# ENHANCING SECURITY IN DOCKER WEB SERVERS USING APPARMOR AND BPFTRACE

by

**Avigyan Mukherjee**

**A Thesis**

*Submitted to the Faculty of Purdue University*

*In Partial Fulfillment of the Requirements for the degree of*

**Master of Science**


Department of Computer and Information Technology

West Lafayette, Indiana

May 2023

# THE PURDUE UNIVERSITY GRADUATE SCHOOL
# STATEMENT OF COMMITTEE APPROVAL

**Dr. Baijan Yang, Chair**

School of Computer and Information Technology

**Dr. Abdul Salam**

School of Computer and Information Technology

**Dr. Deepak Nadig**

School of Computer and Information Technology

**Approved by:**

Dr. John A. Springer

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABBREVIATIONS

| | |
|---|---|
| eBPF | extended Berkeley Packet Filteer |
| CVE | Common Vulnerabilities and Enumeration |
| LSM | Linux Security Module |
| DOS | Denial of Service |
| VM | Virtual Machine |

# ABSTRACT

Dockerizing web servers has gained significant popularity due to its lightweight containerization approach, enabling rapid and efficient deployment of web services. However, the security of web server containers remains a critical concern. This study proposes a novel approach to enhance the security of Docker-based web servers using bpftrace to trace Nginx and Apache containers under attack, identifying abnormal syscalls, connections, shared library calls, and file accesses from normal ones. The gathered metrics are used to generate tailored AppArmor profiles for improved mandatory access control policies and enhanced container security. BPFtrace is a high-level tracing language allowing for real-time analysis of system events.

This research introduces an innovative method for generating AppArmor profiles by utilizing BPFtrace to monitor system alerts, creating customized security policies tailored to the specific needs of Docker-based web servers. Once the profiles are generated, the web server container is redeployed with enhanced security measures in place. This approach increases security by providing granular control and adaptability to address potential threats.

The evaluation of the proposed method is conducted using CVE's found in the open source literature affecting nginx and apache web servers that correspond to the classification system that was created. The Apache and Nginx containers was attacked with Metasploit, and benchmark tests including ltrace evaluation in accordance with existing literature were conducted. The results demonstrate the effectiveness of the proposed approach in mitigating security risks and strengthening the overall security posture of Docker-based web servers. This is achieved by limiting memcpy and memset shared library calls identified using bpftrace and applying rlimits in AppArmor to limit their rate to normal levels (as gauged during testing) and deny other harmful file accesses and syscalls.

The study's findings contribute to the growing body of knowledge on container security and offer valuable insights for practitioners aiming to develop more secure web server deployments using Docker.

# 1. INTRODUCTION

This chapter describes the relevant background, along with significance and purpose of this study. It also contains the research question and discusses the scope along with the assumptions, as well as the limitations and delimitation of this study.

Previous work on linux security have focused on apparmor profiles but not their application towards containerized webservers which are dominant in the combined cloud ecosystem currently.

This study shows how bpftrace, a powerful tracing tool using extended Berkeley Packet Filter (eBPF) technology, can generate AppArmor profiles to secure containerized nginx and Apache httpd web servers.

## 1.1 Background and motivation

Docker has revolutionized the software development industry by providing a lightweight, flexible, and efficient platform for deploying applications using containers. Due to its many advantages, Docker has become the industry standard in both the container market and the associated DevOps ecosystem (Combe et al., 2016). Web servers like nginx and Apache httpd are commonly deployed (Kithulwatta et al., 2022) in Docker environments, making it essential to ensure their security and resilience against various exploits and attacks.

The use of Docker containers has expanded the threat landscape, making them more vulnerable to attacks than traditional hosts (Zhimin et al., 2021). Despite its popularity, security remains a significant challenge for Docker, hindering the widespread adoption of containers in production environments (Combe et al., 2016) (Zhu and Gehrmann, 2022). AppArmor is a Linux Security Module (LSM) used to secure stand-alone applications, and it can be adapted for securing containerized applications as well (Rankin and Hill, 2014). However, existing AppArmor profile generators for Docker environments, such as docker-sec, have been ineffective as they do not prevent harmful attacks on containers such as the execution of cve:2013-2028 in nginx version 3.9 where fraudulent HTTP GET requests are crafted to gain illegal access to nginx's files for example. The Docker documentation's

apparmor profile also is not effective in preventing DOS or continuous avability based attacks on the webserver as it sets no rlimits (docs.docker.com, 2023).

This research aims to investigate the potential benefits of using bpftrace for the innovative generation of tailored AppArmor profiles for containerized web servers like nginx and Apache httpd, with a goal of improving their security and resilience in Docker environments. The increasing adoption of containerization technologies like Docker has revolutionized web server deployments, making them more manageable, scalable, and efficient. While Docker offers various advantages, it also introduces new security challenges that need to be addressed to protect web servers from potential threats and vulnerabilities. According to the World Wide Web Technology Survey, as of the current time, nginx is the most widely used web server with a market share of 33.4%, followed closely by Apache at 31.4% (Kithulwatta et al., 2022). For the purpose of this research, we will be focusing on testing two of the most commonly used web servers, which are nginx and Apache. These web servers have a combined market share of over 64% according to the World Wide Web Technology survey, making them highly relevant targets for security testing and analysis. Developing effective security mechanisms for containerized web servers, such as through novel Apparmor profile generation techniques, is therefore essential for enhancing their security and reliability.

### 1.2 Research Problem

The problem addressed in this research is:

How can we improve the security and resilience of AppArmor profiles for containerized web servers, such as nginx and Apache against various exploits and attacks in Docker environments?

### 1.3 Research Question

This study addresses the following research question:

- Can bpftrace generate more restricted AppArmor profiles and improve web server security compared to docker-sec?

## 1.4 Significance

This research is significant because it addresses the limitations of existing AppArmor profile generators and the need for tailored security solutions for containerized web servers. By developing and evaluating more efficient AppArmor profiles for nginx and Apache httpd web servers in Docker environments, this study aims to contribute to the development of best practices for securing web servers, reducing the risk of security breaches, ensuring the integrity and availability of digital content and services, and encouraging the broader adoption of containerization technologies.

- Businesses need Docker web server security.
- Docker containers are more susceptible to additional threats.
- Security is the biggest obstacle to containerization.
- AppArmor can safeguard standalone Ubuntu programs but manually setting rules for containerized web servers is not properly documented in the literature.
- According to the literature, AppArmor profile generators like docker-sec do not block assaults.
- AppArmor profiles secure Docker containers, but it lacks web server specific security for repeated disruptive attacks.

## 1.5 Scope

The scope of this research is focused on the innovative use of bpftrace to identify gaps in existing profiles and develop tailored AppArmor profiles for containerized web servers like nginx and Apache httpd, aiming to improve the security of these web servers in Docker environments.

- Limited to only apparmor profiles and doesnt cover other MAC security systems such as SELinux.

- Covers nginx and apache httpd web server container deployments and their testing is done with Metasploit and owasp zap.

## 1.6 Assumptions

- Limiting and restricting memcpy and memset shared library calls and restricting syscalls such as recv() would improve the security of apparmor profiles which would be used to deploy docker web servers with.
- The research assumes that reconnaissance is an essential step at the beginning of most attack scenarios and that mitigating it will significantly improve security.

## 1.7 Limitations

The limitations for this study include:

- The research is limited to the development and evaluation of AppArmor profiles for specific web servers (nginx and Apache httpd) .
- The full functionality of the nginx and apache containers was not exhaustively tested and only index.html and access to the conf files were used in testing and evaluation.
- The evaluation of AppArmor profiles is conducted using available tools and methods, which may not capture all aspects of security or potential attack vectors.

## 1.8 Delimitations

The delimitations for this study include:

- The research does not cover other LSMs like SELinux or alternative containerization technologies.
- The research does not include a comprehensive analysis of all possible security vulnerabilities or exploits targeting containerized web servers.
- Focuses on only internal container to container attacks.

## 1.9 Summary

This research aims to address the limitations of existing AppArmor profile generators and the need for tailored security solutions for containerized web servers by innovatively using bpftrace to develop more efficient AppArmor profiles for container-based web servers. The focus is on web servers like nginx and Apache httpd in Docker environments. The scope, assumptions, limitations, and delimitations are outlined to provide context and clarify the research boundaries.

# 2. LITERATURE REVIEW

In this chapter, we will review relevant literature in the following areas: virtualization, containerization, Docker overview, container attack classifications, Linux security with a focus on AppArmor, tracing tools such as BPFtrace, and existing frameworks for container security in web server deployments.

## 2.1 Virtualization and Containerization

Virtualization has grown rapidly in recent years. Thus, a safe, efficient virtualization system is needed (Tomar et al., 2020). Docker outperforms virtual servers in terms of speed overhead, but it still falls short in security (Loukidis-Andreou et al., 2018). This is the main impediment in its adoption. In web server container security, automatic security hardening profile generators learn from container usage to create suitable security profiles. None of the linked research has a high success rate, and most don't show security results.

Due to its scalability, and availability, Docker deploys popular web servers (Bao, 2023). AppArmor and BPFtrace profiles for container relocation improve Docker-based web server security in this thesis. The literature study will cover Docker's design, security, and challenges. We will evaluate web server container security methods like automatic security hardening profile generators. We will also cover Linux Security Modules, particularly AppArmor, and BPFtrace, which is crucial to our novel strategy.

### 2.1.1 Containerization and the Cloud

Most firms today use one cloud service (Kelly et al., 2021). End user spending on public clouds rose 20% to $592 billion in 2023, according to Gartner.

VMware and Microsoft Hyper-V, which virtualize a full operating system layer, have low resource utilization rates. Docker, an open-source containerization technology project, was released in March 2013 to provide a more efficient and lightweight virtualization option.

Containers are mostly used for microservices and the cloud. IBM predicts that 59 percent of business apps will use microservices in the next few years, increasing container use.

Cloud providers offer practical and financial benefits, but a high amount of personal and private data raises security issues, according to Kelly et al., 2021). It would require upgrading their IT systems to use a private or public cloud (or both). This allows threat actors to start availability-based attacks like Denial of Service (DOS) or use network scanners to automate attacks and abuse cloud instance misconfigurations. Deployments of virtualization technologies vary by company (Saraiva de Sousa et al., 2019) but due to its speed, scalability, and modularity, Nginx is often used in containerized settings.

Containerization allows devs to simply build and launch numerous nginx instances in isolated environments. Modularity and isolation ease nginx web server control and improve resource usage, fault tolerance, and load balancing.

This study investigates novel AppArmor profile generation methods using bpftrace scripts to secure containerized web servers like Nginx and Apache httpd, which are becoming more popular. The study will compare the proposed mechanism to current methods for performance metrics and efficacy, giving useful insights for securing containerized web servers in the changing IT ecosystem. In modern tech, virtualisation and containerization are essential. Both methods optimize resource use and isolate program deployment. To protect deployed apps, these platforms also have security issues. AppArmor and BPFtrace will be used to secure Docker-based web servers in our study.

### 2.1.2 Virtualization Technologies

Datacenters run the Internet and cloud. They use virtualization extensively. Cloud computing's benefits exceed its cons. Virtualization and cloud computing have grown rapidly in the IT business. Containers and virtualization software are used in the industry to run apps and save CPU resources (Kaur et al., 2022a).

Virtualization divides a computer into many virtual computers that can conduct their own tasks in near-isolation. It shares numerous clones of a physical instance. Virtualizing systems, files, networks, etc. Cloud server virtualisation is crucial. Virtualization technologies allow numerous operating system (OS) versions to run on the same machine by abstracting hardware from software. Virtualization hypervisors manage resources and isolate versions.

Bare-metal hypervisors run directly on system hardware, while hosted hypervisors run on top of an OS.

The hypervisor handles hardware details and allocates resources to virtual machines based on needs (Goniwada, 2022).

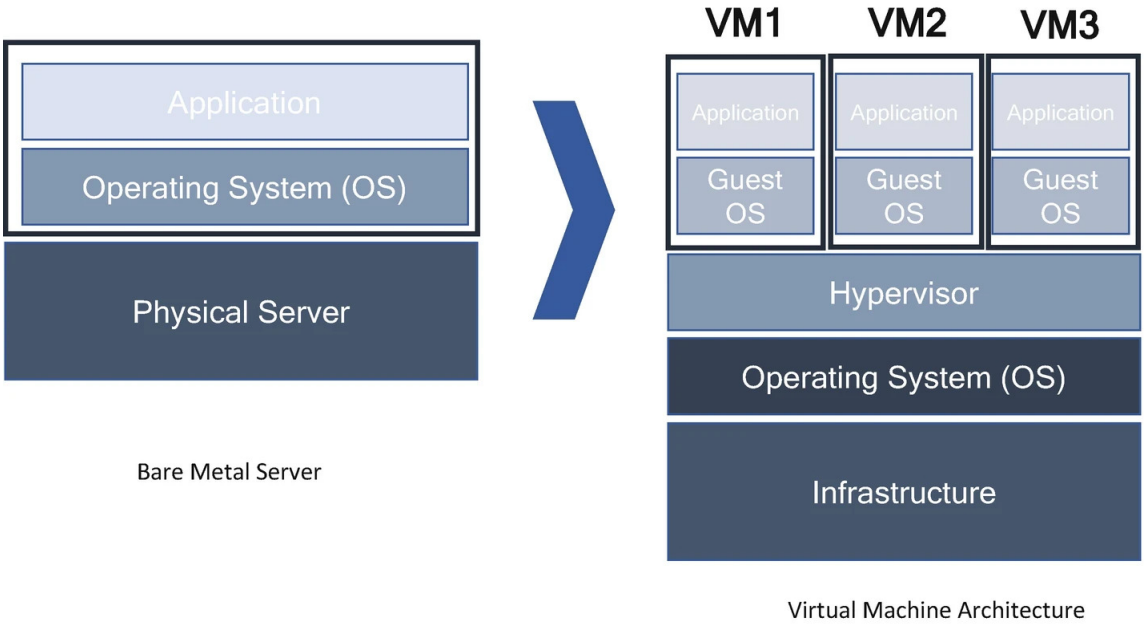


**Figure 2.1.** Cloud Virtualization

VMware's ESX/ESXi, Microsoft's Hyper-V, Oracle's VM Virtual Box, Red Hat Enterprise Virtualization, and Linux's KVM.

Traditional VM deployment is slow because each VM uses its own OS. This limits its use in HPC environments. HPC benchmarks are CPU- and data-intensive.

VMs did 42% worse for CPU-intensive tasks, 15% worse for memory-intensive tasks, and nearly 50% worse for network-intensive metrics. The emulation platform's cost prevents HPC use. A container-based environment was almost as efficient as a raw metal system in all areas (Bhardwaj and Krishna, 2021).

Virtualized settings require strong isolation and expanded attack surfaces. Isolation is important because vulnerabilities in one OS instance can impact others on the same system. In virtualized settings, required access limits and improved isolation can be provided by Linux Security Modules (LSMs) like AppArmor.

