leading to potential system compromise.

The experiment begins with the setup of Docker on an Ubuntu Bionic system. Docker is a popular containerization technology that allows for the encapsulation and easy deployment of applications. The specific version used was Docker 19.03.13, chosen for its compatibility with the exploit conditions at the time of testing.

In the experiment, the initial step involved the configuration of a virtual environment using the `bento/ubuntu-20.04-arm64` Vagrant box. This setup was essential to ensure that the virtual machine was tailored to the specific needs of the experiment. After successfully establishing the virtual environment, the Docker software was installed, specifically the version `docker-ce_19.03.13~3-0`, which was selected for its compatibility with the Ubuntu Bionic operating system on an ARM64 architecture. Following the Docker installation, the environment for the CVE further required the setup of the `containerd` version `1.3.7-1` and the Go programming language, version `1.19.3`. These installations ensured a robust development environment that supports extended functionality and efficiency for container management and software development.

Subsequently, we cloned the GitHub repository by nccgroup [74]. Following the repository setup, we executed commands to build and run the application. Specifically, we ran `go build` to compile the project, followed by `docker build -t abstractshimmer .` to create a Docker image named `abstractshimmer`. Finally, the application was launched using `docker run -rm -d -network host abstractshimmer | xargs docker logs -f`, which started the Docker container in detached mode with network settings configured to use the host's network, and the logs were continuously output to the console.

In line with previous vulnerability assessments, where Auditd was instrumental in gathering system calls, this research follows suit. Utilizing Auditd, we collect system call data, continuing the tradition of fortifying system security through meticulous

auditing measures.

4.4.3.3 CVE-2019-14271

CVE-2019-14271 [66] is a critical vulnerability identified in Docker Engine that compromises the security of containers through the misuse of Loadable Kernel Modules (LKMs). Unveiled in July 2019, this vulnerability underscores the risks associated with kernel-level integrations in container environments.

The vulnerability specifically stems from the implementation of the docker cp command, which allows files to be copied between a Docker container and the host system. Under normal operations, this functionality is intended to provide flexibility in managing container data. However, CVE-2019-14271 [66] exploits this feature through a crafted image that contains a malicious LKM.

When the Docker Engine executes the docker cp command to interact with this image, the malicious LKM is automatically loaded into the host's kernel, bypassing the standard security mechanisms intended to isolate the host from direct container access. The vulnerability exploits the insufficient validation checks during the file copy operation, particularly when the files have specific attributes that are typically associated with kernel modules.

Exploitation of CVE-2019-14271 involves creating a Docker image that includes a malicious LKM disguised as a regular file. When an unsuspecting user executes the docker cp command to transfer files from the container to the host, the malicious module is triggered. This module can execute with kernel-level privileges, providing broad scope for compromising the host system, including manipulating system operations, accessing sensitive information, or creating a backdoor for persistent access.

The exploit is particularly dangerous because it does not require significant user interaction and can be triggered by a single command that is commonly used in Docker environments. This makes it a potent tool for attackers looking to escalate privileges or move laterally within an enterprise network.

We set up our testing infrastructure using Vagrant, running an Ubuntu Bionic system

tailored for ARM64 architecture. The Docker components installed included containerd.io 1.2.2-3, docker-ce-cli 19.03.0, and docker-ce 19.03.0, which were crucial for managing Docker containers on this platform. Subsequently, we utilized an exploit from the GitHub repository at [75].

4.4.3.4 CVE-2022-0847

CVE-2022-0847 [**?**], colloquially known as the "Dirty Pipe" vulnerability, was disclosed in March 2022 and is a critical security flaw affecting Linux kernels from version 5.8 onwards. This vulnerability highlights significant issues in the Linux kernel's handling of pipe buffers, which can lead to unauthorized data modification, privilege escalation, and potential system compromise. The root cause of CVE-2022-0847 lies in the improper management of pipe buffers in the Linux kernel. Pipes in Linux are used for inter-process communication, allowing data to flow from one process to another. Normally, these buffers are managed securely within the kernel to prevent unauthorized access or modification. However, this vulnerability arises due to a flaw in the way Linux clears (or fails to clear) these buffers when certain conditions are met.

Specifically, the issue occurs when a process writes data to a pipe, decides not to send the data by truncating the pipe, and then another process writes new data to the same buffer. The flaw allows unprivileged users to exploit this mechanism to overwrite read-only files and append arbitrary data to files they have access to read, thus "polluting" the pipe buffer — hence the name "Dirty Pipe."

Exploiting CVE-2022-0847 [67] involves a relatively simple but powerful technique where an attacker leverages the vulnerability to inject or modify data within files that should be immutable or protected. For example, an attacker could append malicious code to system scripts or configuration files to escalate privileges or disrupt system operations.

The exploitation process typically involves three steps. A process opens a pipe and writes data intended to be discarded. The process truncates the pipe, but the data remains in the buffer. Before the buffer is cleared by the kernel, another process writes

new data that overlaps with the old data, leading to file corruption or unauthorized modifications.

We installed Ubuntu inside a Vagrant environment using the bento/ubuntu-22.04-arm64 Vagrant box [76]. To meet the requirements of our tests; we needed to use a specific vulnerable kernel version, 5.9.5-050905-generic, and therefore we downgraded the kernel using the script available at [77]. Following the kernel downgrade, we conducted our tests using scripts from the repository [78].

4.4.3.5 CVE-2022-0492

CVE-2022-0492 [**?**], identified in February 2022, is a critical security flaw affecting the Linux kernel's handling of cgroup v2, an essential mechanism for resource management in modern Linux environments, including containers. This vulnerability allows for privilege escalation and can facilitate container escapes, posing a significant risk to containerized infrastructure.

The vulnerability stems from improper validation and enforcement of permissions in the cgroup v2 subsystem of the Linux kernel. cgroups (control groups) are used to organize and manage resources, such as CPU time, memory, network bandwidth, and more, among user-defined groups of tasks. Specifically, CVE-2022-0492 exploits the ability to manipulate cgroup namespaces in ways that were not intended by allowing users with even the most basic privileges to adjust resource controls that affect higher-level system processes.

This security gap is primarily introduced through the Delegate flag, which is meant to allow non-privileged containers to manage their own cgroups within designated limits. However, due to a flaw in the kernel's cgroup setup, the flag allows unprivileged processes to escalate their privileges to that of the root within their namespace.

Exploiting CVE-2022-0492 involves an unprivileged process within a container manipulating the cgroup configuration to gain unauthorized access or control over system resources. The exploit typically follows these steps:

An attacker creates or joins a cgroup and sets the Delegate flag. The attacker exploits

the flawed permission handling to extend control over other cgroup parameters beyond the intended scope. By modifying these parameters, the attacker can exhaust resources or launch denial-of-service attacks on the host or other containers, or potentially execute commands with elevated privileges.

We used the script from the repository at [79] as a proof of concept for testing CVE-2022-0492.

4.4.3.6 Capabilities and Namespaces

In the study of container security, the manipulation and logging of capabilities and user namespaces play crucial roles in understanding and enforcing security boundaries. To comprehensively explore these elements, we designed and conducted a series of experiments using a proof of concept (PoC) container. This section details the experimental procedures undertaken to test the behavior of containers with and without the use of user namespaces, focusing on setting specific capabilities and monitoring their effects through system logs.

4.5 Container Escape Detection Rules

In the realm of security monitoring, the development of rules for analyzing logs, particularly those collected by Auditd [18], is integral to identifying and mitigating potential security threats. These rules are crafted to detect a range of malicious activities, such as unauthorized access attempts to sensitive system areas and privilege escalations. They also monitor modifications to critical system files and unusual spikes in resource usage, which could signify potential security breaches or denial-of-service attacks.

Building on this foundational approach, our experimental section was specifically designed to further our understanding and mitigation strategies against container escapes. We executed scripts for predetermined CVEs, while simultaneously collecting logs with Auditd. These logs were then analyzed using a rule-based method, employing the predefined rules discussed earlier. This strategic combination of rule de-

velopment and practical testing aimed to enhance our detection capabilities, enabling us to identify indicators of container escapes effectively. Through this methodology, we not only reinforced the security framework but also provided a robust response mechanism to potential security vulnerabilities.

To enhance security and detect potential container escape attempts in Docker environments, it is crucial to implement robust monitoring mechanisms. This involves developing specific audit rules that focus on high-risk activities typically associated with container breaches. For instance, monitoring the use of powerful capabilities like CAP_SYS_ADMIN and CAP_NET_ADMIN can provide early warnings of attempts to manipulate the container's network or system settings. Similarly, tracking changes to critical system files or execution of sensitive commands helps identify unauthorized modifications that could indicate an escape. Additionally, monitoring namespace changes and mount operations is vital, as these can be direct indicators of attempts to break out of container isolation. By implementing these audit rules using tools like Auditd, organizations can significantly strengthen their ability to detect and respond to security incidents within their containerized infrastructures. Regular updates to these rules, frequent log reviews, and the integration of supplementary security measures such as SELinux [11] or AppArmor [12] further enhance the security framework, ensuring robust defense against potential container escapes.

Although the Audit rules mentioned above provide a foundational layer for monitoring potential container escapes, they are not always sufficient to detect all types of breaches. This is primarily because each PoC for a container escape can exploit different vulnerabilities that may not be directly covered by standard auditing practices. Consequently, it is crucial to continuously analyze each new PoC and the corresponding vulnerabilities they exploit to tailor specific audit rules that can capture these unique scenarios. By understanding the underlying mechanisms of each exploit, we developed targeted rules designed to detect specific patterns in the logs that signify an escape attempt.