In conclusion, virtualization technologies enable resource efficiency and program deployment in isolated settings. Despite their popularity, virtual machines (VMs) have downsides like higher overhead and lower efficiency. Containerization is gaining popularity as a lightweight, agile computing option. Containerization and Docker, a famous containerization platform, handle some of the limitations of VM-based virtualization while bringing new security challenges and possibilities.

### 2.1.3 Containerization and Docker

Containerization, a form of virtualization, enables users to package and distribute images along with their binaries and dependencies. These images are then utilized to initiate containerized services, such as web servers. By abstracting services from their images, containerization simplifies and streamlines the deployment of cloud-based services across private, public, and hybrid environments. It allows engineers to concentrate on business logic and dependencies (Goniwada, 2022).

The rapid adoption of container technology has been observed in the literature, with major organizations embracing containers at a faster pace than anticipated. By 2022, it was estimated that 75% of global businesses would run more than two containerized applications, representing a 30% increase compared to 2019 (Kaiser et al., 2022).

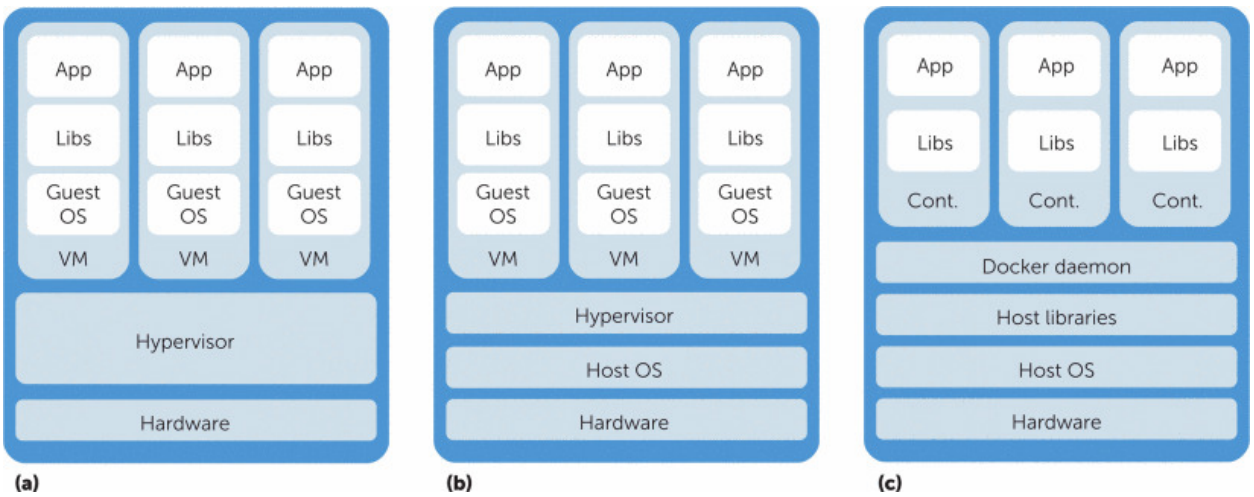


**Figure 2.2.** Hypervisor v Container

Containers share the host OS, libraries, and binaries, making them significantly more lightweight than VMs. By pooling resources, containers enhance hardware utilization in multi-tenant systems. Unlike VMs, containers rely on the host system without requiring a separate kernel or virtual hardware layer. This design reduces syscall execution and enables containers to share software tools, such as libraries, with the host, minimizing code duplication. As a result, containers can load quickly, with image sizes often reduced to a few megabytes. However, their lightweight nature also makes containers more susceptible to various security vulnerabilities.

Docker is a popular containerization platform that enables agile and lightweight application deployment. Although Docker's use of a shared host kernel provides scalability, flexibility, and security, it can also propagate vulnerabilities across containers if a single weakness is exploited.

Various container platforms, including Docker, LXC, LXD, Rocket, Warden, and OpenVZ, have been compared in terms of performance overhead and resource utilization. In a study conducted by Felter, KVM and Docker were used to create virtualized systems, and the Linpack benchmark was employed to measure CPU speed by solving linear equations and calculating FLOPS Morabito et al., 2015).

In the same study, Docker and LXC outperformed KVM in most metrics when assessed using CPU-intensive benchmarks (Y-cruncher and NBENCH), network-intensive benchmarks (Bonnie++, Netperf), and disk-write (dd test). Additionally, (Preeth et al., 2015) evaluated Docker container file system efficiency using Bonnie++ and monitored CPU and memory usage with psutil and custom Python code. But this evaluation has been used in the literature to compare performance, rather than its security.

Docker has gained a reputation as the market's leading containerization technology due to its mobility, scalability, and speed, often referred to as the "most recognizable name in containerization" Leahy and Thorpe, 2022). Docker's resource efficiency, lightweight virtualization design, and overall performance make it a more suitable option for HPC applications than hypervisor-based solutions. However, container-based virtualization also presents security challenges, as a single vulnerability in the shared host kernel can be exploited (Bhardwaj and Krishna, 2021).

In light of these security concerns, my research aims to secure Docker-based web servers using AppArmor and BPFtrace. BPFtrace serves as a dynamic tracing tool for kernel and user-space program behavior, while AppArmor enforces mandatory access controls. Together, these tools are intended to enhance the security of Docker-based web server configurations. Subsequent sections will delve deeper into Docker security and its implications for your research.

## 2.2 Docker Overview

Docker consists of several components, including Docker Client and Server, Images, Registries, and Containers (Rad et al., 2017).

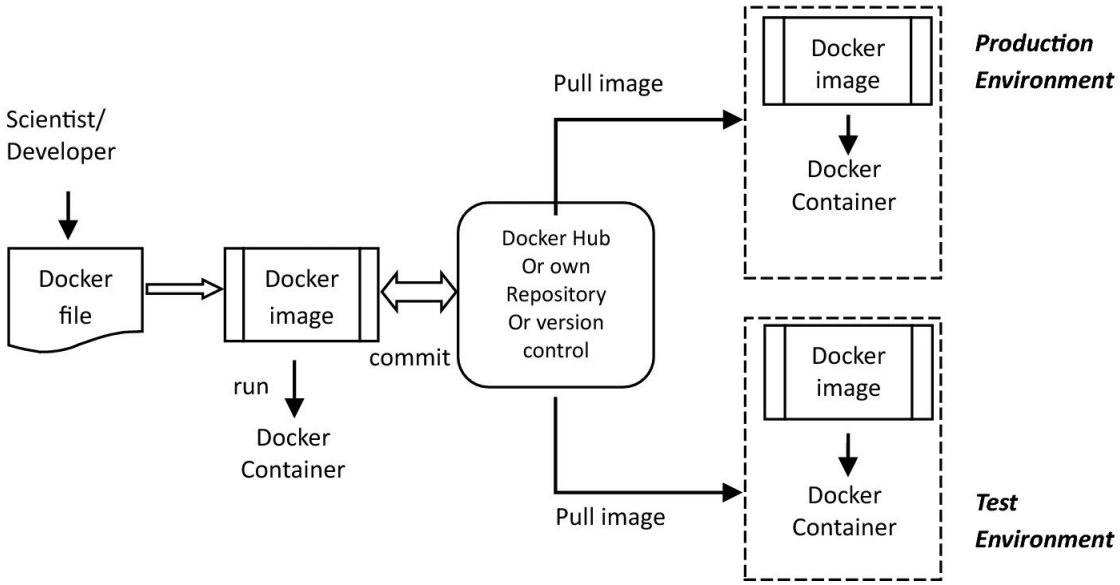


**Figure 2.3.** Docker Application Life-cycle (Bentaleb et al., 2022)

### 2.2.1 Docker Client and Server

Docker's client-server architecture connects client containers to the Docker daemon on the host computer. The Docker daemon is responsible for building, running, and managing containers. The daemon and container can be deployed locally or remotely, with the network bridge Docker0 connecting the server and containers. Docker Client interacts with the Docker

Daemon to manage Docker operations and communicates via a UNIX port or RESTful API (Wenhao and Zheng, 2020).

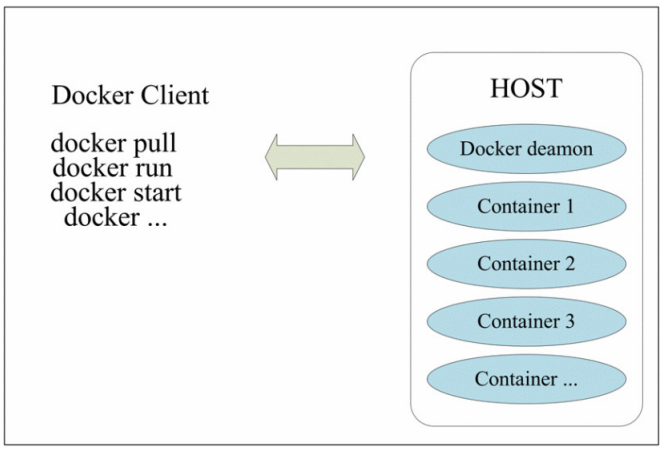


**Figure 2.4.** Docker Schema (Wenhao and Zheng, 2020)

### 2.2.2 Docker Images

Dockerfiles enable users to define a base image and build commands. Using the "Docker build" command in the bash terminal, an image is built following the Dockerfile (Wenhao and Zheng, 2020).

Docker images consist of JSON metadata layers and are stored at /var/lib/docker/<driver>/, where the driver is the storage driver (e.g., aufs, Btrfs, VFS, device mapper, or OverlayFS) (Martin et al., 2018). From a base copy, each layer modifies the filesystem, with images organized in trees, except for the base image. This structure allows Docker to ship only the image changes linked to it (Combe et al., 2016).

Docker containers require secure base files. AppArmor and BPFtrace can be employed to identify security flaws in Docker images. AppArmor restricts processes running in containers, while BPFtrace monitors and analyzes system calls made by containerized apps to uncover security risks.

### 2.2.3 Docker Registry

Registries store Docker files and function similarly to source code repositories, allowing users to publish and retrieve images. Both public and private registries exist, with Docker Hub serving as a public registry that enables users to download and upload images freely. Docker Hub's features support the proper distribution of images (Wenhao and Zheng, 2020).

Securing Docker registries ensures the protection of pushed and pulled files. AppArmor and BPFtrace can be utilized to monitor and analyze registry actions. AppArmor policies can limit unauthorized registry access, while BPFtrace can audit registry interactions to detect anomalies and security breaches.

### 2.2.4 Docker Containers

Docker images give rise to containers, which isolate programs by containing the application kit. For example, an Ubuntu OS image with nodeJS can be launched using docker run to create a container and run the nodeJS server on Ubuntu (Wenhao and Zheng, 2020).

Docker containers offer advantages but also pose risks. In this research, AppArmor enforces mandatory access controls on containers to secure application execution, while BPFtrace analyzes container kernel and user-space application activity. This combination aims to identify and mitigate security vulnerabilities in Docker-based web servers.

## 2.3 Webservers

Web servers receive requests from web browsers and return web pages and HTML documents (Nurwarsito and Sejahtera, 2020). With today's internet architecture relying heavily on web servers, the need for a reliable and widely available web server infrastructure is essential. Web server clustering can increase stability and availability for high-traffic online apps. Docker containers can run popular web servers like nginx and Apache httpd.

Web servers play a crucial role in delivering digital content and services to users worldwide (Jader et al., 2019). Nginx and Apache are increasingly popular, making secure and scalable

deployment choices essential (Sultan et al., 2019). This study aims to use AppArmor and BPFtrace to secure Docker-based web servers such as Nginx and Apache.

Due to their portability and scalability, containerized web servers are becoming more popular. Nginx and Apache httpd are two widely used containerized web servers.

### 2.3.1 nginx Web Server

Nginx is a high-performance web server, reverse proxy, and load manager. Initially designed to address the C10K problemefficiently managing 10,000 simultaneous connectionsnginx is known for its event-driven design, high concurrency, and low memory usage, making it suitable for containerized web application hosting.

The official nginx Docker image on Docker Hub can run nginx as a containerized web server. This simplifies nginx setup on Docker-supported systems.

### 2.3.2 apache httpd Web Server

Apache Software Foundation's open-source web server and servlet container is Apache httpd. It is also used to serve http web pages.

Similar to nginx, Apache httpd's Docker Hub image can be used to containerize it, simplifying httpd setup and management on Docker-supported systems.

## 2.4 Security of Docker

Containers run on the host system, unlike VMs, which allows them to operate nearly as efficiently as a native OS implementation (Kaur et al., 2022b). However, security concerns such as resource utilization and privilege escalation still hinder containerization technology from reaching its full potential.

Security in a host-container setup involves four use cases: defending containers from malicious apps, protecting containers from each other, safeguarding the host, and securing hosts that may be malicious or semi-malicious (Sultan et al., 2019).

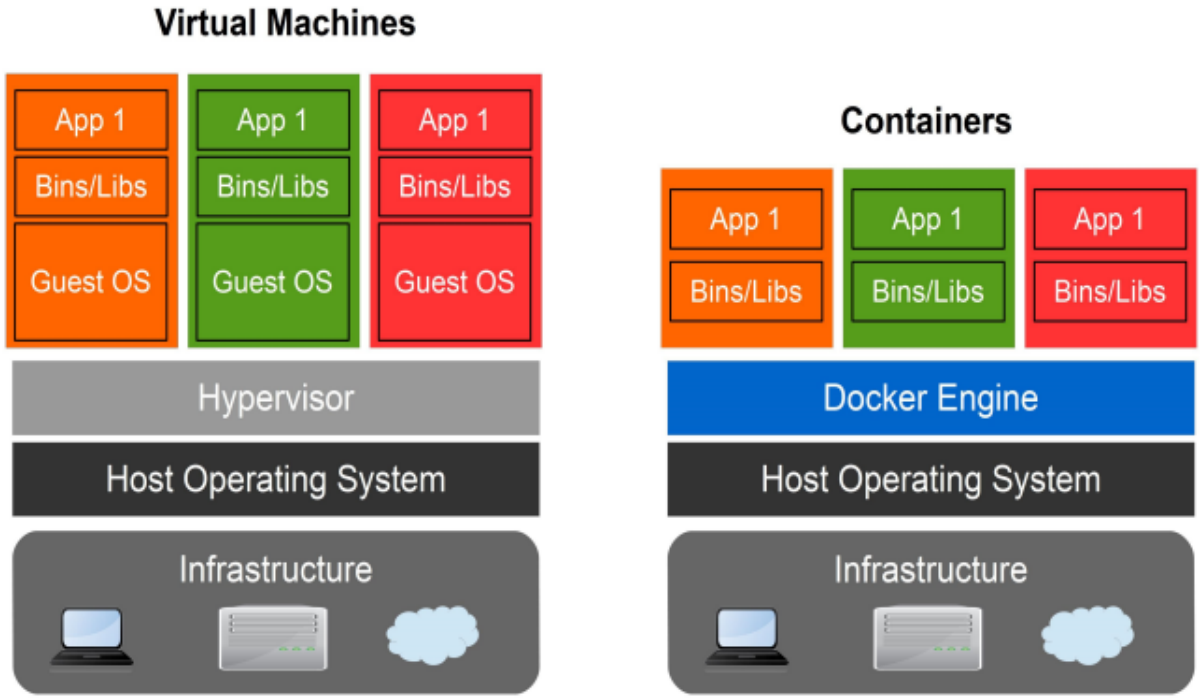


**Figure 2.5.** VMs and Docker

Exploits can be categorized basis of Bypassing protection mechanisms, gaining illegal privileges, DoS attacks, reading application data, and executing unauthorized code or commands (Zhu and Gehrmann, 2021).

Docker employs cgroups, namespaces, seccomp, capabilities, and Mandatory Access Control of Linux Security Modules (SELinux, AppArmor) to isolate and limit resources. Nevertheless, cgroups have been exploited, failing to limit resource usage in containers (Gao et al., 2019), and even with all security features enabled, 50% of all exploits in a native Linux system succeed in the Docker container (Lin et al., 2018).

For containerized web servers like nginx and Apache httpd, AppArmor, a Linux Security Module (LSM) that provides mandatory access control, helps secure Docker containers. Manually applying AppArmor policies to containerized web servers is complex and requires an understanding of the web server application's features (Loukidis-Andreou et al., 2018).

AppArmor focuses on controlling process access rather than overall system security. Ubuntu systems commonly utilize AppArmor.

The Docker documentation's nginx AppArmor configuration may not be comprehensive enough to secure containerized web servers. This research aims to develop tailored AppArmor profiles for web servers like nginx and Apache httpd to address potential vulnerabilities in generic profiles. However, directly evaluating their security is not feasible.

The primary goal of this research is to enhance security against a wide range of exploits and attacks by creating tailored AppArmor profiles with metrics identified using bpftrace for containerized web servers and using ltrace to compare their restrictiveness. This could improve web server deployments in Docker environments, helping organizations protect their online assets and services.

### 2.5 Enhancing Security for Docker Deployments

In previous research, various aspects of Docker security have been explored. Studies have examined Docker container internal security, kernel extra security methods, and Linux hardening for container security. These works have touched upon the use of SELinux, AppArmor, and other Linux Security Modules (LSMs) to enhance security.

Some researchers have also investigated Docker container DDoS attacks, proposing ways to secure containers, such as using Access Control Policy Modules, downloading images from authenticated sources, and implementing SELinux/AppArmor. However, these studies do not thoroughly address the issue of availability attack protection and fail to account for potential vulnerabilities in cgroup implementation.

In contrast to previous works, recent research has aimed to develop more comprehensive security measures, such as a new infrastructure-level container security taxonomy, automatic VirusTotal scanning of Docker images, and Zero Trust Network platforms for Docker container operations.

Our research aims to build upon these existing studies by focusing on the development of tailored AppArmor profiles for containerized web servers like nginx and Apache httpd.

By using auditd to compare the efficiency of our approach with existing methods, we hope to improve security against a wide range of exploits and attacks.

By enhancing web server deployments in Docker environments, our research will contribute to helping organizations better protect their online assets and services. Our work will address potential gaps and vulnerabilities in generic profiles, providing more secure deployment options for popular web servers.

Bui, 2015 was one of the first to analyse the security concerns of Docker containers. The study looks at two aspects: the internal security of Docker containers and how Docker containers interact with the kernel's extra security mechanisms.

MP et al., 2016 corroborate about linux hardening methods for docker container security and provides a review of its measures. They are digitally signed and verified images, using docker with SELinux Modules or Apparmor LSMs and also talks about LicShield, which is an automated Apparmour profile generator. It recommends to not run containers on privileged mode by setting the –security opt flag, and use with SELinux and Apparmour hardening. It also makes a list of all the capabilities which are "safe" to drop as shown in table 2.1.

But, this list of capabilities cause web servers to creash as they would need access to most of the capabilities mentioned above.

Müller and Pieters (2016) review Linux hardening for Docker container security and discuss using digitally signed and verified files with SELinux Modules or AppArmor LSMs. They also mention LicShield, an automatic AppArmor profile generator, and suggest using SELinux and AppArmor hardening with the –security-opt flag to avoid running containers in privileged mode.

Chelladhurai et al., 2016 were the first to examine Docker container DDoS attacks. They use the kernel security's cgroups tool to set memory bounds by employing the command "docker run -it -m=512mb Ubuntu:latest /bin/bash". This method is unreliable as it is not specific to web server traffic.

Combe et al., 2016 evaluates Docker security and proposes leveraging Linux internal security modules for Docker container protection, but as we will see this are not enough in the current environoment.

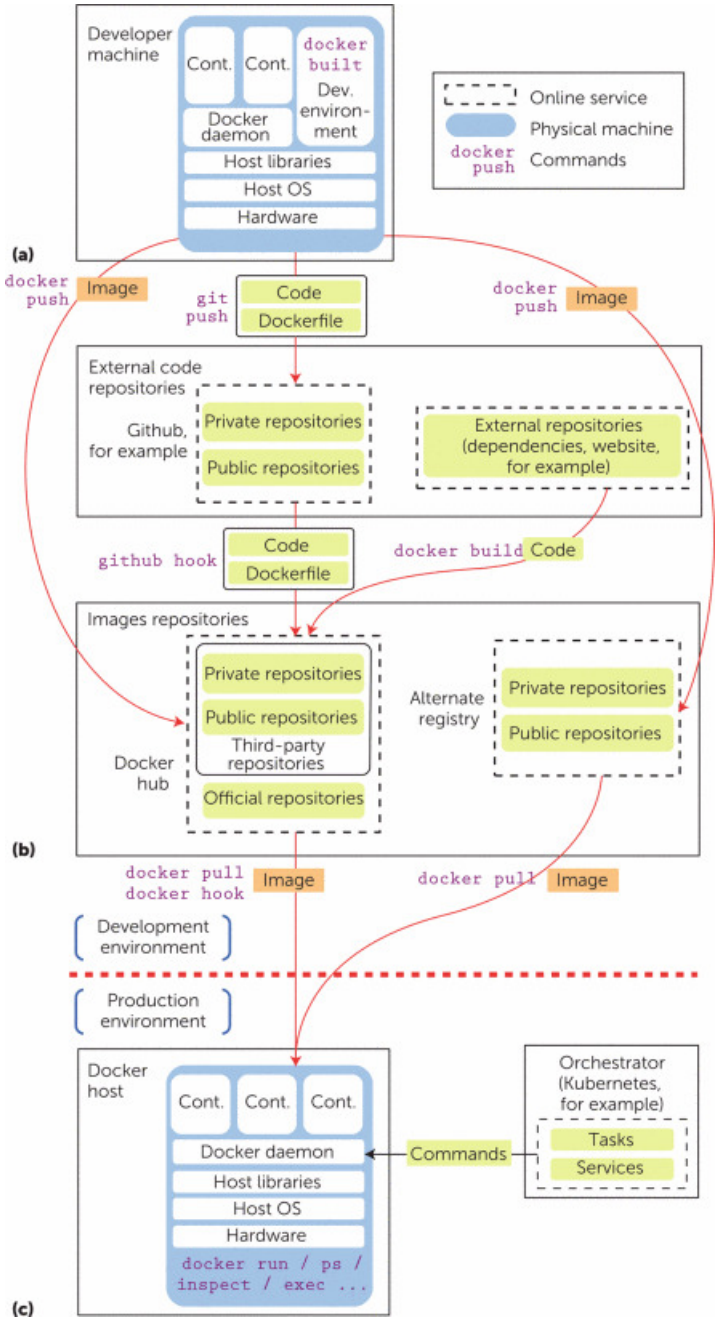


**Figure 2.6.** Docker Ecosystem (Martin et al., 2018)

Manu et al., 2016 conducts a in depth exploration into cloud Platform as a Service (PaaS) security, and compared other Linux containerization technologies to docker in terms of security and find similar results for their comparisons.

Martin et al., 2018 provides a deep dive in the Docker ecosystem (figure 2.8) and its vulnerabilities at all points in the ecosystem. It both provides an overview of docker internal security services along with the docker hub automated pipeline architecture and its use cases. It further uses those use cases to draw up a relative vulnerability assessment scale against Docker and its ecosystem. Findings related to vulnerability assessment is shown in the table below.

**Table 2.1.** Vulnerability Assessment

| Vulnerability categories | Docker use-cases | Widely used-cases | CaaS use-cases |
|---|---|---|---|
| Insecure Configuration | Moderate | Very high | High |
| Image distribution, verification, decompression and storage process | Very High | Moderate | High |
| Inside the images | Moderate | Very High | Moderate |
| Directly linked to Docker or libcontainer | Similar level across use-cases | N.A | N.A |
| In the kernel | Similar level across use-cases | N.A | N.A |

Yasrab, 2018 analyzes Docker and suggest ways to secure containers. Their recommendations include using Access Control Policy Modules to limit container freedom, downloading images from authenticated sources supporting cryptographic signatures, utilizing SELinux/AppArmor, and logging and auditing. They do not, however, account for availability attack protection and cgroup houdining (Gao et al., 2019).

Bélair et al., 2019 provides a new taxonomy of container security related to the infrastructure level in comparison to previous works and classify defense frameworks based on those taxonomy. They classify the methods by which the host OS can improve the security of the container.

This way of classification was broadly subjective and does not reflect attacks as they take place in real world scenarios. We have hence selected the CVE classification method

proposed by Zhu and Gehrmann, 2021 and other subsequent research to base our attack testing set's classification - namely Bypass, gain priviledge, dos, execute code.

Brady et al., 2020 proposes an automated solution for Docker in the cloud similar to the current DSS, to address image distribution and image production vulnerabilities by proposing two new CI/CD pipelines and automated scanning of the images with VirusTotal as shown in figures below.

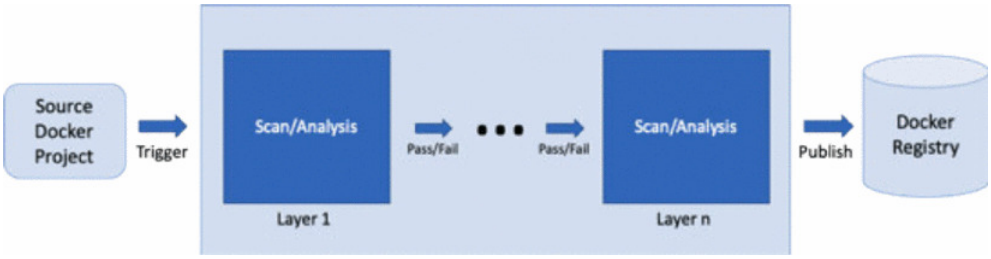


**Figure 2.7.** CI/CD pipeline (Brady et al., 2020)

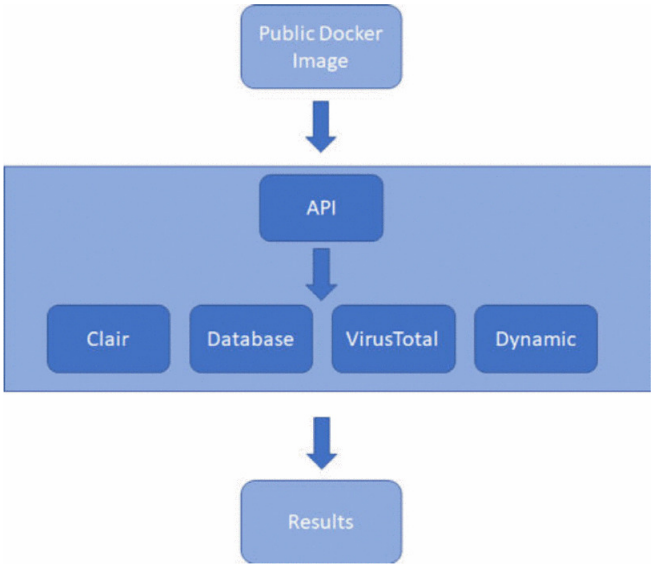


**Figure 2.8.** Scanning images (Brady et al., 2020)

Wenhao and Zheng, 2020 conducts a thorough review of vulnerability assessment of docker containers and brings forth its internal security, and also MAC (Mandatory Access Control), and other security features related to network isolation, seccomp, etc.

Tomar et al., 2020 presents a threat model for Denial Of Service attacks in docker containers, and also proposes an attack taxonomy as shown in figures. It classifies attacks using attack layers and then maps the different modes of attacks to them.

Jain et al., 2022 audits all the previously provided taxonomies on Docker security mechanisms and technologies. It conducts an extensive File and Directories Audit with Linux Audit Daemon which allows

- Auditing file accesses and modifications
- Monitoring syscalls
- Detecting intrusions in the system
- Logging commands entered by users

Wist et al., 2021 scans around 2500 DockerHub images and reports that more than 80% of public certified images still have vulnerabilities in them. The Common Vulnerability Scoring System (CVSS) is used to mark vulnerabilities as critical, high or medium, etc.

Leahy and Thorpe, 2022 presents a Zero trust Network framework to apply to docker container deployments to increase their security.

In later subsections, we will see overviews of the security features Docker utilizes and leverages and work done related to them.

### 2.5.1 Internal Security Measures: Safeguarding Containerized Applications

Limiting the attack surface and isolating instances on the Docker host is crucial. Internal security measures include updating the Docker host and tools, restricting Docker API access, segmenting and isolating networks, and vulnerability scanning of images. Trusted base images and digital signatures also play an essential role in securing containerized applications.

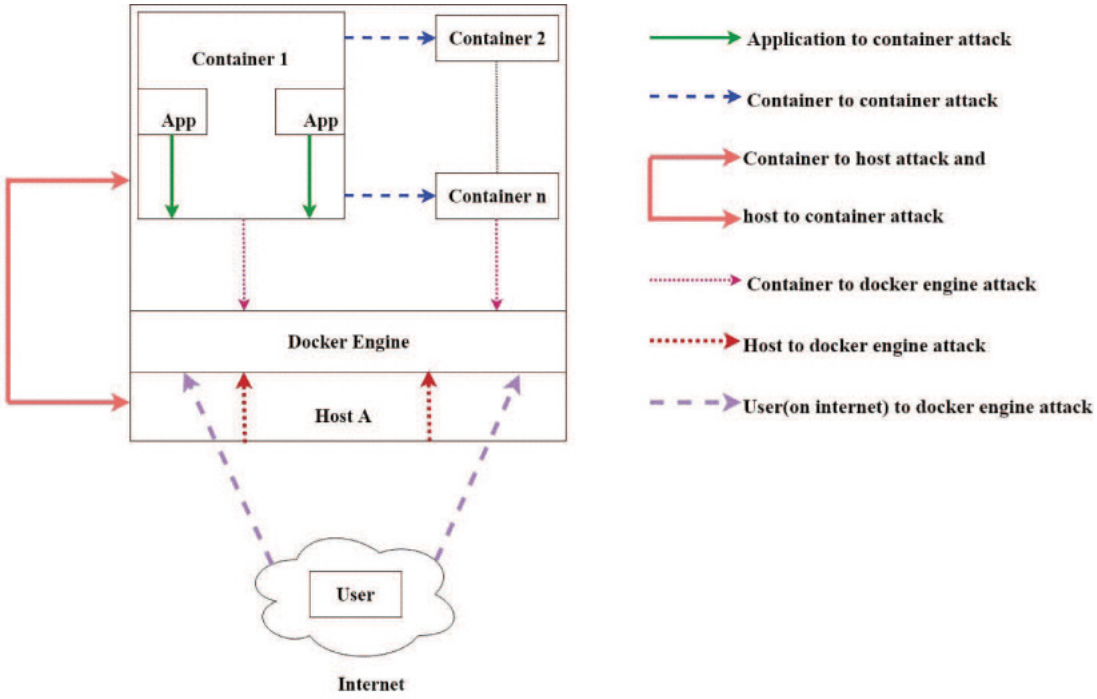


**Figure 2.9.** Threat Model (Tomar et al., 2020)

Docker secures its containers using Linux namespaces and cgroups, which are vital components of its security model. Namespaces separate operating system resources, and control groups (cgroups) limit each container's programs' resource access, helping prevent DoS attacks.