In our experiments, our primary objective is to establish the capability to detect a container escape through log analysis, even when security measures like AppArmor and seccomp are not active. Although such security frameworks are highly effec-

tive at thwarting attacks when enabled—essentially preventing any unauthorized actions—they do not address scenarios where these protections might be bypassed or deactivated. Therefore, our focus is on developing a rule-based detection system that can identify signs of a container escape from the logs. This approach aims to provide a safety net, ensuring that potential security breaches are detected and mitigated promptly, irrespective of the state of built-in security features. By scrutinizing patterns and anomalies in the logs, we can formulate rules that help pinpoint suspicious activities indicative of an escape attempt, thereby enhancing our overall security posture even in the absence of active containment measures.

The table provided below delineates a list of system calls (syscalls) that are potentially hazardous within containerized environments. It highlights syscalls that should generally not be executed from within containers due to their ability to breach container isolation, manipulate critical host system functions, or disrupt the overall integrity and security of the host. Each syscall is described with its potential misuse in container escape scenarios, providing insights into how their execution might signify or facilitate an escape, thus emphasizing the need for stringent monitoring and restriction policies in secure container deployments.

Table 4.8: Syscalls indicative of potential container escapes

Syscall	**Potential Use in Container Escape**	**Explanation**
`setns`	Changing namespace settings	Allow processes to enter namespaces of other processes, potentially breaking out of container isolation.
`mount`	Mounting filesystems	Use to mount unauthorized filesystems, accessing sensitive host files or compromising host integrity.

Continued on next page

Table 4.8 – continued from previous page

Syscall	Potential Use in Container Escape	Explanation
`umount`	Unmounting filesystems	Disrupt file system integrity, affecting both the container and the host, similar to `mount()`.
`kexec_load`	Loading a new kernel	Execute a new kernel, bypassing standard boot processes and security mechanisms of the host.
`init_module`	Loading kernel modules	Introduce new functionalities into the kernel, potentially undermining the security of the host system.
`delete_module`	Removing kernel modules	Affect system stability and security, potentially opening up new vulnerabilities.
`clone`	Creating child processes with new namespaces	Create processes outside of the container's designated namespaces, leading to isolation breaches.
`unshare`	Disassociating parts of process execution context	Disassociate a process from its namespace, potentially leading to unauthorized access to host resources.
`reboot`	Rebooting the system	Control at the host level; its use within a container can disrupt host operations.
`swapoff` / `swapon`	Managing swap spaces	Mismanage swap, leading to performance issues or unauthorized data access on the host.

Continued on next page

Table 4.8 – continued from previous page

Syscall	Potential Use in Container Escape	Explanation
`sethostname`	Changing the system hostname	Alter the hostname from a container, leading to unauthorized network behaviors mimicking or affecting the host.

4.5.1 Analysis and Rule Crafting for CVE-2019-5736

This vulnerability allows an attacker to execute arbitrary code on the host operating system from within a container by exploiting improper handling of system file descriptors. To exploit CVE-2019-5736 [64], an attacker typically overwrites the runc binary or other critical binaries to gain unauthorized access or escalate privileges on the host system. Given the severity and exploitability of this vulnerability, we have devised a set of detection rules based on auditd log analysis. These rules are specifically tailored to monitor and flag patterns of activity that may indicate attempts to exploit this particular vulnerability, thereby enabling timely detection and response to potential security breaches.

The histogram of system calls extracted from log files provides a key visual into syscall frequency within a presumably containerized setup, crucial in analyzing CVE-2019-5736 vulnerabilities. Common syscalls like openat, read, write, and close are typically frequent and benign, whereas syscalls like execve and clone are more significant as they can indicate potential exploit attempts, such as those aiming to overwrite binaries and gain unauthorized access.

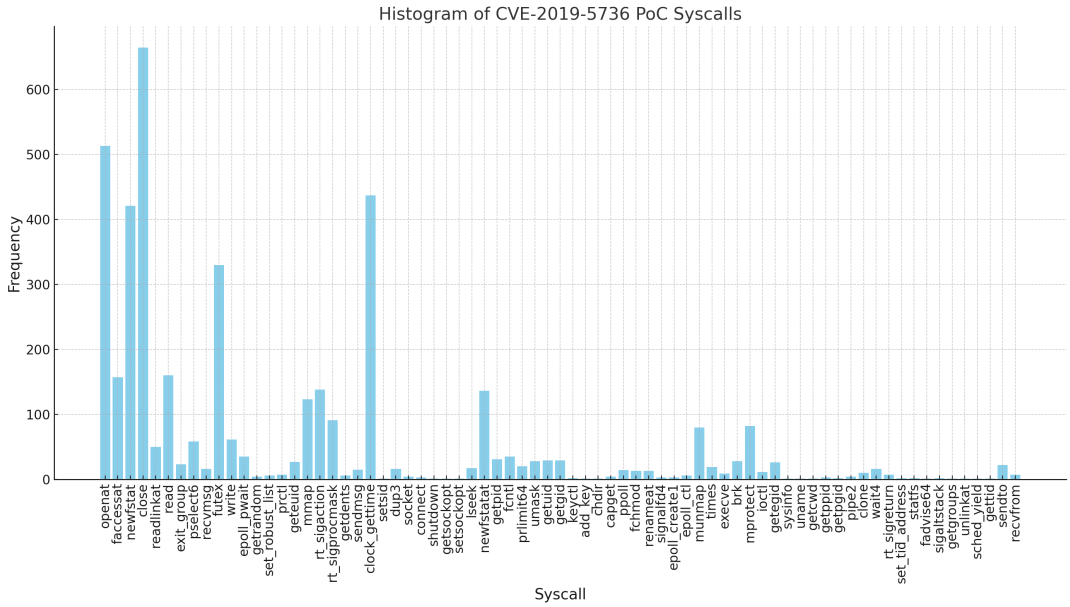


Figure 4.1: Histogram of syscall frequencies extracted from CVE-2019-5736 audit logs

Additionally, syscalls related to network and IPC activities, such as sendto, recvfrom, connect, and socket may suggest malicious networking activities. Particularly, unusual or infrequent syscalls that affect system configuration or access host-level resources warrant heightened scrutiny as potential indicators of container escapes.

Below, we have outlined the specific detection rules that are designed to identify and respond to potential exploitations of CVE-2019-5736.

- **Monitor for `execve` System Calls:**
  - *Trigger Condition:* Any `execve` system call executing a binary not typically associated with normal operations.
  - *Details to Check:* Command (comm), execution path (exe), and process title (proctitle).
- **Check for Modifications or Access to Critical System Files:**
  - *Trigger Condition:* Access to critical system files that should not be modified.

  - *Details to Check:* Path, filename, inode, and file mode (permissions).

- **Unusual Process and File Descriptors:**
  - *Trigger Condition:* Use of abnormal file descriptors in execution parameters.
  - *Details to Check:* Command-line arguments pointing to sensitive locations.

- **Context and Environment from which the Command is Executed:**
  - *Trigger Condition:* Execution from an unusual working directory or by unexpected users.
  - *Details to Check:* Current working directory (cwd) and user ID (uid).

- **Capability Changes and Permissions:**
  - *Trigger Condition:* Changes in capabilities before and after execution.
  - *Details to Check:* Changes in file capabilities (fp, fi, fe, etc.).

Use of `docker-runc` or `runc` Command

Audit logs revealed multiple instances of `EXECVE` calls executing `docker-runc` commands. As the underlying container runtime for Docker and other platforms, `runc` is critical for container management. Manipulation of `runc` through commands that deviate from standard operations may signify attempts to interfere with or escape from the containerized environment. Such activities require further investigation as they could involve unauthorized modifications or exploitation of vulnerabilities in `runc`.

Execution of Non-Standard Entry Points

The logs include executions from non-standard entry points such as `/entrypoint` and direct execution of `docker-runc` with arguments managing container states (e.g., `state`, `start`, `delete`). These actions raise concerns if they are not part of normal operational scripts or configurations. This could indicate attempts to execute unauthorized commands or scripts, potentially leading to unauthorized access or escape from the container.

Modification Access to Critical System Files

Entries in the audit trail showing access to critical system files like `/lib/ld-linux-aarch64.so.1` and execution binaries such as `/usr/bin/docker-runc` and `/bin/bash` could suggest attempts at privilege escalation or container breakout. These are common tactics in targeted attacks on containerized environments.

Suspicious Command-Line Arguments

Certain logs recorded command-line arguments involving file descriptors (e.g., `/proc/self/fd/3`), which may indicate attempts to leverage existing file descriptors for malicious purposes like code injection or escaping containment. Such usage patterns are unusual and potentially malicious, warranting detailed security reviews and monitoring.

Unusual Capability and Permission Changes

`BPRM_FCAPS` entries indicated capability changes before and after the execution of commands. These changes could facilitate unauthorized privilege escalation within the host system, representing a serious security risk. It is essential to scrutinize such changes to determine if they are part of legitimate operations or malicious interventions.