### Enhancing Security with AppArmor Profiles

Our research aims to develop tailored AppArmor profiles for containerized web servers like nginx and Apache httpd, focusing on improving security against a wide range of exploits and attacks. By using ltrace to compare the efficiency of our approach with existing methods, we hope to enhance web server deployments in Docker environments.

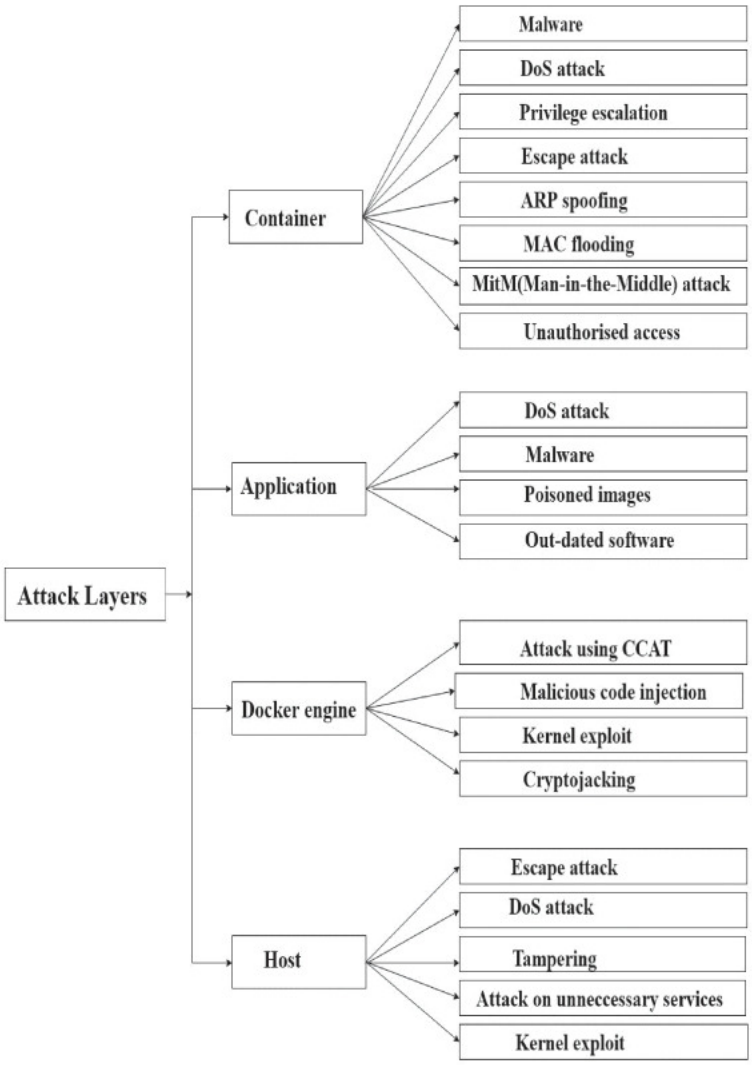


**Figure 2.10.** Attack Taxonomy (Tomar et al., 2020)

### 2.5.2 Capabilities: Restricting Docker Container Privileges

Docker assigns container rights using Linux's capabilities permissions module. Flaws in this system can be exploited to escape containers and target the Docker Service or host. Linux enables Docker administrators to grant or limit container privileges, ensuring that containers only have the necessary permissions. The capability mechanism divides Linux

permissions, allowing for more fine-grained control over container privileges (Jain et al., 2022).

In the context of our research, we will focus on refining the capability mechanism and further subdividing privileges to create more secure environments for containerized web servers. This approach will enhance the security of Docker deployments, helping organizations protect their online assets and services better.

## 2.6 Access Control: Implementing SELinux and AppArmor in Docker Environments

Most people use discretionary access control (DAC, Discretionary Access Control). This model lets object owners randomly change access mode. The object's owner is usually its creator. Windows, UNIX, Linux, and VMS use this paradigm. Linux kept Unix's dilemma that only DAC access control applies to methodically built file system objects, notwithstanding its restrictions. An ACL can't limit network interface access (Jain et al., 2022). Mandatory Access Control (MAC) is a set of access control models in which a strict access control policy decides what permissions a subject has for an object. There are many MAC models, but for this study, we will focus on modules with armor ("Overview", 2022).ă

Mandatory Access Control (MAC) frameworks like SELinux and AppArmor can be used to enhance Docker container security. These frameworks allow you to define policies that dictate the allowed actions and access permissions for containers.

### 2.6.1 SELinux

When used in a container context, the important SELinux apps are mainly TE (Type Enforcement) and MCS (Multi-Category Security).

It assigns a tag with two attributes to test subjects and objects. Hierarchical categories for the major may exist, but they are not applicable here. The second attribute can handle multivalued attributes. To identify the access modes, a predetermined policy that compared the names was used. These models mark objects and subjects for TE (Type Enforcement) (Jain et al., 2022). Unlike previous models, Multi-Category Security (MCS) is based on a

SELinux-specific implementation and is only detailed informally in a few web documents by people engaged in its development.

While CentOS uses SELinux, Ubuntu (the more popular distro), is shipped with AppArmor as it is easier to implement, which is hence focused on in this study.

### 2.6.2 AppArmor

AppArmor is another Linux kernel security module that provides a simpler approach to MAC when compared to SELinux. It uses security profiles to restrict the capabilities of individual applications or containers. Docker has built-in support for AppArmor, which means that you can create and apply AppArmor profiles to your containers without the need for additional installation or configuration.

AppArmor produces a list of resources (such as container names) for each prohibited application. Simple abstractions like dbus, kde, and nameservice are used to arrange low-level program rights and define rules (Schreuders et al., 2011). AppArmor has two modes: enforcement andălearning. The profile's rules are enforced using the enforcement mechanism. In the learning mode, however, infractions of profile restrictions are not only allowed, but also recorded. This log may subsequently be used to create new profiles (Bui, 2015).

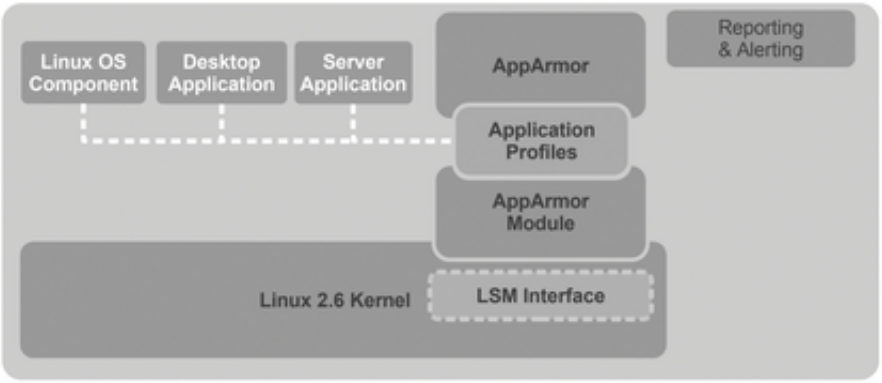


**Figure 2.11.** Apparmor in Linux 2.7 Implementation (Cowan, 2007)

There have been various attempts to automate the security of docker containers either by generating apparmor profiles, or by continuous monitoring of the container and checking whether its metrics go outside the norm (generally used to prevent DOS-type attacks).

As of 2023, Apparmor is shipped with all ubuntu distros making it the go-to for container security using profiles.

## 2.7 Enhancing Docker Web Server Security with AppArmor: Addressing the Limits of Existing Approaches

Existing frameworks utilizing Apparmor with Docker for its security utilize Mandatory Access Control (MAC) to generate rules for linux processes and by extension for containers such as Docker. While AppArmor-based approaches to enhancing Docker web server security offer promising results, existing methods still face certain limitations. In this section, we examine two AppArmor-based security mechanisms, Docker-sec and Lic-sec, and discuss their strengths and weaknesses. We will then explore potential improvements to address these limitations, specifically regarding Docker web server security.

### 2.7.1 Docker-sec: Limitations and Addressing the Research Gap

In the context of container security, Docker-sec generates container profiles using configuration and application behavior guidelines. This process combines static analysis and dynamic monitoring to create and enrich profiles for container processes throughout a specified testing duration.

Static Analysis collects container and operation data from user-provided command line arguments or Docker-generated information. This data is used to establish basic security rules and launch profiles for new containers. Docker-sec gathers essential details from command line arguments and Docker utilities, such as container volumes, user privileges, and the container's SHA256 checksum. During the container setup phase, Docker-sec may temporarily enforce an AppArmor profile before switching to the one used throughout the container's runtime.

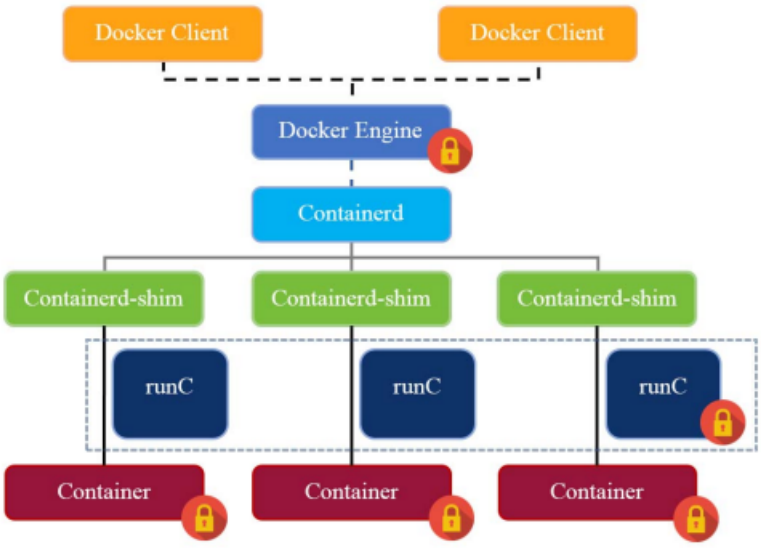


**Figure 2.12.** Things protected by Docker-sec (Loukidis-Andreou et al., 2018)

Dynamic Monitoring enables users to select a training time for a container, during which Docker-sec records behavior data. The user engages with the program's relevant section, allowing Docker-sec to identify necessary rights for successful container operation. Following the training period, Docker-sec reviews the audit log and updates the container's execution profile, potentially reducing capabilities granted by the static analysis runtime profile. The training process can be repeated until all essential functionality is documented in the container profile.

Understanding the limitations of Docker-sec is crucial for our study, as it provides a basis for exploring alternative methods, such as using bpftrace, to address these shortcomings and improve container security.

The performance overhead of docker-sec results are shown in figure 2.17 below.

Docker-sec is further extended in Kub-sec (Zhu and Gehrmann, 2022), which implements docker-sec on a kubernetes cluster and proposes a model for cloud deployment. But as we see from the figure above that the performance overhead of docker-sec also may be significantly higher in HPC environments.

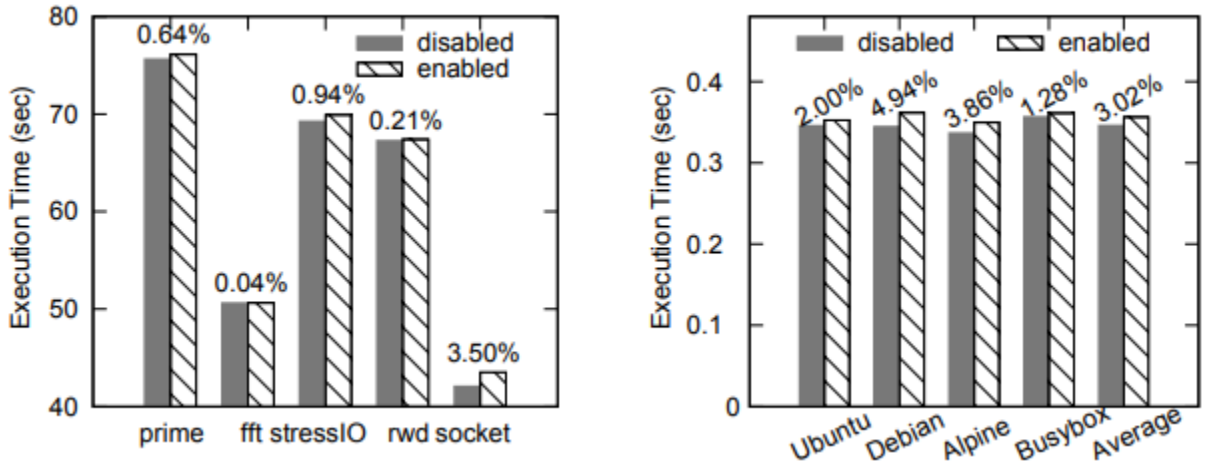


**Figure 2.13.** Performance Overhead of Docker-sec (Loukidis-Andreou et al., 2018)

While Docker-sec provides a significant level of security for containerized applications, its limitations in certain categories indicate the need for additional security measures. The success rates for various categories are as follows:

- Bypass:

  Web Application: 0%

  Server: 0%

- Gain Privilege (Inside Container):

  Docker in Docker: 0%

  IBM DB2: 0%

- Gain Privilege (Container Escape):

  Docker in Docker: 0%

  IBM DB2: 0%

- DoS (Denial of Service):

  Web Application: 0%

Server: 0%

Docker in Docker: 0%

IBM DB2: 0%

- Gain Information:

  Web Application: 0%

  Server: 0%

  Database: 0%

- Execute Code:

  Web Application: 6.67%

It can be observed that Docker-sec had limited success in preventing user space program targeted exploits, especially for web servers. The success rate for web server exploits was 0%, which means that Docker-sec was unable to prevent any of the 8 tested web server exploits. These limitations suggest that Docker-sec's protection is not comprehensive, particularly when it comes to user space program targeted exploits. As a result, there is a research gap in providing a more effective security solution for containerized web applications and servers.

By integrating eBPF and bpftrace for runtime container monitoring and using the insights gained to enhance AppArmor profiles, we can address the research gap in Docker-sec's limitations and provide a more secure environment for containerized web applications and servers.

### 2.7.2 Lic-sec:

Lic-Sec is an AppArmor profile generator that combines the strengths of Docker-sec and LiCShield, aiming to provide a more robust and comprehensive security solution for Docker containers. While it focuses on automatically constructing AppArmor profiles for each container and all Docker components, ensuring stricter confinement of privileges inside the container and supporting extensibility of functionality, it still has limitations when it comes to web-server exploits.

In the context of security of docker based web servers, it is important to recognize that Lic-Sec, despite its advancements in tracing and profile generation, does not adequately address the security challenges posed by web-server exploits. These limitations can potentially leave Docker containers vulnerable to a range of attacks specifically targeting web servers.

The security performance evaluation of Lic-Sec focuses on the most severe exploits and configurations, particularly privilege escalation attacks. Compared to Docker-sec, Lic-Sec successfully defended 8 more kernel-targeted exploits aiming to gain privilege inside containers and on the host, in the Docker container running image 'Docker in Docker' and 'IBM DB2'. For other userspace program targeted attacks, Lic-Sec has the same performance as Docker-sec.

Lic-Sec's defense principles against both privilege escalation attacks inside containers and the corresponding container escape attacks include denying the call of functions such as system() and execl(), which need to execute shell commands. Since Lic-Sec does not identify any prompted shells during the tracing, it generates the corresponding deny rules for shell prompting, effectively blocking these functions.

Lic-Sec's primary goal is to automatically construct AppArmor profiles for each container and all Docker components, aiming to achieve stricter confinement of privileges inside the container and support extensibility of functionality. However, despite its advancements in tracing and profile generation, it still falls short in addressing certain types of attacks, particularly web-server exploits.

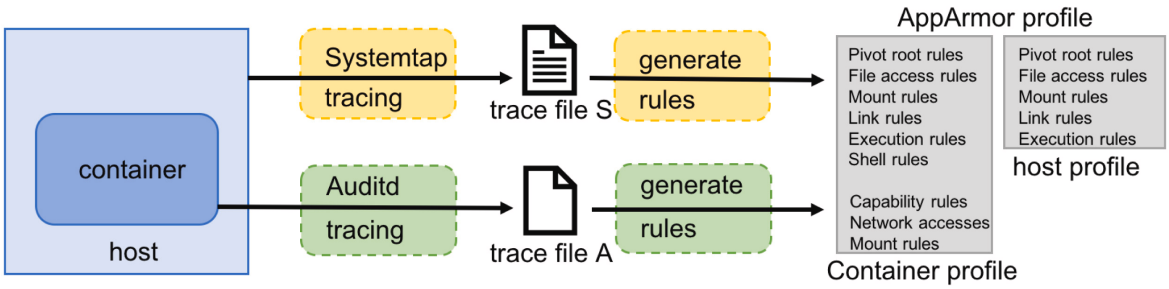


**Figure 2.14.** Lic-Sec (Zhu and Gehrmann, 2021)

The trace log structure is modified to include an 'executable process name' element for identifying processes with empty executable paths. Process tracing is allowed for processes outside the target tree, unlike in LicShield.

LiCShield's rule generator is updated to prevent privilege escalation attacks by rejecting shell prompting within containers. If no commands like '/bin/bash', '/bin/sh', or '/bin/dash' are observed during tracing, all actions to such commands are denied. The rules are further classified into container and host profiles for improved accuracy.

To automate mount rule production, the training procedure is updated, incorporating mount activity auditing along with network and capabilities. A method for creating mount rules in the container's namespace, based on log data, is also provided. For the purposes of this study, we will explore the use of bpftrace as an alternative to SystemTap and Auditd, aiming to address the limitations of Lic-Sec and enhance container security.

The results and comparison are shown below.

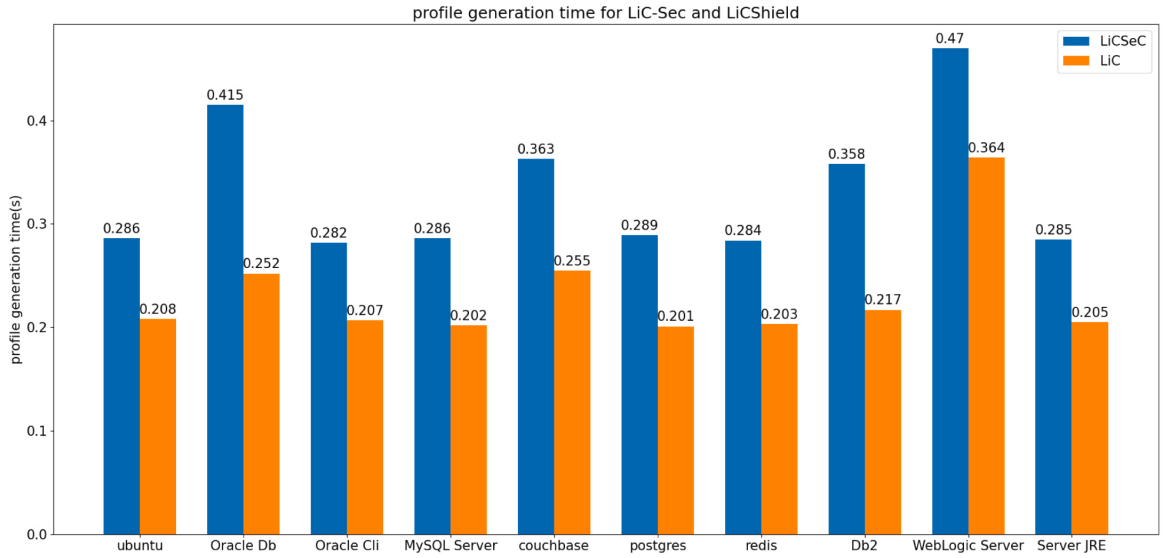


**Figure 2.15.** Profile generation time comparison between LicShield and Lic-sec (Zhu and Gehrmann, 2021)

As we can see, Docker-sec and Lic-Sec neither cater to web-server exploits (0% success rate).

| Exploits | Userspace program targeted | | | Kernel targeted | | |
|---|---|---|---|---|---|---|
| Categories | Objects | | | | | |
| | Web application (*Doc*/*Sec*/*Lic*[a]) | Server (*Doc*/*Sec*/*Lic*[a]) | Database (*Doc*/*Sec*/*Lic*[a]) | Other images (*Doc*/*Sec*/*Lic*[a]) | Docker in Docker IBM DB2 (*Doc*/*Sec*/*Lic*[a]) | Pihole/pihole[b] kylemanna/openvpn (*Doc*/*Sec*/*Lic*[a]) |
| Bypass | 4/4/4 | 3/3/3 | – | – | – | – |
| Gain privilege (Inside Container) | – | – | 2/2/2 | 4/0/4 | 4/4/4 | 4/1/4 |
| Gain privilege (Container Escape) | – | – | – | 4/0/4 | 4/4/4 | 4/1/4 |
| DoS | – | – | 1/1/1 | 1/1/1 | 1/1/1 | 1/1/1 |
| Gain information | 3/3/3 | 2/2/2 | 1/1/1 | – | – | – |
| Execute code | 13/12/13 | 9/9/9 | 2/2/2 | – | – | – |
| Total | 17/16/17 | 9/9/9 | 5/5/5 | 9/1/9 | 9/9/9 | 9/3/9 |

[a]'Doc' denotes the number of exploits executed successfully on containers launched with Docker, 'Sec' denotes the number of exploits executed successfully on containers launched with Docker-sec, 'Lic' denotes the number of exploits executed successfully on containers launched with LiCShield.
[b]'pihole/pihole' is vulnerable only when DHCP is used.

**Figure 2.16.** Successful exploits (Zhu and Gehrmann, 2021)

## 2.8 Tracing Tools for Containerized Web Server Security

Tracingpoints are points in the OS program which can be accessed by tracing tools. They allow injection of code directly in kernel space. Tracing tools help audit or log kernel level operations or operations in between user space and kernel space (example syscalls). There have been many tracing tools around such as strace, dtrace, ftrace, SystemTap, Perf and more recently eBPF (extended Berkeley Packet Filter).

eBPF causes a lower overhead in logging than Linux's auditd used in Docker-sec as well as Lic-Sec (Aich, 2021).

### 2.8.1 SystemTap

SystemTap is a 2005 tracing tool. stap is SystemTap's front-end application. It reads probing instructions in its scripting language, converts them to C code, compiles it, and installs the resulting kernel module into a running Linux kernel to perform system trace or probe operations. SystemTap has some disadvantages compared to auditd and bpftrace:

- Stability: SystemTap has a history of causing system crashes and being incompatible with legacy systems, which makes it less reliable than auditd and bpftrace.

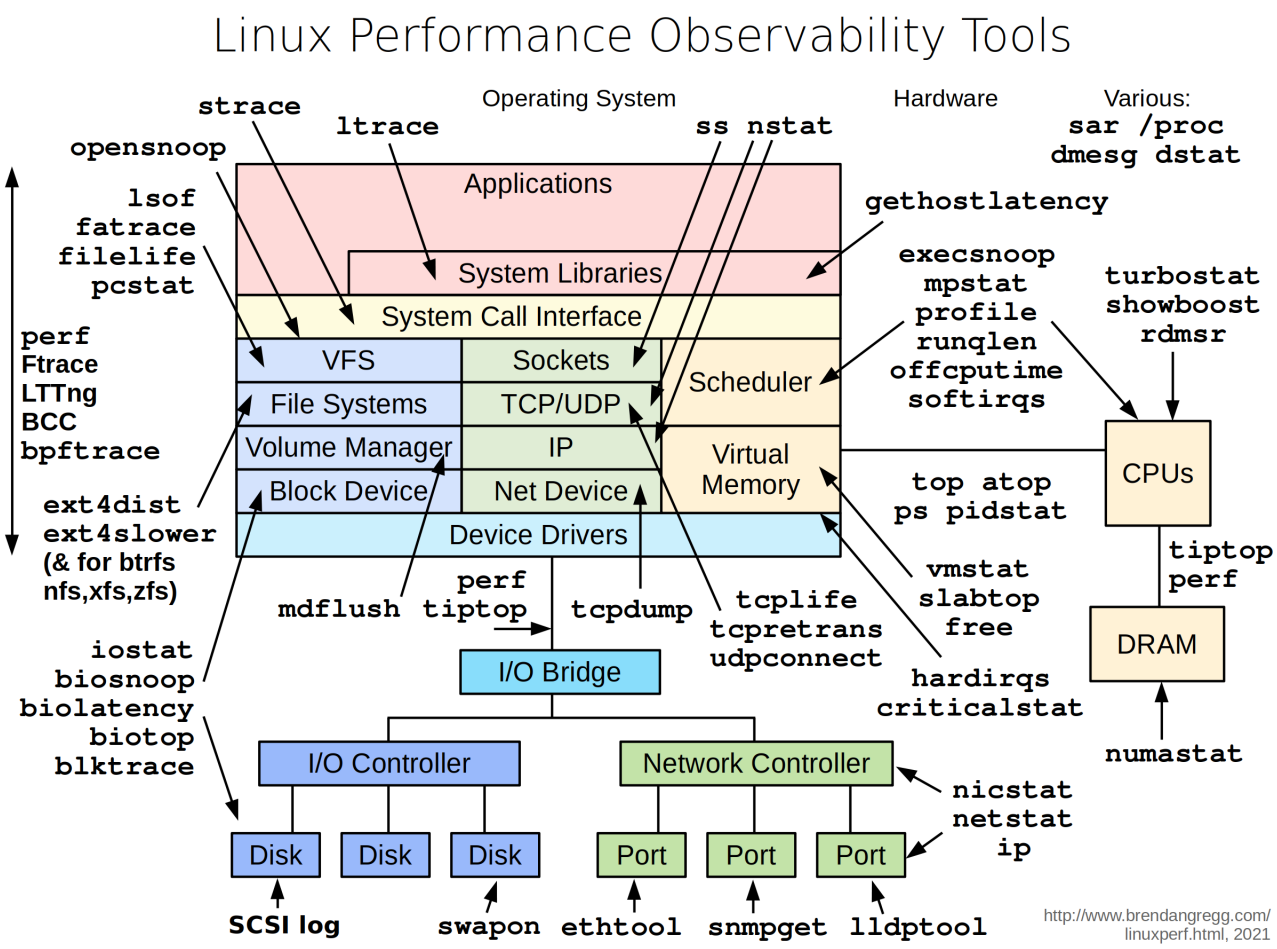


**Figure 2.17.** Linux Observability Tools (Gregg, 2015)

- Complexity: SystemTap requires users to write scripts in its own language and compile them into kernel modules, which can be more complicated and time-consuming than using auditd rules or the high-level bpftrace scripting language.

- Overhead: SystemTap can impose higher overhead on the system, especially when compared to bpftrace's eBPF-based low overhead approach.

- Flexibility: Although SystemTap can trace a wide range of kernel events, it may not be as versatile as bpftrace, which can also dynamically trace user-space events and is built on the more powerful eBPF framework.

SystemTap, a tracing tool from 2005, has several disadvantages compared to auditd and bpftrace, such as stability issues, higher complexity, increased overhead, and less flexibility. Due to these limitations, SystemTap is less suitable for containerized web server security research.

### 2.8.2 auditd

A Linux auditing system that captures system-level events for security monitoring and analysis. Primarily focused on logging system calls and file access events. Utilizes audit rules to define what events should be captured and logged. Stores audit logs on the local system or forwards them to a remote server. Provides a set of tools (such as ausearch, aureport, and autrace) for processing and analyzing audit logs. Lacks a high-level scripting language, making it less flexible than bpftrace in terms of creating custom tracing programs.

It is a Linux auditing system that logs system-level events for security monitoring and analysis. While it is effective for logging system calls and file access events, it lacks the flexibility and low overhead offered by bpftrace, making it less optimal for containerized web server security research. However, we will use this tool's evaluation feature to evaluate the syscalls count of the loaded profile.

### 2.8.3 Ebpf and bpftrace

eBPF, an extended Berkeley Packet Filter, is a robust and modern Linux kernel tracing framework that allows the creation of custom programs to trace kernel and user-space events. It supports attaching to various types of events, including system calls, function entry/exit, kernel tracepoints, and network events. With a high-level scripting language, tracing programs can be created with ease. eBPF is known for its low overhead, making it suitable for production environments (Aich, 2021). Its dynamic tracing capability allows users to attach to running processes without recompiling or restarting them.

eBPF architecture is based on hooks, which are event-driven eBPF applications. Hooks are initiated when the kernel or an application reaches a particular point. Pre-defined kernel hooks are available for system calls, function entry/exit, kernel tracepoints, network events,

and more. For kernel/user application functions without pre-defined hooks, eBPF's 'kprobe' or 'uprobe' capabilities can be used.