The specific exploitation of runc through manipulation of system call behaviors, such as execve, and the modification of critical system binaries, represent a direct threat to container isolation—a foundational security principle in containerized environments. The established detection rules and observed attack patterns specific to CVE-2019-5736 not only highlight the need for rigorous system monitoring but also underscore the importance of updating and securing container management tools against such sophisticated intrusion techniques

4.5.2 Analysis and Rule Crafting for CVE-2020-15257

CVE-2020-15257 [65] is a critical vulnerability that affects containerized environments utilizing containerd-shim, a component integral to the containerd, widely used in orchestration systems like Kubernetes. The flaw arises from improper handling of process namespaces within containerd-shim, particularly involving the clone, un-

share, and setns syscalls. These syscalls, which are essential for namespace management, can be exploited to allow unauthorized access to the host namespace from within a container. The exploitation sequence involves initiating a syscall that targets the host namespace, which, due to the vulnerability, is not properly restricted by containerd-shim, enabling the attacker to escape the container's isolation. This escape grants the attacker elevated privileges, often as root, leading to potential full system compromise, data breaches, and further exploitation.

The histogram below illustrates the frequency of various syscalls from the new log file, revealing several key insights. The syscalls ppoll and recvfrom dominate with the highest frequencies, indicating extensive network communication and IO multiplexing. Other frequent syscalls like exit_group and write suggest numerous process terminations and substantial writing operations. Moderately frequent syscalls such as newfstatat, openat, and futex reflect regular file status checks, file opening activities, and thread synchronization.

Although low in frequency, critical syscalls like clone, execve, setns, unshare, and pivot_root are significant due to their roles in process and namespace management, often associated with container escapes. Rare syscalls such as capset and keyctl denote specific configuration changes. Overall, the histogram highlights a highly active system with considerable interaction between processes and resources, underlining the importance of monitoring critical syscalls to detect potential security breaches.

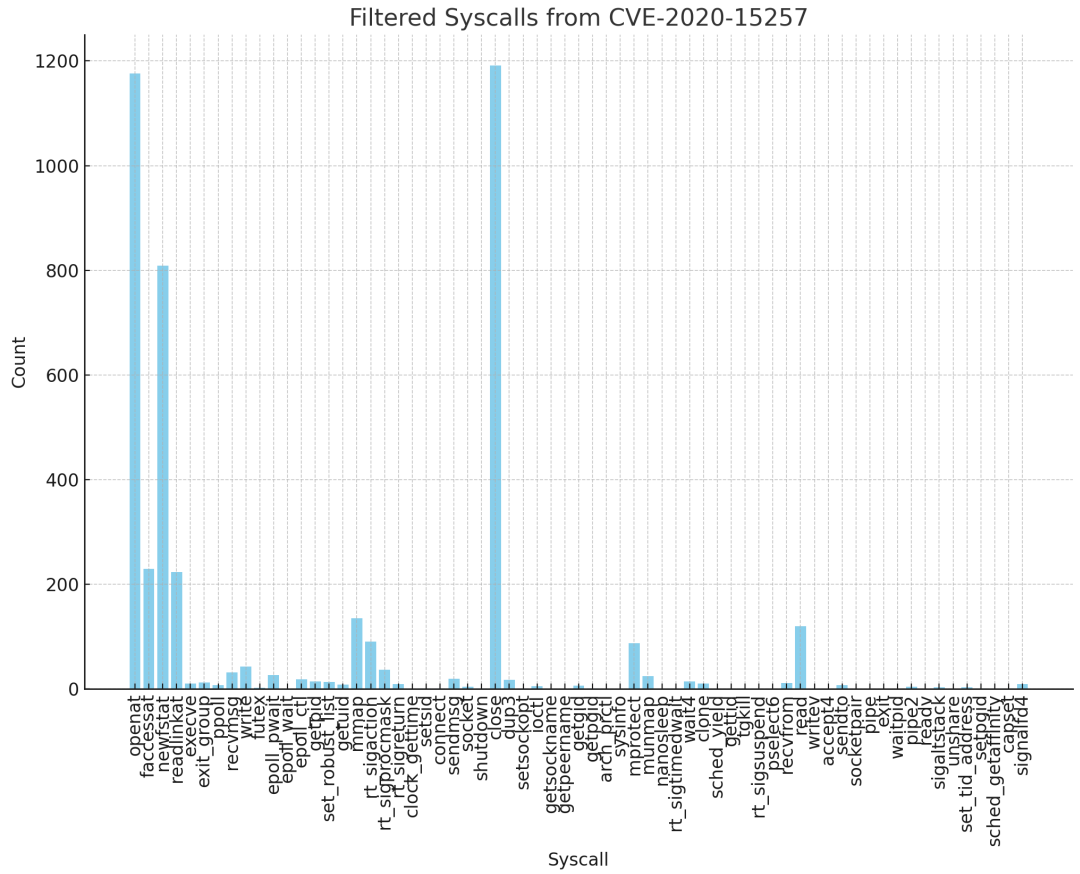


Figure 4.2: Histogram of syscall frequencies extracted from CVE-2020-15257 audit logs

- **Execution of Shell Command (Potential Reconnaissance):**
  - *Trigger Condition:* Execution of a shell command to gather system information and write to a temporary file.
  - *Details to Check:* Command (comm), execution path (exe), and process title (proctitle).
- **Syscall `clone` (Process Creation):**
  - *Trigger Condition:* Use of the `clone` syscall to create a new process.
  - *Details to Check:* Command (comm), syscall parameters, and parent process ID (ppid).
- **Changes to User and Group IDs:**

- *Trigger Condition:* Changes to group and user IDs.
- *Details to Check:* Syscall parameters, user ID (uid), and group ID (gid).

The log file analysis reveals multiple instances where the execve system call was utilized, running various processes with root (0) UID/EUID permissions. These processes, employing various system tools such as sudo, dash, mktemp, gawk, grep, ls, cat, cmp, rm, and auditctl, indicate typical system management and configuration activities. However, when misused, these calls can also be exploited to leverage security vulnerabilities.

Instances of execve and capset system calls are documented. Notably, the execve system call has been used successfully, changing the user ID (UID) and group ID (GID) to root. Furthermore, the capset system call was used to assign extensive capability sets, posing potential risks related to gaining or modifying high-level privileges.

The execve system call executed a shell command to gather system information and write it to a temporary file:

```
type=EXECVE msg=audit(03/24/24 23:40:01.074:14230756) :
argc=3 a0=/bin/sh a1=-c a2=( echo "# id" ; id ; echo
"# cat /proc/self/status"; cat /proc/self/status ) >
/tmp/shimmer.out
type=SYSCALL msg=audit(03/24/24 23:40:01.074:14230756) :
arch=aarch64 syscall=execve success=yes exit=0
a0=0xaaaadd21ba26 a1=0xffffc65ba320 a2=0xaaaadd2328c0
a3=0xffffc65ba338 items=2 ppid=14027 pid=14028
auid=root uid=root gid=root euid=root suid=root
fsuid=root egid=root sgid=root fsgid=root tty=(none)
ses=887 comm=sh exe=/usr/bin/dash key=execve-key
```

This command runs with root privileges, indicated by the UID and GID set to root. Commands like `id` and `cat /proc/self/status` are used to check process identity and status, providing valuable information for system reconnaissance.

```
type=CAPSET msg=audit(03/24/24 23:40:01.062:14229242) :
pid=14027 cap_pi=chown,dac_override,dac_read_search,
fowner,fsetid,kill,setgid,setuid,setpcap,linux_immutable,
net_bind_service,net_broadcast,net_admin,net_raw,
ipc_lock,ipc_owner,sys_module,sys_rawio,sys_chroot,
cap_pp=chown,dac_override,...cap_pa=none
```

The clone system call was successfully used to create a new process. The new process was created with root privileges, as shown by the UID and GID set to root. The cron process appears to have initiated this, suggesting it might be part of a scheduled task.

Examination of /etc/crontab File Access

Access to the /etc/crontab file was noted, potentially to view or modify system scheduled tasks. A malicious user or software could manipulate the crontab to schedule unwanted commands at regular intervals, posing a significant security risk. Any access to /etc/crontab, especially with intent to modify, should be scrutinized.

```
type=SYSCALL msg=audit(03/24/24 23:40:01.062:14229242) :
arch=aarch64 syscall=clone success=yes exit=0
a0=0xaaaadd21b7d4 a1=0xaaaadd21b7dc a2=0xaaaadd21b7dc
a3=0x0 items=0 ppid=901 pid=14027 auid=unset
uid=root gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=(none) ses=unset
comm=cron exe=/usr/sbin/cron key=all-syscalls-64
```

Subsequent Socket Creation and Connection Attempt by Cron

The syscall log entries also indicate multiple socket system calls, as well as attempts to connect to various sockets. These entries are critical in analyzing potential container escape attempts that exploit socket communications.

```
type=TITLE msg=audit(03/24/24 23:40:01.062:14229141) :
proctitle=/usr/sbin/cron -f type=SYSCALL
```

```
msg=audit(03/24/24 23:40:01.062:14229141) :
arch=aarch64 syscall=lseek success=yes
exit=0 a0=0x7 a1=0x0 a2=SEEK_CUR
a3=0xffffbac8a668 items=0 ppid=901
pid=14027 auid=unset uid=root gid=root
euid=root suid=root fsuid=root egid=root
sgid=root fsgid=root tty=(none)
ses=unset comm=cron exe=/usr/sbin/cron
key=all-syscalls-64

----

type=PROCTITLE msg=audit(03/24/24 23:40:01
.062:14229142): proctitle=/usr/sbin/cron -f
type=SYSCALL msg=audit(03/24/24 23:40:01.062:142291
42): arch=aarch64 syscall=newfstat success=yes
exit=0 a0=0x7 a1=0xffffc6578e28
a2=0xffffc6578e28 a3=0x7 items=0
ppid=901 pid=14027 auid=unset
uid=root gid=root euid=root suid=root
fsuid=root egid=root sgid=root
fsgid=root tty=(none) ses=unset
comm=cron exe=/usr/sbin/cron
key=all-syscalls-64
```

We have also plotted timestamp vs socket system calls to notice the frequency of socket calls.