Linux byte-code eBPF applications are created by compiling BPF-C apps with LLVM (Gregg, 2015). After finding the hooking event, eBPF is loaded into the kernel using bpf(2). The eBPF verifier and JIT compiler follow (Aich, 2021). Since programs run directly in the kernel, the code must not harm memory or cause it to crash. The eBPF verifier ensures that eBPF meets safety criteria, such as loop-prevention.

It is a high-level tracing language and tool for Linux that allows users to analyze and monitor system events in real-time. It leverages eBPF (Extended Berkeley Packet Filter) to enable efficient and powerful tracing without significant overhead. BPFtrace can be used to enhance security by providing insights into system calls and events, which can be analyzed to create security profiles.

It can be used to analyze the behavior of a containerized web server environment for enhancing the security with AppArmor profile generation and automation. Here's how bpftrace can be utilized and why it is better than auditd:

- Flexibility and versatility: bpftrace allows users to write custom scripts to trace and analyze a wide range of system events and metrics. This flexibility makes it possible to create more targeted and specific monitoring, which can help in generating more accurate AppArmor profiles for containerized environments.

- Lower overhead: eBPF-based tools like bpftrace generally impose a lower performance overhead compared to auditd. This is due to the efficient and optimized way eBPF processes and filters events within the kernel. The reduced overhead makes bpftrace a better choice for monitoring containerized environments without significantly impacting performance.

- Real-time analysis: bpftrace enables real-time analysis and monitoring of system events, providing up-to-date information on the behavior of the containerized environment. This real-time analysis can help in quickly detecting security issues and refining the AppArmor profiles accordingly.

- Easy integration with AppArmor: bpftrace can be used to collect data on the necessary permissions, access controls, and resource usage for a containerized web server environment. This information can then be utilized to generate accurate and optimized AppArmor profiles, automating the process of securing the environment.

In summary, bpftrace offers better performance, flexibility, and real-time analysis capabilities compared to auditd. These advantages make it an ideal choice for enhancing the security of containerized web server environments through AppArmor profile generation and automation.

eBPF and bpftrace provide a powerful and flexible tracing framework for investigating containerized web server environments. The low overhead, real-time analysis, and easy integration with AppArmor make them ideal for enhancing security through automated profile generation. By using bpftrace to collect data on permissions, access controls, and resource usage, researchers can generate accurate and optimized AppArmor profiles, leading to improved security for containerized web server environments.

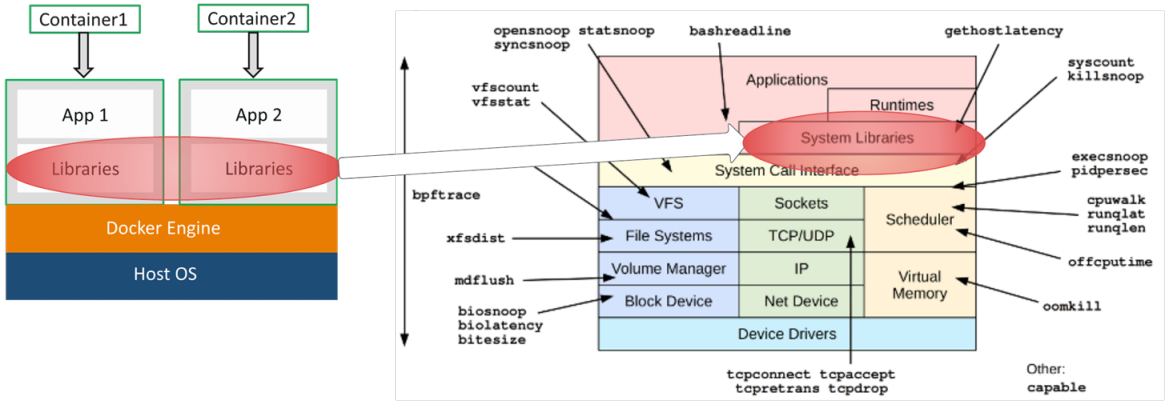


**Figure 2.18.** Libraries that can be traced by bpftrace

## 2.9 Docker and Container Web Server Vulnerabilities

Tunde-Onadele et al., 2019 provides an overview of container exploitation. They collected vulnerabilities for the CVE database and they ran victim containers on a virtual machine using Docker v17.05.0 to avoid interference from other activities on the host. The virtual

computer has 2 GB of RAM and a 40 GB drive, and it runs 64-bit Ubuntu v16.04. Each victim container is running a vulnerable application that is linked to a particular CVE. Clair v2.0.0 does static vulnerability scanning. Sysdig v0.19.1 is used to capture the syscall trace. They ăgather system calls issued by the victim containers in a short period of time to assess the success of each detection strategy (e.g., real-time) and to restore the container realistic use situations (i.e., short-livedness). For each vulnerability, they first launch the victim container and then run the vulnerable application. The workloads are then sent from the VM, and the Sysdig tracing module is launched. Sysdig captures system call traces for around six minutes, including system calls generated by the program under normal workload and during the attack (i.e., from the time the attack is launched until the time the attack succeeds). The time vectors and frequency vectors are then extracted from the raw system call traces in 100 millisecond increments. They use several dynamic detection methods to those feature vectors. They come to the conlcusion that SOM based unsupervised ML method is better at detecting vulnerabilites.

Tomar et al., 2020 provides a throrough taxonomy and a threat model on how docker containers may be attacked and also classify different types of threats.

Most of the other relevant literature deal with kernel security and priviledge escalation attacks, or attacks based on Docker Hub image distribution infrastructure with exploits from CVE or exploit-db. For the purposes of this study, we focus on Denial of Service Availibility attacks, which we have seen previously are successful against previously discussed LSMs (Jain et al., 2022).

In order to conduct a comprehensive study of container vulnerabilities and their potential impact on security, our research will involve creating an exploit testing list based on a targeted selection of CVEs. We derived this list by reviewing relevant literature and identifying key vulnerability types, such as Bypass, Gain Privilege, Denial of Service (DoS), Gain Information, and Execute Code.

Kim et al., 2021 discusses the importance of analyzing shared libraries in container images, particularly Glibc, musl libc, and OpenSSL, which are commonly used and may pose security risks if not properly configured. The paper's findings suggest that profiling container images for shared libraries can help identify potential security issues and inform more

effective security policies. Keeping in line, in this study Docker images of popular web servers were scanned, including Nginx, NextCloud, Apache, and Tomcat, and web scraped using selenium and bs4 the CVE and description of vulnerabilities associated with each. After mapping these vulnerabilities to the vulnerability types identified in the literature, we generated a curated list of CVE exploits for each web server for testing.

The proposition of this exploit testing list and a detailed discussion of the testing process will be presented in Chapter 3 and Chapter 4, respectively. By using this targeted list of CVE exploits, our research aims to provide valuable insights into the effectiveness of various security measures and potential areas for improvement within the Docker ecosystem.

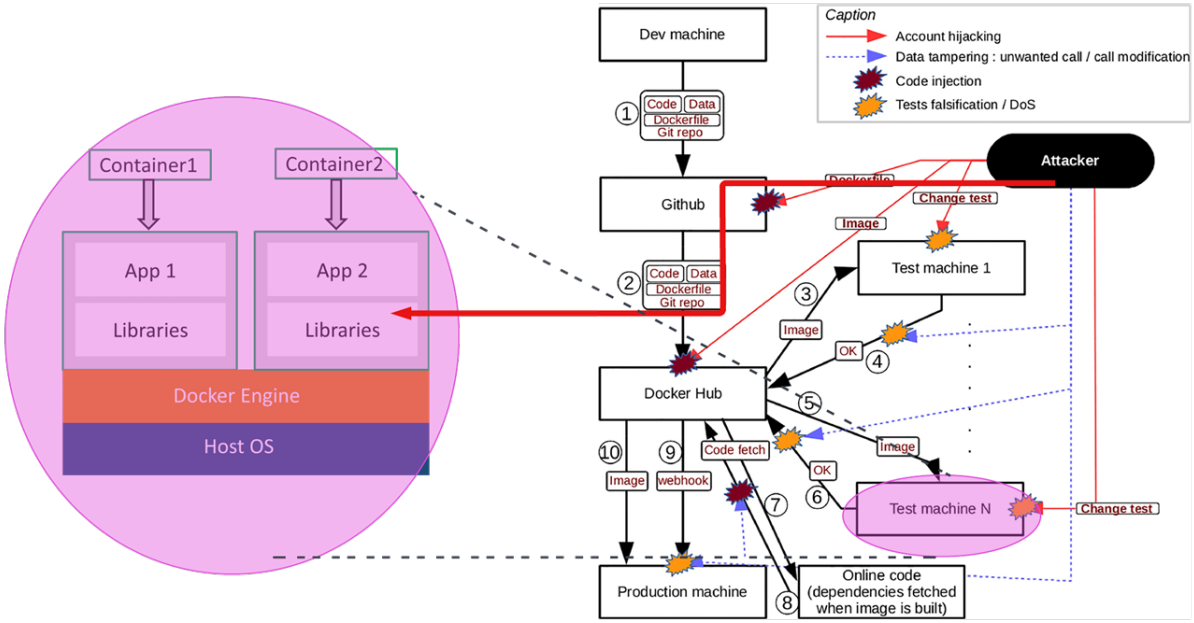


**Figure 2.19.** The docker-ecosystem, and the container to container attacks

## 2.10 Summary

In this chapter, a comprehensive review of relevant literature related to virtualization, containerization, and Docker is conducted. The chapter begins by discussing the fundamental concepts of virtualization and containerization, and it introduces Docker as a leading containerization platform. A detailed overview of Docker's components, including the client,

server, images, registry, and containers, is provided. The chapter also delves into enhancing Docker security through the use of AppArmor and BPFtrace.

The literature review covers the existing security measures and techniques for safeguarding containerized applications, with a focus on capabilities and restricting privileges for Docker containers. Access control mechanisms such as SELinux and AppArmor are explored, along with their implementation in Docker environments. The chapter also evaluates existing approaches to Docker web server security, highlighting the limitations of current solutions and identifying areas for further research.

Tracing tools, such as SystemTap, auditd, eBPF, and bpftrace, are discussed in the context of containerized web server security. The chapter examines vulnerabilities in the Docker ecosystem and web server containers, based on studies by Martin et al. (2018), Tunde-Onadele et al. (2019), and Tomar et al. (2020). The importance of analyzing shared libraries in container images, as emphasized by Kim, Kim, and Lee (2021), is also discussed.

The chapter concludes by outlining the development of an exploit testing list derived from a targeted selection of CVEs, which will be used to assess the effectiveness of various security measures and identify potential areas for improvement in Docker security. This testing process will be further explored in Chapters 3 and 4.

# 3. FRAMEWORK AND METHODOLOGY

In this chapter, the focus will be on developing a methodology to secure Docker web server containers using AppArmor profiles and automating the process with bpftrace. The methodology will address CVE vulnerabilities obtained from a dataset made of exploits denoted in exploitDB along with manual scanning of the said images using docker scan and obtaining the CVEs found after scanning multiple container images in alignment with the classification provided by the existing literature. Nginx and Apache were finally selected as they are the most popular W3Net. The effectiveness of this approach will be compared to docker-sec and existing security AppArmor profiles for web servers deployed through containers with the help of seeing how they perform in denying syscalls with the help of ltrace as has been used to evaluate docker-sec.

The methodology aims to generate AppArmor profiles based on bpftrace logs (capable.bt, upoint attached to glibc) to enhance security for web server containers

Shared libraries in container images can pose security risks if not properly configured

Steps:

1. Attacking the chosen web servers with curated attacks guided by existing literature.
2. Collecting bpftrace logs during the attack and identifying vulnerable patterns of shared lib calls, syscalls.
3. Using the logs to generate AppArmor profiles.

## 3.1 Hypothesis

H01: The use of the proposed profile generation method will not result in a significant difference in the restriction and prevention capabilities of AppArmor profiles compared to those generated without using bpftrace logs.

H$\alpha$1: The use of the proposed profile generation method will result in a significant difference in the restriction and prevention capabilities of AppArmor profiles compared to those generated without using bpftrace logs.

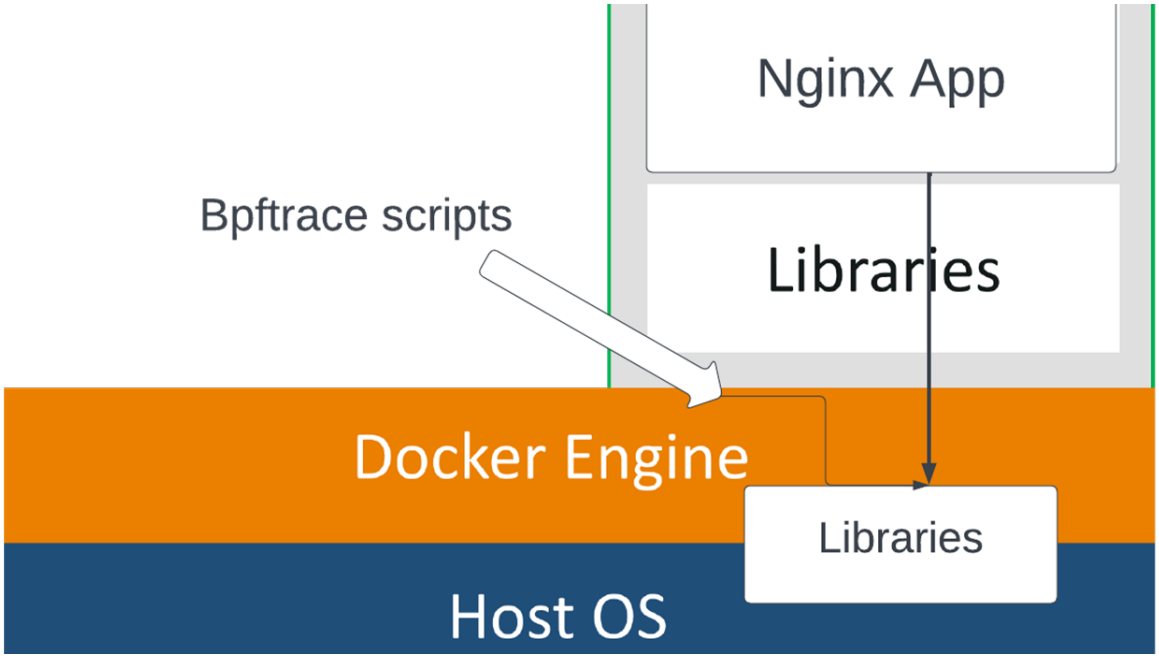


**Figure 3.1.** bpftrace scripts attaching to the pid of the webserver container

## 3.2 Overview of the Proposed Methodology

The proposed methodology aims to generate AppArmor profiles for containerized web servers by:

1. Selecting Docker-based web server containers (such as nginx and apache).
2. Attacking the chosen web servers with curated attacks guided by existing literature.
3. Collecting bpftrace logs during the attack.
4. Using the logs to generate AppArmor profiles.