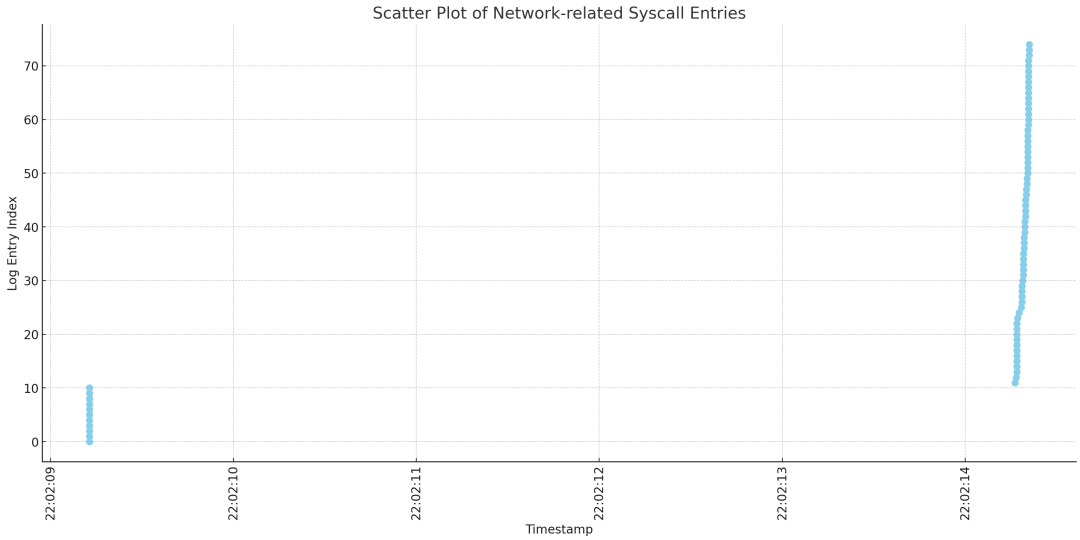


Figure 4.3: Scatter Plot of syscall frequencies extracted from CVE-2020-15257 audit logs

Although this was a brief test run conducted in a non-production environment, resulting in a limited number of log entries, the network syscall plot remains crucial. This plot helps us understand the frequency and distribution of network-related system calls, highlighting potential anomalies. The limited scope of the test emphasizes the importance of performing comprehensive analyses in realistic scenarios to gain a complete understanding of system behavior and identify potential issues.

The exploitation of CVE-2020-15257, particularly via the manipulation of the capset system call and modification of critical container runtime files, poses a significant threat to container isolation. Further complicating the issue, the misuse of cron jobs to schedule harmful tasks and the exploitation of socket system calls for unauthorized network communications increase the risk. Attackers can leverage cron jobs for persistence and use socket calls to create covert communication channels.

4.5.3 Analysis and Rule Crafting for CVE-2019-14271

Exploiting CVE-2019-14271 [66] typically involves manipulating Docker API calls to inject malicious payloads. Attackers can craft special inputs to be sent to the Docker daemon, which, if improperly handled, can trigger the execution of arbitrary

code with root privileges. This exploit breaks the container's isolation, granting attackers control over the host system and posing significant risks to the infrastructure.

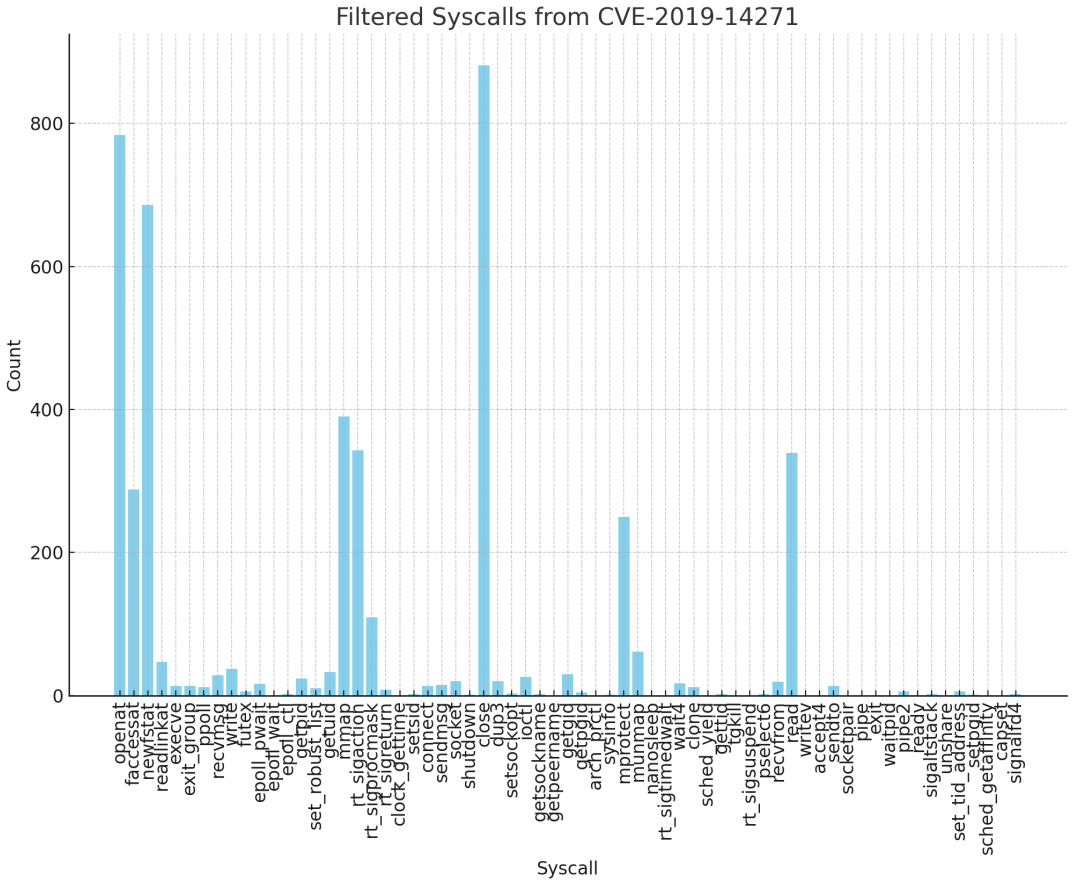


Figure 4.4: Histogram of syscall frequencies extracted from CVE-2019-14271 audit logs

The histogram displays the frequency of specific syscalls from the CVE-2019-14271 log file, highlighting those that are commonly associated with container operations, and potential escape mechanisms. Syscalls such as execve, mmap, and setns are noteworthy as they play critical roles in process execution, memory management, and namespace manipulation, respectively. The presence and frequency of these syscalls can indicate attempts to execute new processes, alter memory mappings, or change namespace contexts, which are essential steps in container escape exploits. Monitoring and restricting these syscalls can significantly enhance the security posture by preventing unauthorized actions that could lead to a container breakout.

- **Syscall `clone` (Process Creation):**
  - *Trigger Condition:* Use of the `clone` syscall to create a new process.
  - *Details to Check:* Command (comm), syscall parameters, and parent process ID (ppid).
- **Syscall `openat` (File Access):**
  - *Trigger Condition:* Use of the `openat` syscall to access files in sensitive directories.
  - *Details to Check:* Accessed file path, syscall parameters, and user ID (uid).
- **Syscall `write` (File Write):**
  - *Trigger Condition:* Use of the `write` syscall in sensitive directories.
  - *Details to Check:* Written file paths, syscall parameters, and user ID (uid).
- **Syscall `execve` (Process Execution):**
  - *Trigger Condition:* Use of the `execve` syscall in sensitive directories.
  - *Details to Check:* Executed command and paths, syscall parameters, and user ID (uid).
- **Docker `untar` Operation:**
  - *Trigger Condition:* Execution of the `docker-untar` command accessing or modifying files in /lib, /usr, or /etc directories.
  - *Details to Check:* Command (proctitle), syscall type (`openat`, `write`, `execve`), and accessed or modified file paths.

To detect container escapes related to CVE-2019-14271 through syscall monitoring, it is crucial to focus on specific syscalls. Key syscalls to monitor include chroot for changes to root directories, execve for the execution of new programs, open and openat for file access, fork and vfork for process creation, mount for filesystem mounting, and prctl for process behavior modifications. Observing these syscalls can reveal unauthorized chroot operations, unexpected program executions, abnormal file

accesses, new process initiations, and unusual filesystem mounts. This targeted monitoring, especially within the context of the docker-tar process, enhances the ability to detect and respond to exploitation attempts, thereby strengthening the security of containerized environments.

4.5.4 Analysis and Rule Crafting for CVE-2022-0847

CVE-2022-0847 [67], also known as "Dirty Pipe", is a significant kernel vulnerability identified in the Linux kernel in 2022. This vulnerability stems from a flaw in the handling of pipe buffers, which can be exploited by local attackers, to escalate privileges. The attack involves creating a pipe, filling its buffer with arbitrary data, and then using the splice system call to transfer this data into a target file, effectively bypassing the file's read-only permissions. This allows attackers to overwrite sensitive files, inject malicious code or alter system configurations, potentially leading to root access.

Like other critical vulnerabilities discussed in this thesis, such as CVE-2019-14271 [66], CVE-2022-0847 [67] is particularly concerning in the context of containerized environments. Despite the isolation that containers are supposed to provide, these kernel vulnerabilities can be leveraged to escape from the container and gain control over the host system. Once root access is obtained on the host, an attacker can manipulate all containers running on it, leading to extensive security breaches. This section and the subsequent ones focus on these kernel vulnerabilities due to their potential to enable container escapes, highlighting the critical need for continuous monitoring, timely patching and robust security practices to protect containerized infrastructures.

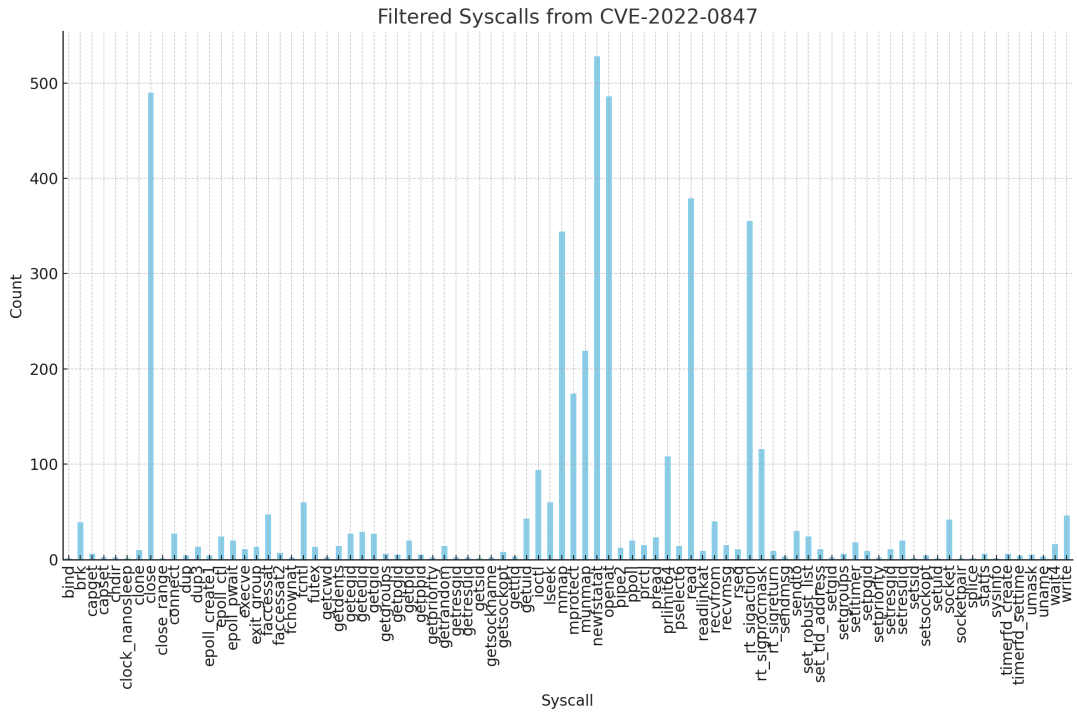


Figure 4.5: Histogram of syscall frequencies extracted from CVE-2022-0847 audit logs

- **Sequence of `pipe` and `splice` Syscalls:**
  - *Trigger Condition:* Use of the `pipe` and `splice` syscalls in sequence.
  - *Details to Check:* This combination is a strong indicator of an attempt to manipulate pipe buffers and overwrite files.
- **File Access Attempts:**
  - *Trigger Condition:* Unusual `open` syscalls for read-only files.
  - *Details to Check:* Especially if followed by `splice`.
- **Execution of New Processes:**
  - *Trigger Condition:* `execve` syscalls.
  - *Details to Check:* This may indicate the execution of malicious scripts or binaries.

By monitoring specific syscall patterns such as the frequent and sequential use of pipe and splice, unusual open syscalls for read-only files followed by splice, and execve

syscalls, we can effectively identify and mitigate attempts to exploit the CVE-2022-0847 "Dirty Pipe" vulnerability. These patterns are strong indicators of malicious activity aimed at manipulating pipe buffers and overwriting files, a core mechanism of this exploit. Detecting these specific sequences allows for real-time intervention, preventing privilege escalation and maintaining data integrity.

4.5.5 Analysis and Rule Crafting for CVE-2022-0492

The exploitation of CVE-2022-0492 [**?**] involves several steps. First, an attacker must gain access to a non-privileged user account on the target system. This can be achieved through various means, such as phishing, exploiting another vulnerability, or using stolen credentials. Once the attacker has user-level access, they can create a new cgroup and configure its release_agent to point to a malicious script or binary that they control.

The attacker then triggers the condition for the cgroup to become empty, typically by moving all processes out of the cgroup. This action causes the kernel to execute the specified release_agent, running the attacker's malicious script with root privileges. Through this process, the attacker gains root access to the system, allowing them to perform any administrative actions, such as modifying system configurations, installing backdoors or exfiltrating sensitive data.

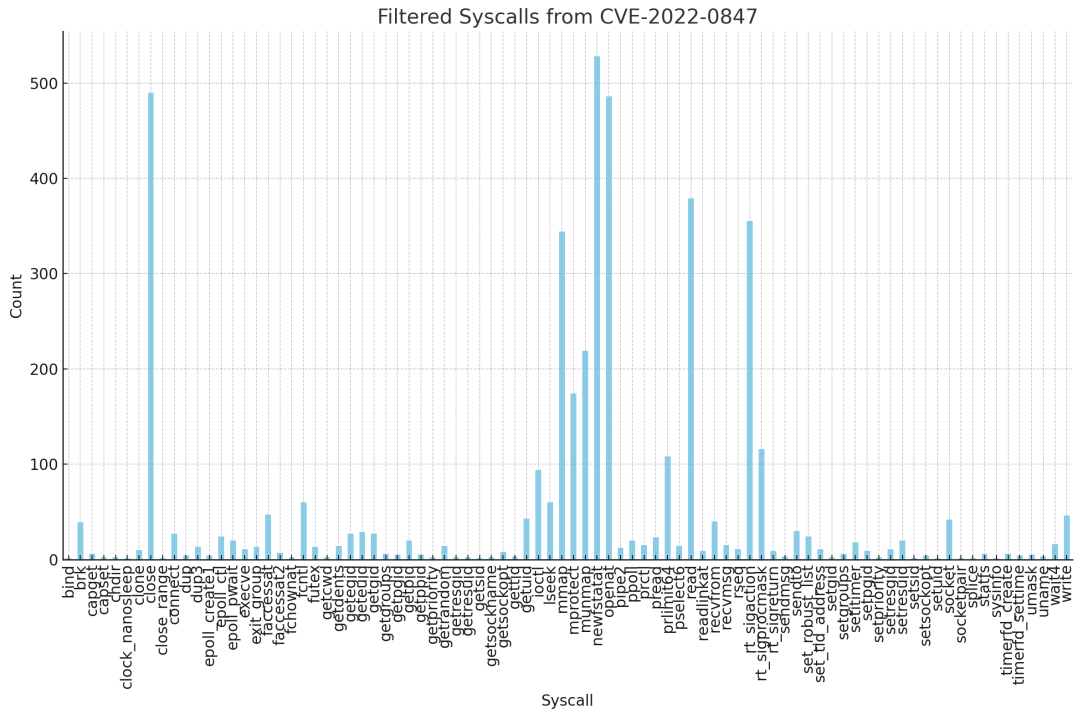


Figure 4.6: Histogram of syscall frequencies extracted from CVE-2022-0492 audit logs

- **User Namespace Creation:**
  - *Trigger Condition:* `unshare` syscalls.
  - *Details to Check:* Look for entries containing `unshare`. This indicates the creation of a new user namespace.
- **Cgroup Mounting:**
  - *Trigger Condition:* `cgroup` mount operations.
  - *Details to Check:* Look for entries containing `cgroup`. This indicates the mounting of a cgroup filesystem.
- **Setting the release_agent:**
  - *Trigger Condition:* Configuration changes involving `release_agent`.
  - *Details to Check:* Look for entries containing `release_agent`. This indicates an attempt to set or modify the release agent.

- **Execution of release_agent:**
  - *Trigger Condition:* `execve` syscalls.
  - *Details to Check:* Look for entries containing `execve`. This may indicate the execution of the configured release agent, potentially running a malicious script or binary.

To detect potential container escape attempts, we examined the log entries for specific actions indicative of the exploit process. The analysis focused on identifying user namespace creation, cgroup mounting, setting the release_agent, and executing the release_agent.