## 3.3 Data Collection

The data collection process involves selecting web server containers, scanning for vulnerabilities, and compiling a modern CVE dataset.

From this approach, we find these are the types we will classify our exploits into:

1. Bypass Protection Mechanism:

2. Gain Privileges or Assume Identity:

3. DoS: CPU Resource Consumption:

4. Read Application Data:

5. Execute Unauthorized Code or Commands:

| CVE ID | Description | Affected Software |
|---|---|---|
| CVE-2013-2028 | Integer signedness error and stack-based buffer overflow in Nginx due to chunked Transfer-Encoding request | Nginx 1.3.9 - 1.4.0 |
| CVE-2019-20372 | HTTP request smuggling with certain error_page configurations in Nginx | Nginx |
| CVE-2020-1927 | Open redirect vulnerability in mod_rewrite module in Apache | Apache HTTP Server |
| CVE-2017-12613 | Out-of-bounds array dereference in apr_time_exp*() functions in Apache | Apache HTTP Server |
| CVE-2017-7529 | Integer overflow vulnerability in Nginx range filter module, resulting in leak of sensitive information | Nginx 0.5.6 - 1.13.2 |
| CVE-2019-0211 | Vulnerability in Apache HTTP Server with MPM event, worker, or prefork | Apache HTTP Server 2.4 |
| CVE-2014-0133 | Heap-based buffer overflow in the SPDY implementation in Nginx | Nginx 1.3.15 - 1.5.12 |

**Figure 3.2.** CVE List

### 3.4 BPFtrace Scripts and Monitoring

BPFtrace scripts are written and executed to monitor system calls and events related to the exploits being tested. The output of these scripts is saved to log files for further analysis, providing valuable information for generating AppArmor profiles to enhance security.

BPFtrace scripts are written and executed to monitor system calls and events related to the exploits being tested. The output of these scripts is saved to log files for further anal-

ysis. These logs provide valuable information for generating AppArmor profiles to enhance security.

Examples for creating bpftrace scripts that could be useful for detecting activities associated with these exploits.

Monitor file operations related to log files:

```
bpftrace -e 'tracepoint:syscalls:sys_enter_open
{ @[str(args->filename)] = count(); }'
```

This script will track the count of open system calls for each file. Looks for log files being accessed or created.

similarly, using bpftrace we can:

- Monitor excessive resource consumption
- Monitor shared library calls (memcpy, memset, etc)
- Monitor command execution and network activity

Other scripts that will be used in the monitoring process:

1. capable.bt: This script traces security capability checks (cap_capable()) and prints the details of these checks, including timestamp, UID, PID, capability name, and audit status.
2. dcsnoop.bt: This script traces directory entry cache (dcache) lookups and prints the details of these lookups, including timestamp, PID, command, type of lookup (hit or miss), and file name.
3. execsnoop.bt: This script traces new processes created via exec() syscalls and prints the details of these processes, including timestamp, PID, and the command arguments.
4. opensnoop.bt: This script traces open() syscalls and prints the details of these syscalls, including PID, command, file descriptor, error code, and the file path.

5. setuids.bt: This script traces setuid family syscalls (privilege escalation) and prints the details of these syscalls, including timestamp, PID, command, UID, syscall name, and arguments.

6. syscount.bt: This script counts system calls and prints the top 10 syscall IDs and the top 10 processes making syscalls.

7. tcpaccept.bt: This script traces TCP accept()s and prints the details of these connections, including timestamp, PID, command, remote address and port, local address and port, and backlog information.

8. tcpconnect.bt: This script traces TCP connect()s and prints the details of these connections, including timestamp, PID, command, remote address and port, local address and port, and the socket family (IPv4 or IPv6).

The .bt scripts mentioned above are available in the open source literature.

For example, to trace syscalls made by a specific container, we can use the following bpftrace command:

```
bpftrace -e 'tracepoint:syscalls:sys_enter_* { @[probe] = count(); }'
```

Similarly to measure shared lib calls, we can attach uprobe to the tracepoint. We can leverage that to see how many shared lib calls are made which is an indicator for a DOS attack.

```
bpftrace -e 'tracepoint:uprobe:/path/to/shared/library:
function_name { @[probe] = count(); }'
```

The path to the shared library for the memset and memcpy shared library calls for an nginx container inside an Ubuntu 22.04 LTS VM is /lib/x86_64-linux-gnu/libm.so.6.

The memset and memcpy functions are defined in the C standard library, which is typically implemented as a shared library. The shared library for the C standard library is typically named libm.so.6 on Linux systems.

In an Ubuntu 22.04 LTS VM, the shared library for the C standard library is located in the directory /lib/x86_64-linux-gnu. Therefore, the path to the shared library for the memset and memcpy shared library calls is /lib/x86_64-linux-gnu/libm.so.6.

### 3.5 Profile Generation Algorithm

1. Process the bpftrace logs to identify patterns, behaviors, or syscalls that need to be addressed in the AppArmor profile.
2. Generate the AppArmor profile using the processed data, incorporating appropriate rules and restrictions such as file system accesses, shared library call limiting, syscall restriction.

- Set up the Nginx/Apache container with AppArmor in complain mode
- Run various bpftrace tests on the container to generate logs
- Use the logs to create the following profiles:
- Profile 1: Based on Docker documentation's profile and AppArmor complain logs
- Profile 2: Based on Docker documentation's profile, bpftrace logs, and container runtime analysis
- Evaluate and compare the generated profiles

For example, the generated AppArmor profile for an nginx container may look like:

```
#include <tunables/global>

profile nginx flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  #include <abstractions/nginx>

  # Custom rules based on bpftrace logs
```

```
  deny /path/to/specific/file r,
  deny /path/to/specific/directory/ r,
  capability chown,
  ...
}
```

### 3.5.1 Bash Script for Automating the Process

Create a bash script that:

1. Executes the bpftrace commands to gather relevant system data during the attack.
2. Processes the data collected by bpftrace, extracting the necessary information for generating the AppArmor profile.
3. Generates the AppArmor profile using the processed data.
4. Loads and enforces the newly created AppArmor profile on the target container.

For example:

```
#!/bin/bash

# Run bpftrace scripts in the background and redirect output to
temporary files
bpftrace -e 'tracepoint:syscalls:sys_enter_open
{ printf("%s\n", str(args->filename)); }'
> bt_open.out & bpftrace -e 'tracepoint:syscalls:sys_enter_execve
{ printf("%s\n", comm); }' > bt_exec.out &
bpftrace -e 'u:libc:malloc
{ printf("%s\n", comm); } u:libc:free
{ printf("%s\n", comm); }' > bt_heap.out &
```

```
# Monitor bpftrace output for some time (e.g., 60 seconds)
sleep 60

# Kill bpftrace processes
pkill -f bpftrace

# Generate an AppArmor profile based on the collected data
profile_name="my_docker_container"
output_file="${profile_name}.apparmor"

cat > "${output_file}" <<EOL
#include <tunables/global>

profile ${profile_name} flags=(attach_disconnected,mediate_deleted
    ) {
  #include <abstractions/base>

  # Read files
EOL

# Read bt_open.out for accessed files
while read -r file; do
  echo "  \"${file}\" r," >> "${output_file}"
done < bt_open.out

cat >> "${output_file}" <<EOL

  # Capabilities
  capability,
```

```
  deny capability sys_ptrace,

  # Network
  network inet stream,
  network inet6 stream,
  network inet dgram,
  network inet6 dgram,

  # Miscellaneous
  deny mount,
  deny ptrace,
}
EOL

# Remove temporary files
rm bt_open.out bt_exec.out bt_heap.out

echo "Generated AppArmor profile: ${output_file}"
```

### 3.6 Evaluation

Evaluate the generated AppArmor profiles by comparing their effectiveness using ltrace to measure the performance of the two AppArmor profiles (the one generated using bpftrace logs and the existing security AppArmor profile without bpftrace logs) by seeing which of the two profiles are more restrictive when under attack. The evaluation focuses on the security improvements.

1. Deploy the web server containers with the generated AppArmor profile and the existing security AppArmor profile.

2. Configure ltrace to attach to the pid of the container being attacked and log its output.

3. Analyze the ltrace logs for each profile, comparing the security-related events recorded.

4. Assess the effectiveness of the generated AppArmor profile in mitigating security risks compared to the existing security AppArmor profile.

By following these steps, the evaluation process ensures that the study addresses the security improvements provided by the generated AppArmor profile compared to the existing security AppArmor profiles for web servers deployed through containers. This approach enables a comprehensive understanding of the impact of the proposed methodology on the security of containerized web servers like Nginx and Apache.

## 3.7 Summary

This chapter presents a comprehensive methodology for securing Docker web server containers, specifically Nginx and Apache, using AppArmor profiles and bpftrace. The methodology consists of data collection and analysis, attack and reconnaissance, profile generation algorithm, and evaluation. The results obtained from this methodology will be presented in Chapter 4: Results, where the effectiveness of the generated AppArmor profiles in securing Docker web server containers will be discussed. Finally, Chapter 5: Discussion and Conclusion will provide a comprehensive analysis of the findings and outline future research directions.

# 4. RESULTS

In this chapter, the results obtained from the evaluation of the proposed methodology for securing Docker web server containers using AppArmor profiles and bpftrace will be presented. This includes a comparison of the generated AppArmor profiles with existing security frameworks, such as docker-sec. The focus will be on assessing the effectiveness of the generated profiles in mitigating security risks in containerized web servers like Nginx and Apache httpd.

## 4.1 Overview of the Results

The results indicate that bpftrace is able to identify shared library calls such as memcpy, memset, as well as syscalls such as multiple instances of recv(), 257 instances of stat, 186 instances of close(), and 202 instances of mprotect(). Additionally, a non-exhaustive list of attacker IPs such as 185.199.110.133 (Zap news fetcher), 104.21.1.121 (Zap-cfu), and 104.21.1.121 (ZAP-telemetry) can be observed and are blocked in the subsequent generated AppArmor profile. Multiple accesses to shared libraries and file accesses are noticed, which are used to generate the AppArmor profile. The profile is loaded with the apparmor_parser module and set to complain mode. We also realize that the nature of attacks dictate that multiple calls for shared libraries can be noticed in the bpftrace scripts of dcache lookups - dcsnoop.bt. These indicate memcpy and memset calls, which have been subsequently addressed in our generated apparmor profile by using the proper r_limits. We can limit the overall memory usage of a process using RLIMIT_AS (address space limit) or RLIMIT_DATA (data segment size limit) to restrict the total memory a process can allocate, which would indirectly affect the use of memcpy(). It's important to note that setting these limits too low may cause unintended consequences or crashes, as they affect the entire process and not just memcpy() usage.

1. Launch Metasploit

2. Search for modules: Search command to find relevant exploit, auxiliary, or scanner modules related to the CVEs we want to test. For example, we can run search cve:2017-7529 to find modules related to CVE-2017-7529.

3. Select a module

4. Configure the module: like set RHOSTS 172.17.0.2.

5. Run the module

6. Repeat the steps for 14 exploits (6 CVEs for nginx, 8 for Apache)

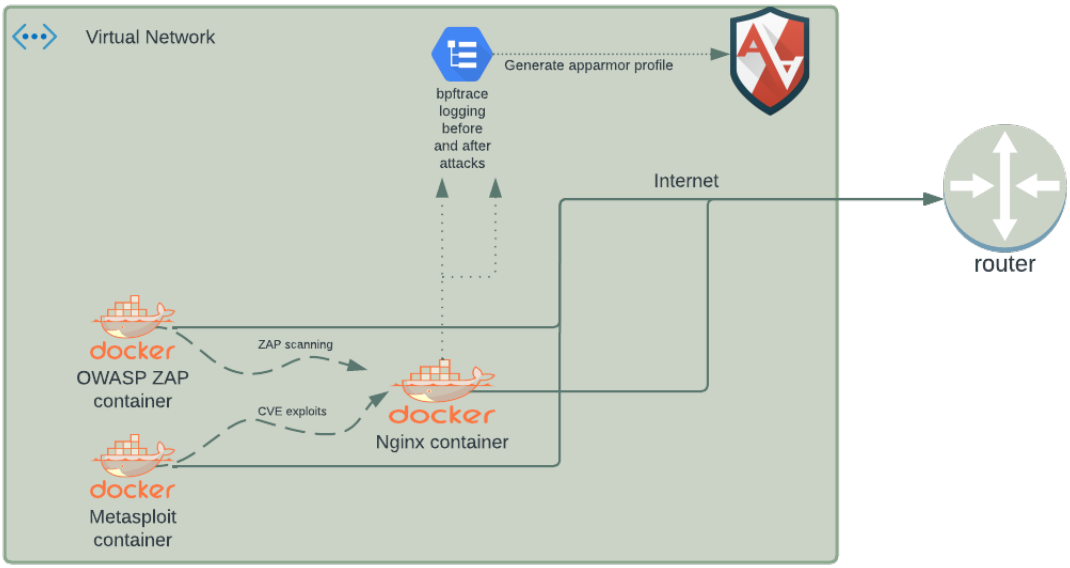


**Figure 4.1.** Experimental Setup

We can then evaluate the two profiles by first attaching ltrace to the containers loaded with docker-sec's generated profile for nginx, and then attacking it with metasploit with a module from one of the cve's taken from the dataset. This would generate calls that would show up on the lrace output. The ltrace outputs show visible difference in calls, especially to the shared libraries, and the syscalls mentioned. With this we can conclude that the profile is more restrictive and hence more secure.

This is how the dataset of cve's used in the metasploit module was collected:

- A business account on Snyk.io was established to gain access to the docker scan feature on Snyk on an Ubuntu 22.04 Virtual Machine.

- The docker scan command was run against the top 22 docker images per downloads as per January 2023, from DockerHub repository. For our research, we specifically focused on Apache and Nginx web server images.
- This process resulted in a list of vulnerabilities that needed to be addressed per image as per the Snyk scanner in the docker files of each image. These vulnerabilities were stored in 22 text files.
- A custom python script (scrape.py) was used to ping each website and store the details on each CSV file per image.
- The dataset was uploaded in CSV format on GitHub as an open-source resource (can be found at: https://github.com/RoughPatch1/DED).
- Then this dataset was further processed using exploit-db's extensive database of CVE's and metasploitable CVE's were selected for the final testing which fell under the classification system that was devised for this study.

This study classified the CVE's on the basis of:

1. Existing literature, specifically the Lic-Sec paper, to identify the classification and type of CVEs used in previous research.
2. Following the framework established in the literature, we classified the collected CVEs into categories based on their attack type and severity.
3. We selected CVEs relevant to our research focus, ensuring that the chosen CVEs represent a variety of attack types and securities, allowing for a comprehensive evaluation of the security provided by our generated AppArmor profiles.