The first step in detecting the exploit involves looking for entries related to the unshare syscall, which is used to create new namespaces. This action is critical for exploiting CVE-2022-0492, as it allows the container to gain the CAP_SYS_ADMIN capability in the new user namespace. An example log entry indicating user namespace creation is:

```
type=PROCTITLE msg=audit(03/30/24 12:49:03.424:24263) :
proctitle=/bin/bash /can-ctr-escape-cve-2022-0492.sh
type=PATH msg=audit(03/30/24 12:49:03.424:24263) :
item=0 name=/usr/bin/unshare inode=43475 dev=00:33
mode=file, 755 ouid=root ogid=root rdev=00:00
nametype=NORMAL cap_fp=none cap_fi=none
cap_fe=0 cap_fver=0

type=SYSCALL msg=audit(03/30/24 12:49:03.424:24263) :
arch=aarch64 syscall=execve success=yes exit=0
a0=0xaaaae8234a90 a1=0xaaaae8231c60
a2=0xaaaae8231940 a3=0xffffb83570b0 items=2
ppid=3861 pid=3876 auid=unset uid=root gid=root
euid=root suid=root fsuid=root egid=root sgid=root
fsgid=root tty=pts0 ses=unset comm=unshare
exe=/usr/bin/unshare key=execve-key
```

Next, we searched for entries related to the mounting of cgroup filesystems. This step is necessary for the container to manipulate the cgroup settings. An example log entry indicating cgroup mounting is:

```
type=PROCTITLE msg=audit(03/30/24 12:49:03.424:24293) :
proctitle=bash -c mount -t cgroup -o blkio cgroup
/tmp/.cve-2022-0492-test 2>&1 >/dev/null && test -w
/tmp/.cve-2022-0492-test/release_agent

type=SYSCALL msg=audit(03/30/24 12:49:03.424:24293) :
arch=aarch64 syscall=openat success=yes exit=3
a0=0xffffffffffffff9c a1=0xaaab13ee9e30
a2=O_WRONLY|O_CREAT|O_TRUNC a3=0x1b6 items=2
ppid=3876 pid=3877 auid=unset uid=root gid=root
euid=root suid=root fsuid=root egid=root sgid=root
fsgid=root tty=pts0 ses=unset comm=bash
exe=/usr/bin/bash key=openat-watch-64
```

We then looked for entries that indicate setting the release_agent, which is used to specify a script or binary to be executed when a cgroup is released. An example log entry indicating the setting of release_agent is:

```
type=PROCTITLE msg=audit(03/30/24 12:49:03.440:24322) :
proctitle=/bin/bash /can-ctr-escape-cve-2022-0492.sh
type=EXECVE msg=audit(03/30/24 12:49:03.440:24322) :
argc=6 a0=unshare a1=-UrmC a2=--propagation=unchanged
a3=bash a4=-c a5=mount -t cgroup -o devices cgroup
/tmp/.cve-2022-0492-test 2>&1 >/dev/null && test -w
/tmp/.cve-2022-0492-test/release_agent
```

```
type=SYSCALL msg=audit(03/30/24 12:49:03.440:24322) :
arch=aarch64 syscall=execve success=yes exit=0
a0=0xaaaae8236350 a1=0xaaaae8231c60 a2=0xaaaae8231940
a3=0xffffb83570b0 items=2 ppid=3861 pid=3878 auid=unset
uid=root gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=pts0 ses=unset
comm=unshare exe=/usr/bin/unshare key=execve-key
```

By systematically searching for these specific log entries, we were able to detect signs of the CVE-2022-0492 exploit, demonstrating how a container could potentially escape and gain elevated privileges.

4.5.6 Analysis and Rule Crafting for Capabilities

Linux capabilities allow splitting the root privileges into distinct units, which can be independently enabled or disabled. This granularity helps in limiting the actions that can be performed by processes within a container, reducing the attack surface.

- **Capability Change Detection:**
  - *Trigger Condition:* `capset` and `prctl` syscalls.
  - *Details to Check:* Look for entries where `syscall` is `capset` or `prctl`. This may indicate attempts to change capabilities, which could be a sign of container escape attempts.
- **Non-Root User ID Detection:**
  - *Trigger Condition:* Syscalls with non-root user IDs.
  - *Details to Check:* Look for entries where `uid`, `euid`, `suid`, or `fsuid` are not `root`. This may indicate processes running within user namespaces, potentially reducing privileges for security.
- **Frequent Capability Changes:**
  - *Trigger Condition:* Multiple `capset` or `prctl` syscalls within a short time frame.

  - *Details to Check:* Analyze logs for frequent capability changes by the same process or user. This could indicate an attempt to escalate privileges or manipulate system capabilities repeatedly.

- **User Namespace Usage:**
  - *Trigger Condition:* `unshare` and `clone` syscalls.
  - *Details to Check:* Look for entries where `syscall` is `unshare` or `clone` with `CLONE_NEWUSER`. This indicates the creation of a user namespace, which could be used to isolate processes with different user IDs.

- **Execution of Non-Root Processes:**
  - *Trigger Condition:* `execve` syscalls with non-root user IDs.
  - *Details to Check:* Look for entries where `syscall` is `execve` and `uid`, `euid`, `suid`, or `fsuid` are not `root`. This can help detect when non-root processes are being executed, which is common in user namespaces.

By focusing on key aspects such as capset and prctl syscalls for capability changes, non-root user ID detection, frequent capability changes, user namespace usage, and the execution of non-root processes, potential security breaches can be identified and mitigated. For instance, an example log entry from the collected audit logs for capability change detection is:

```
type=SYSCALL msg=audit(11/03/24 14:55:52.151:96330) :
arch=aarch64 syscall=capset success=yes exit=0
a0=0xaaaaf6f94b74 a1=0xaaaaf6f94b7c
a2=0xaaaaf6f94b7c a3=0x0 items=0 ppid=524
pid=24633 auid=unset uid=root gid=root
euid=root suid=root fsuid=root egid=root
sgid=root fsgid=root tty=(none) ses=unset
comm="(genrules)" exe="/usr/lib/systemd/systemd"
key="audit_all_syscalls"
```

For non-root user ID detection, the trigger condition is syscalls with non-root user IDs. We should look for entries where uid, euid, suid, or fsuid are not root, as this may indicate processes running within user namespaces, potentially reducing privileges for security. An example log entry would be:

```
type=SYSCALL msg=audit(03/30/24 14:55:52.151:96330) :
arch=aarch64 syscall=write success=yes exit=4
a0=0xa a1=0xffffe18cae00 a2=0x4 a3=0xffffa8a3f000
items=0 ppid=524 pid=24633 auid=unset
uid=unknown(231072) gid=root euid=unknown(231072)
suid=unknown(231072) fsuid=unknown(231072)
egid=root sgid=root fsgid=root tty=(none)
ses=unset comm="runc:[1:CHILD]" exe="/"
key="write-watch-64"
```

4.6 Proposed Design Model for Container Escape Detection System

In this section, we present a proposed design model for a real-time Docker container escape detection system. This system is designed to address existing shortcomings in tools such as Tetragon [22], and Tracee [21], focusing on improved performance, flexibility, and backward compatibility.

4.6.1 System Overview

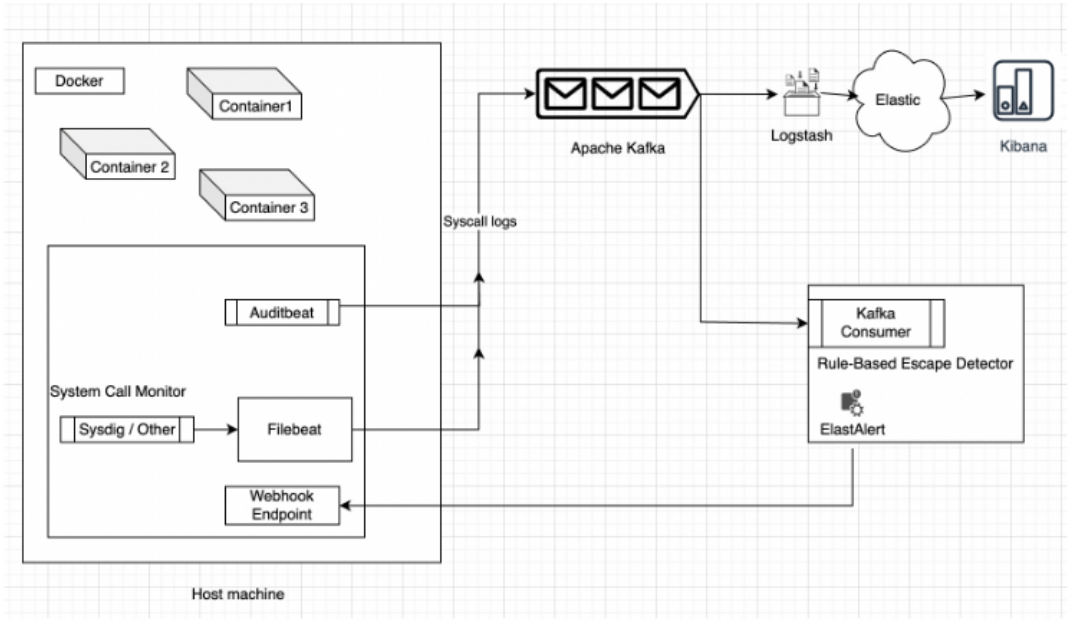


Figure 4.7: Proposed system architecture

The proposed design leverages lightweight system call monitoring, scalable centralized log management, and rule-based detection mechanisms. The system aims to provide real-time detection of container escape attempts with minimal performance impact on the host system. By integrating these components, the system ensures that logs are efficiently collected, stored, and analyzed while maintaining compatibility with older kernel versions and minimizing resource consumption.

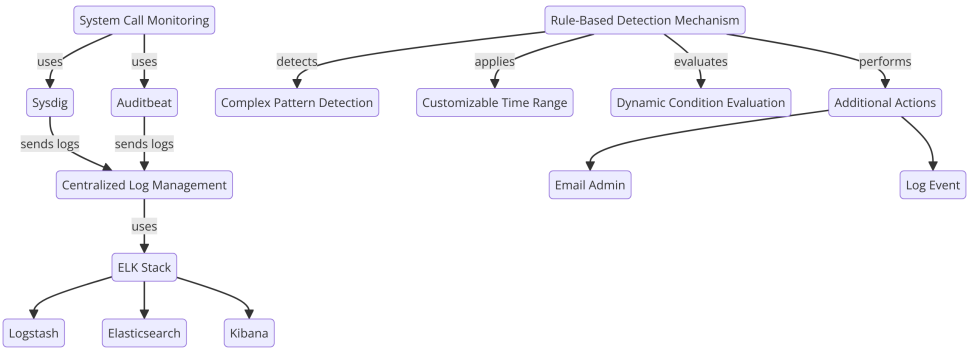


Figure 4.8: Proposed system graph diagram

4.6.2 Components of the System

4.6.2.1 System Call Monitoring

The system employs tools such as Sysdig and Auditbeat to monitor system calls on the host system. These tools are configured to filter and collect only relevant events to reduce overhead and ensure compatibility with a wide range of kernel versions.

Sysdig is a powerful tool for capturing system calls and other OS-level events. It operates by intercepting system calls using kernel modules and provides detailed information about each event, including the process ID, command, user ID, and other contextual information. This allows for precise monitoring of activities within Docker containers.

Auditbeat is an alternative monitoring tool that integrates with the Linux audit framework. It captures system call data and other security-related events, sending this information to a central log management system. Auditbeat is particularly useful in environments where detailed auditing is required, as it can provide a comprehensive view of system activities.

Configuration examples for Sysdig and Auditbeat are provided below to illustrate how these tools can be set up to monitor specific system calls associated with container escapes:

4.6.2.2 Centralized Log Management

Logs generated by Sysdig [80] or Auditbeat [81] are sent to a centralized log management system like the ELK Stack (Elasticsearch [82], Logstash [83], Kibana [84]). This setup ensures efficient storage, indexing, and retrieval of log data.

Logstash acts as a data processing pipeline that ingests logs from Sysdig or Auditbeat, processes them, and forwards them to Elasticsearch. It supports various input and output plugins, making it highly flexible for integrating with different data sources and destinations.

Elasticsearch is a distributed, RESTful search and analytics engine capable of storing large volumes of log data. It provides powerful indexing and search capabilities, allowing for fast and efficient querying of log data.

Kibana is a data visualization tool that works on top of Elasticsearch. It allows users to create visualizations, dashboards, and reports based on the log data stored in Elasticsearch. This provides a user-friendly interface for monitoring system activities and detecting suspicious events.

4.6.2.3 Rule-Based Detection Mechanism

The system employs a set of predefined rules to detect suspicious activities indicative of container escape attempts. These rules are dynamically updatable to address new threats as they emerge.

While Tetragon can detect individual system calls, our system focuses on detecting complex sequences and patterns of system calls that are more indicative of sophisticated container escape attempts. By monitoring the order and combination of system calls, we can reduce false positives and increase the accuracy of detection. Below are some of the features we have considered in this proposal:

- **Complex Pattern Detection**: The bool query within `must` allows for the detection of sequences and combinations of system calls (`execve`, `setns`, `capset`), reducing false positives by ensuring multiple conditions are met within a short time frame.
- **Customizable Time Range**: The range filter ensures that the detected pattern occurs within a specified time frame (e.g., the last minute), making the detection more relevant and timely.
- **Dynamic Condition Evaluation**: The script condition allows for more complex logic to determine if the rule should trigger an alert, providing flexibility for future enhancements.
- **Additional Actions**: Besides sending an email, the system also logs the event, which helps in maintaining a historical record for further analysis and auditing.

```
{
  "trigger": {
    "schedule": {
      "interval": "10s"
    }
  },
  "input": {
    "search": {
      "request": {
        "indices": ["sysdig-logs-*", "auditbeat-logs-*"],
        "body": {
          "query": {
            "bool": {
              "must": [
                {
                  "match": {
                    "syscall": "execve"
                  }
                },
                {
                  "match": {
                    "syscall": "setns"
                  }
                },
                {
                  "match": {
                    "syscall": "capset"
                  }
                }
              ]
            }
          }
        }
```

```
      }
    }
  },
  "condition": {
    "compare": {
      "ctx.payload.hits.total": {
        "gte": 1
      }
    }
  },
  "actions": {
    "email_admin": {
      "email": {
        "to": "admin@example.com",
        "subject": "Potential Container Escape Pattern",
        "body": "Container escape has been detected."
      }
    }
  }
}
```

This JSON rule example defines a rule for detecting patterns indicative of a container escape attempt. The rule is triggered every 10 seconds, searches for specific system call patterns (execve, execveat, setns, capset) within the last minute, and sends an email alert to the administrator if the pattern is detected.

CHAPTER 5

CONCLUSION

The exploration and analysis of container escape vulnerabilities in Docker environments, as detailed in this thesis, underscore the critical need for robust security measures in containerized systems. The study successfully implemented Proof of Concepts (PoCs) for specific Common Vulnerabilities and Exposures (CVEs) on ARM-based architectures, demonstrating the viability and implications of these vulnerabilities in less commonly studied hardware environments. By leveraging system call logging with Auditd and employing rule-based log analysis, the research provided a structured approach to detect and understand malicious activities, significantly advancing the field of container security.

5.1 Future Work and Limitations

Despite the significant findings and contributions of this research, several limitations and areas for future work remain:

- **Broadening the Scope of Vulnerabilities:** While this study focused primarily on Docker vulnerabilities, future research should also consider kernel-level vulnerabilities that can impact Docker containers. Expanding the analysis to include a wider range of vulnerabilities will provide a more comprehensive security framework.
- **Cross-Platform Analysis:** Extending the analysis to include other container platforms, such as Kubernetes, will help develop universally applicable security strategies. This will ensure that the proposed system can be adapted to protect

different containerized environments.

- **Implementation and Validation of the Proposed System:** Future work should focus on the actual implementation of the proposed container escape detection system. This includes deploying the system in a real-world environment, validating its effectiveness, and optimizing its performance based on empirical data.

In conclusion, the proposed Rule-Based Log Analysis system design offers a robust framework for detecting and mitigating Docker-specific container escape vulnerabilities. Continuous enhancements and adaptation to evolving threats are essential to maintaining the system's effectiveness in protecting Docker environments.

REFERENCES

[1] I. Docker, "Docker: Enterprise container platform." https://www.docker.com, 2023. Accessed: 2023-07-24.

[2] Google, "Google kubernetes engine." https://cloud.google.com/kubernetes-engine/, 2023. Accessed: 2023-07-24.

[3] Y. Wu, L. Lei, Y. Wang, K. Sun, and J. Meng, "Evaluation on the security of commercial cloud container services," in *Information Security Conference*, 2020.

[4] S. Kim, B. J. Kim, and D. H. Lee, "Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 278–287, Sept. 2021. ISSN: 2159-6190.

[5] M. Reeves, D. J. Tian, A. Bianchi, and Z. B. Celik, "Towards Improving Container Security by Preventing Runtime Escapes," in *2021 IEEE Secure Development Conference (SecDev)*, pp. 38–46, Oct. 2021.

[6] HashiCorp, "Vagrant." https://developer.hashicorp.com/vagrant, 2024. Accessed: 2024-06-24.

[7] Oracle Corporation, "Oracle vm virtualbox," 2023. Accessed: 2023-07-24.

[8] VMware, Inc., "Vmware," 2023. Accessed: 2023-07-24.

[9] Microsoft Corporation, "Hyper-v," 2023. Accessed: 2023-07-24.

[10] Linux Kernel Organization, "Linux security modules (lsm)," 2023. Accessed: 2023-07-24.

[11] The kernel development community, "Selinux project," 2023. Accessed: 2023-07-24.

[12] AppArmor Project, “Apparmor,” 2023. Accessed: 2023-07-24.

[13] T. L. man-pages project, *seccomp - operate on Secure Computing state of the process*, 2021. Accessed: 2024-07-14.

[14] T. M. Corporation, “Common vulnerabilities and exposures (cve),” 2024. Accessed: 2024-07-14.

[15] The Linux Foundation, *strace - Trace System Calls and Signals*, 2024.

[16] T. eBPF Community, “ebpf - introduction, tutorials & community resources,” 2024. Accessed: 2024-07-14.

[17] I. Documentation, “truss command - monitor and trace system calls,” 2024. Accessed: 2024-07-14.

[18] The Linux Foundation, *auditd - Linux Auditing System*, 2024.

[19] Oracle Corporation, “Oracle solaris 11.4 documentation,” 2024. Accessed on 2024-07-14.

[20] The FreeBSD Project, “Freebsd handbook,” 2024. Accessed on 2024-07-14.

[21] Aqua Security Team, “Tracee: Linux Runtime Security and Forensics using eBPF,” 2024.

[22] Cilium Project Team, “Tetragon: eBPF-based Security Observability and Runtime Enforcement,” 2024.

[23] B. Gregg, A. Starovoitov, D. Borkmann, *et al.*, “Bcc - bpf compiler collection.” `https://github.com/iovisor/bcc`, 2014.

[24] I. Kraiem, F. Ghozzi, A. Péninou, G. Roman-Jimenez, and O. Teste, “Automatic classification rules for anomaly detection in time-series,” pp. 321–337, 2020.

[25] W. Gan, L. Chen, S. Wan, J. Chen, and C.-M. Chen, “Anomaly rule detection in sequence data,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, pp. 12095–12108, 2021.

[26] T. Vollmer, J. Alves-Foss, and M. Manic, “Autonomous rule creation for intrusion detection,” *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pp. 1–8, 2011.

[27] I. T. Aktolga, E. S. Kuru, Y. Sever, and P. Angin, "Ai-driven container security approaches for 5g and beyond: A survey," *ArXiv*, vol. abs/2302.13865, 2023.

[28] M. Azab, B. Mokhtar, A. S. Abed, and M. Eltoweissy, "Toward Smart Moving Target Defense for Linux Container Resiliency," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pp. 619–622, Nov. 2016. arXiv:1611.03065 [cs].

[29] T. Lu and J. Chen, "Research of Penetration Testing Technology in Docker Environment," in *Proceedings of the 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*, (Chongqing, China), Atlantis Press, 2017.

[30] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – Vulnerability Analysis," *Computer Communications*, vol. 122, pp. 30–43, June 2018.

[31] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, (San Juan PR USA), pp. 418–429, ACM, Dec. 2018.

[32] Z. Jian and L. Chen, "A Defense Method against Docker Escape Attack," in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, (Wuhan China), pp. 142–146, ACM, Mar. 2017.

[33] L. Rice, *Container Security: Fundamental Technology Concepts that Protect Containerized Applications*. O'Reilly Media, 2020.

[34] M. J. Reeves, "Investigating escape vulnerabilities in container runtimes," master's thesis, Purdue University, 2021.

[35] M. Abbas, S. Khan, A. Monum, F. Zaffar, R. Tahir, D. Eyers, H. Irshad, A. Gehani, V. Yegneswaran, and T. Pasquier, "PACED: Provenance-based Automated Container Escape Detection," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, (CA, USA), pp. 261–272, IEEE, Sept. 2022.

[36] K. Wu, D. Yang, X. Gao, W. Yang, M. Li, and D. Wang, "Process based container escape monitoring and resource isolation scheme," in *2022 13th Interna-*

tional Conference on Information and Communication Systems (ICICS), (Irbid, Jordan), pp. 54–58, IEEE, June 2022.

[37] Z. Guo, Z. Lv, N. Li, T. Yuan, X. Gao, and Z. Yuan, "Comprehensive defense scheme against container escape related to container management procedure," in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, (Suzhou, China), pp. 263–266, IEEE, Oct. 2022.

[38] N. Yang, C. Chen, T. Yuan, Y. Wang, X. Gu, and D. Yang, "Security hardening solution for docker container," in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, (Suzhou, China), pp. 252–257, IEEE, Oct. 2022.

[39] S. Song, S. Suneja, M. V. Le, and B. Tak, "On the Value of Sequence-Based System Call Filtering for Container Security," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, (Chicago, IL, USA), pp. 296–307, IEEE, July 2023.

[40] M. Hassan, T. Tahir, M. Farrukh, A. Naveed, A. Naeem, F. Zaffar, F. Shaon, A. Gehani, and S. Rahaman, "Evaluating Container Debloaters," in *2023 IEEE Secure Development Conference (SecDev)*, (Atlanta, GA, USA), pp. 88–98, IEEE, Oct. 2023.

[41] R. Yasrab, "Mitigating docker security issues," *arXiv preprint arXiv:1804.05039*, 2018.

[42] D. Reti and N. Becker, "Escape the fake: Introducing simulated container-escapes for honeypots," *arXiv preprint arXiv:2104.03651*, 2021.

[43] J. Pope, F. Raimondo, V. Kumar, R. McConville, R. Piechocki, G. Oikonomou, T. Pasquier, B. Luo, D. Howarth, I. Mavromatis, P. Carnelli, A. Sanchez-Mompo, T. Spyridopoulos, and A. Khan, "Container Escape Detection for Edge Devices," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, (Coimbra Portugal), pp. 532–536, ACM, Nov. 2021.

[44] J. Wenhao and L. Zheng, "Vulnerability Analysis and Security Research of

Docker Container," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 354–357, Sept. 2020.

[45] A. Semjonov, "Security analysis of user namespaces and rootless containers," Jan. 2020. Publisher: TUHH Universitätsbibliothek.

[46] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia Computer Science*, vol. 175, pp. 157–164, 2020.

[47] A. Farshteindiker and R. Puzis, "Leadership Hijacking in Docker Swarm and Its Consequences," preprint, MATHEMATICS & COMPUTER SCIENCE, May 2021.

[48] C. Abdelmassih, "Container Orchestration in Security Demanding Environments at the Swedish Police Authority,"

[49] containerd project, "containerd," 2023. Accessed: 2023-07-24.

[50] The opencontainers/runc community, "Cli tool for spawning and running containers according to the oci specification," 2023. Accessed: 2023-07-24.

[51] Google, "gVisor: Application kernel for containers." `https://gvisor.dev`, May 2018. Initial release 2 May 2018.

[52] The Kata Containers community, "Kata containers - open source container runtime software," 2023. Accessed: 2023-07-24.

[53] Cloud Native Computing Foundation, "Cri-o - lightweight container runtime for kubernetes," 2023. Accessed: 2023-07-24.

[54] "Linux Containers." `https://linuxcontainers.org/`, 2023. Accessed: 2023-07-24.

[55] I. CoreOS, "rkt: The pod-native container engine." `https://github.com/rkt/rkt`, 2023. Accessed: 2023-07-24.

[56] I. Syslabs, "Singularity container services." `https://sylabs.io/`, 2023. Accessed: 2023-07-24.

[57] "GitHub - facebookincubator/nvdtools: A set of tools to work with the feeds (vulnerabilities, CPE dictionary etc.) distributed by National Vulnerability Database (NVD) — github.com." https://github.com/facebookincubator/nvdtools. [Accessed 03-04-2024].

[58] "National vulnerability database." https://nvd.nist.gov/. Accessed: 2023-07-24.

[59] T. L. Foundation, "Cloud native computing foundation." https://www.cncf.io/, 2023. Accessed: 2023-07-24.

[60] T. L. Foundation, "Open container initiative." https://opencontainers.org/, 2023. Accessed: 2023-07-24.

[61] Trickest, "Gather and update all available and newest cves with their poc." https://www.cncf.io/, 2023. Accessed: 2023-07-24.

[62] "Github." https://www.github.com/. Accessed: 2023-07-24.

[63] Metarget, "Metarget: A framework for automatic constructions of vulnerable infrastructures." https://github.com/Metarget/metarget, 2024.

[64] NVD, "CVE-2019-5736." https://nvd.nist.gov/vuln/detail/CVE-2019-5736, 2019. Accessed: 2023-06-24.

[65] NVD, "CVE-2020-15257." https://nvd.nist.gov/vuln/detail/CVE-2020-15257, 2020. Accessed: 2023-06-24.

[66] NVD, "CVE-2019-14271." https://nvd.nist.gov/vuln/detail/CVE-2019-14271, 2019. Accessed: 2023-06-24.

[67] NVD, "CVE-2022-0847." https://nvd.nist.gov/vuln/detail/CVE-2022-0847, 2022. Accessed: 2023-06-24.

[68] NVD, "CVE-2022-0492." https://nvd.nist.gov/vuln/detail/CVE-2022-0492, 2022. Accessed: 2023-06-24.

[69] 262588213843476, "Vagrant and VMWare Fusion 13 on Apple M1 Pro — gist.github.com." https://gist.github.com/sbailliez/2305d831ebcf56094fd432a8717bed93. [Accessed 07-04-2024].

[70] "VMware Fusion 13.5.0." Accessed: 2024-04-07.

[71] Docker, Inc., "Docker ce for ubuntu 18.04 (bionic) - arm64," 2018.

[72] Unit 42, Palo Alto Networks, "Breaking Docker via runC: Explaining CVE-2019-5736," 2019. Accessed: 2024-06-24.

[73] Twistlock, "RunC CVE-2019-5736," 2019.

[74] N. Group, "Proof of concept for cve-2020-15257 in containerd." https://github.com/nccgroup/abstractshimmer, 2020.

[75] T. Yang, "Exploits for cve-2019-14271." https://github.com/thomasyang1207/Exploits, 2019.

[76] Bento, "Ubuntu 22.04 ARM64 Vagrant Box." https://app.vagrantup.com/bento/boxes/ubuntu-22.04-arm64, 2022. Accessed: April 20, 2024.

[77] Pimlie, "Ubuntu Mainline Kernel Installer Script." https://github.com/pimlie/ubuntu-mainline-kernel.sh, 2022. Accessed: April 20, 2024.

[78] Al1ex, "Scripts for CVE-2022-0847." https://github.com/Al1ex/CVE-2022-0847, 2022. Accessed: April 20, 2024.

[79] Palo Alto Networks, "can-ctr-escape-cve-2022-0492." https://github.com/PaloAltoNetworks/can-ctr-escape-cve-2022-0492, 2022. Accessed: April 20, 2024.

[80] L. Degioanni, "Sysdig." https://sysdig.com/opensource/sysdig/, 2013. Accessed: 2024-07-24.

[81] Elastic, "Auditbeat." https://www.elastic.co/beats/auditbeat, 2018. Accessed: 2024-07-24.

[82] S. Banon, "Elasticsearch." https://www.elastic.co/elasticsearch/, 2010. Accessed: 2023-07-24.

[83] J. Sissel, "Logstash." https://www.elastic.co/logstash/, 2013. Accessed: 2023-07-24.

[84] R. Khan, “Kibana.” https://www.elastic.co/kibana/, 2013. Accessed: 2023-07-24.