We collected data from CVE vulnerabilities related to Docker-based web servers, specifically focusing on Nginx and Apache servers. To obtain this dataset, we performed a Docker scanning for both Nginx and Apache, which revealed several vulnerabilities in the affected packages. The collected data was mapped to classifications found in existing literature and

exploit databases. Some of the key vulnerability classifications included bypass protection mechanisms, gain privileges or assume identity, DoS (denial of service) with CPU resource consumption, read application data, and execute unauthorized code or commands. These included uncontrolled recursion, improper locking, out-of-bounds read, integer overflow or wraparound, allocation of resources without limits or throttling, improper input validation, missing release of resource after effective lifetime, and open redirect.

Certain trend and patterns were observed in the data. For example, several vulnerabilities were related to improper input validation, which is a common issue in software development. Additionally, we found that out-of-bounds read vulnerabilities and allocation of resources without limits or throttling were frequent occurrences in the dataset.

For example, both Nginx and Apache had vulnerabilities related to bypass protection mechanisms, such as buffer overflow (e.g., CVE-2013-2028 for Nginx) and buffer over-read (e.g., CVE-2017-7679 for Apache). These vulnerabilities when executed repeatedly tried to exploit the 'chunked encoding' module by sending fraudulent HTTP GET calls that would cause the container to bypass permissions.

```
These can be limited by setting rlimit for the input data byte.
```

Another common vulnerability type was related to privilege escalation, which allows attackers to gain elevated privileges or assume the identity of another user (e.g., CVE-2017-7529 for Nginx and CVE-2019-0211 for Apache). Additionally, we found that both web servers had vulnerabilities associated with DoS attacks targeting CPU resource consumption, such as HTTP/2 implementation vulnerabilities (e.g., CVE-2019-9511 for Nginx and CVE-2018-17199 for Apache). Furthermore, vulnerabilities that enabled unauthorized access to application data or the execution of arbitrary code were also identified in both Nginx and Apache web servers (e.g., CVE-2014-0133 and CVE-2019-20372 for Nginx; CVE-2020-1927 and CVE-2017-12613 for Apache).

```
These can be limited by setting the rlimit for memlock for
memory resource limiting.
```

In summary, the data collection and analysis revealed a variety of vulnerability types affecting Docker-based Nginx and Apache web servers. The observed trends and patterns indicate that r limits must be set in apparmor profiles for improvement in mandatory access control for containerized web servers.

```
testavi@testavi-VirtualBox:/etc/apparmor.d$ sudo docker logs nginx
/docker-entrypoint.sh: 13: cannot create /dev/null: Permission denied
/docker-entrypoint.sh: No files found in /docker-entrypoint.d/, skipping config
uration
2023/04/05 15:45:35 [notice] 1#1: using the "epoll" event method
2023/04/05 15:45:35 [notice] 1#1: nginx/1.23.4
2023/04/05 15:45:35 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6
)
2023/04/05 15:45:35 [notice] 1#1: OS: Linux 5.19.0-35-generic
2023/04/05 15:45:35 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/04/05 15:45:35 [notice] 1#1: start worker processes
2023/04/05 15:45:35 [notice] 1#1: start worker process 9
172.17.0.1 - - [05/Apr/2023:16:02:26 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozil
la/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/111.0" "-"
2023/04/05 16:02:26 [error] 9#9: *1 open() "/usr/share/nginx/html/favicon.ico"
failed (2: No such file or directory), client: 172.17.0.1, server: localhost, r
equest: "GET /favicon.ico HTTP/1.1", host: "localhost", referrer: "http://local
host/"
172.17.0.1 - - [05/Apr/2023:16:02:26 +0000] "GET /favicon.ico HTTP/1.1" 404 153
 "http://localhost/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/2
0100101 Firefox/111.0" "-"
testavi@testavi-VirtualBox:/etc/apparmor.d$
```

**Figure 4.2.** GET calls in nginx logs with rlimit set

We can see the get call made by the localhost through firewall which opened the nginx index page, but no call was registered on the ltrace output when secured with our profile.

## 4.2 Experimental Setup

In this study, we aimed to test the security of a Docker container running Nginx 1.15.0 and Apache 9.0.16 web servers on a virtual machine with Ubuntu 22.04 LTS, with 4096 MB RAM and 110 GB virtual storage. We followed a multi-step process to test the security of the container, including:

- Setting up the virtual machine and installing Docker and its dependencies.
- Creating a Dockerfile for building the Nginx and Apache containers and configuring their respective web servers.

| | Name | Link | Affected Package | Description |
|---|---|---|---|---|
| 0 | Uncontrolled Recursion in pcre3 \| CVE-2017-11164 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2017-11164 | Ubuntu:22.04 | In PCRE 8.41, the OP_KETRMAX feature in the |
| 1 | Improper Locking in openssl \| CVE-2022-3996 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-3996 | Ubuntu:22.04 | If an X.509 certificate contains a malformed p |
| 2 | Out-of-bounds Read in ncurses \| CVE-2022-29458 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-29458 | Ubuntu:22.04 | ncurses 6.3 before patch 20220416 has an ou |
| 3 | CVE-2022-3857 in libpng1.6 \| CVE-2022-3857 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-3857 | Ubuntu:22.04 | © 2023 Snyk Limited |
| 4 | Integer Overflow or Wraparound in krb5 \| CVE-2018-5709 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2018-5709 | Ubuntu:22.04 | An issue was discovered in MIT Kerberos 5 (a |
| 5 | CVE-2022-3219 in gnupg2 \| CVE-2022-3219 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-3219 | Ubuntu:22.04 | © 2023 Snyk Limited |
| 6 | Allocation of Resources Without Limits or Throttling in glibc \| CVE-2016-20013 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2016-20013 | Ubuntu:22.04 | sha256crypt and sha512crypt through 0.6 all |
| 7 | Improper Input Validation in coreutils \| CVE-2016-2781 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2016-2781 | Ubuntu:22.04 | chroot in GNU coreutils, when used with --us |
| 8 | Uncontrolled Recursion in binutils \| CVE-2021-3530 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2021-3530 | Ubuntu:22.04 | A flaw was discovered in GNU libiberty within |
| 9 | Improper Input Validation in binutils \| CVE-2019-1010204 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2019-1010204 | Ubuntu:22.04 | GNU binutils gold gold v1.11-v1.16 (GNU bin |
| 10 | Missing Release of Resource after Effective Lifetime in binutils \| CVE-2018-20657 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2018-20657 | Ubuntu:22.04 | The demangle_template function in cplus-de |
| 11 | Allocation of Resources Without Limits or Throttling in binutils \| CVE-2017-13716 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2017-13716 | Ubuntu:22.04 | The C++ symbol demangler routine in cplus- |
| 12 | Out-of-bounds Write in bash \| CVE-2022-3715 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-3715 | Ubuntu:22.04 | A flaw was found in the bash package, where |
| 13 | Open Redirect in wget \| CVE-2021-31879 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2021-31879 | Ubuntu:22.04 | GNU Wget through 1.21.1 does not omit the |
| 14 | Off-by-one Error in systemd \| CVE-2022-3821 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-3821 | Ubuntu:22.04 | An off-by-one Error issue was discovered in S |
| 15 | CVE-2022-43552 in curl \| CVE-2022-43552 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-43552 | Ubuntu:22.04 | © 2023 Snyk Limited |
| 16 | Cleartext Transmission of Sensitive Information in curl \| CVE-2022-43551 \| Snyk | https://www.cve.org/CVERecord?id=CVE-2022-43551 | Ubuntu:22.04 | A vulnerability exists in curl <7.87.0 HSTS che |

**Figure 4.3.** Snippet of httpd vulnerabilities

- Testing the containers one by one using bpftrace scripts to measure system calls and shared library calls made by the containers. We also conducted Metasploit testing and OWASP ZAP scans to simulate potential attacks on the containers.

- Analyzing the bpftrace logs and singling out harmful syscalls and shared library calls, such as recv(), memcpy, and memset, and applying rlimits and capability restrictions to prevent potential attacks.

- Comparing the effectiveness of our security measures with an AppArmor profile generated by Docker-sec, using ltrace to attach to both profiles and counting the number of harmful syscalls made.

- To carry out our experiments, we used several tools, including bpftrace, Metasploit, OWASP ZAP, and ltrace. We chose bpftrace for measuring system calls and shared library calls as it provides a powerful and flexible way to trace kernel and user-space events. Metasploit was used for conducting penetration testing on the containers, while OWASP ZAP was used for simulating reconnaissance phase of an attack. Lastly, we

used ltrace to attach to our AppArmor profile and Docker-sec's profile to compare their effectiveness.

Overall, our experimental setup aimed to thoroughly test the security of the Docker container running Nginx and Apache web servers, using a combination of different tools and techniques. By applying rlimits and capability restrictions and comparing our AppArmor profile with Docker-sec's profile, we were able to evaluate the effectiveness of our security measures and identify areas for improvement.

To attempt the exploitation of CVE-2013-2028 vulnerability in a modern Nginx container, challenges with the stack canary trace were encountered and could not be resolved. Since the vulnerability is specific to Nginx versions 1.3.9 to 1.4.0 and no corresponding image is maintained in the official repository, an Nginx 1.15.0 image was built using a Dockerfile:

```
FROM nginx:1.15.0

COPY nginx.conf /etc/nginx/nginx.conf
```

An accompanying nginx.conf file was created:

```
    user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
```

```
    access_log  /var/log/nginx/access.log  main;

    include /etc/nginx/conf.d/*.conf;
}
```

Subsequently, multiple bpftrace scripts, including capable.bt, dcsnoop.bt, execsnoop.bt, and others, were executed one after another to establish a control baseline.

Before running the bpftrace commands, the bpftrace tool must be installed on the system. The installation instructions for specific operating systems can be found in the bpftrace GitHub repository: https://github.com/iovisor/bpftrace .

```
sudo bpftrace -p <PID of the nginx/httpd container>
capable.bt
```

The above command is for the 'capable.bt' script which logs capability calls to the system by that specific process.

To obtain the PID of the process:

```
docker inspect --format '{{.State.Pid}}' nginx_container
```

The IP address of the web server container was found through

```
docker inspect --format '{{.NetworkSettings.IPAddress}}'
nginx_container
```

This was then used in metapsploit modules along with creating OWASP ZAP sessions for the attacks.

This setup provided the foundation for the analysis of bpftrace logs under control and attack scenarios, ultimately enabling the identification of potential security issues and vulnerabilities.

### 4.2.1 Control Logs

Control logs provide valuable information on the normal operation of a web server, helping to establish a baseline for comparison with logs from attack scenarios. Here, we analyze the

control logs for various bpftrace scripts, including tcpconnect.bt, dcsnoop.bt, capable.bt, and execsnoop.bt.

### tcpconnect.bt Control Log

The tcpconnect.bt control log records the TCP connections initiated by processes within the Nginx container during normal operation. This log helps identify legitimate network connections and the processes responsible for them.

In the control log, we observe connections from the Nginx process (PID 230) to other services within the container, using source port 80 and destination ports 53456 and 53460. No unexpected or suspicious connections are identified in the control environment.

### dcsnoop.bt Control Log

The dcsnoop.bt control log monitors Docker container events, such as start, stop, and pause, in the control environment. It provides insights into the normal operation of the Nginx container and any other containers running simultaneously.

In the control log, we see expected events such as the Nginx container's start event (PID 230) and stop event (PID 2847). There are no unexpected or malicious container events observed in the control environment.

### capable.bt Control Log

The capable.bt control log captures Linux capability checks made by processes within the Nginx container during normal operation. This log helps identify which capabilities are required for the proper functioning of the container.

In the control log, we see expected capabilities such as CAP_DAC_READ_SEARCH by systemd-resolve (PID 101), CAP_NET_BIND_SERVICE by nginx (PID 230), and CAP_NET_RAW by systemd-udevd (PID 1000). No unusual or excessive capabilities are observed in the control environment.

**execsnoop.bt Control Log**

The execsnoop.bt control log records the execution of new processes within the Nginx container during normal operation. This log helps identify the processes involved in the regular functioning of the web server and can aid in detecting any unexpected or malicious process executions.

In the control log, we observe the expected processes such as nginx (PID 230), systemd-resolve (PID 101), systemd-udevd (PID 1000), and sudo (PID 2847). No unexpected or malicious process executions are identified in the control environment.

By analyzing the control logs for various bpftrace scripts, we establish a baseline of the normal operation of the Nginx web server. This baseline helps in detecting any deviations or anomalies in the logs generated during attack scenarios, providing valuable insights into potential vulnerabilities or security issues.

**opensnoop.bt Control Log**

By analyzing the control logs we see that the opensnoop2.log file is tracing the open system calls being made by the systemd-oomd process, which is responsible for monitoring and controlling the system's out-of-memory killer.

The log shows that the systemd-oomd process is repeatedly opening and reading the /proc/meminfo file and the /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem directory, which contain information about the system's memory usage and the memory usage of processes running on the system. This behavior is expected from the systemd-oomd process as it needs to constantly monitor the system's memory usage and make decisions on which processes to terminate in case of low memory situations.

### 4.2.2 Attack Logs

The attack logs provide insights into the events that occurred during the attempted exploitation of the system. These logs can help identify potential vulnerabilities and anomalies that may have been leveraged by an attacker.

```
avigyan7@avigyan7-VirtualBox:~/bpftrace/build$ sudo bpftrace capable.bt -p 2698
[sudo] password for avigyan7:
Attaching 3 probes...
Tracing cap_capable syscalls... Hit Ctrl-C to end.
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
23:43:28 1000   2211   gmain            2    CAP_DAC_READ_SEARCH  0
23:43:28 1000   2211   gmain            1    CAP_DAC_OVERRIDE     0
23:43:28 1000   2211   gmain            2    CAP_DAC_READ_SEARCH  0
23:43:28 1000   2211   gmain            1    CAP_DAC_OVERRIDE     0
23:43:29 1000   3504   sudo             29   CAP_AUDIT_WRITE      0
23:43:29 1000   3504   sudo             29   CAP_AUDIT_WRITE      0
23:43:29 1000   3504   sudo             7    CAP_SETUID           4
23:43:29 1000   3504   sudo             7    CAP_SETUID           4
23:43:29 1000   3504   sudo             6    CAP_SETGID           4
23:43:29 1000   3504   sudo             7    CAP_SETUID           4
23:43:29 1000   3504   sudo             29   CAP_AUDIT_WRITE      0
23:43:29 0      230    systemd-journal  19   CAP_SYS_PTRACE       0
23:43:29 0      230    systemd-journal  19   CAP_SYS_PTRACE       0
23:43:29 0      230    systemd-journal  19   CAP_SYS_PTRACE       0
23:43:29 0      230    systemd-journal  19   CAP_SYS_PTRACE       0
23:43:29 0      230    systemd-journal  19   CAP_SYS_PTRACE       0
```

**Figure 4.4.** Snippet of capablelog.bt

```
Attaching 4 probes...
Tracing dcache lookups... Hit Ctrl-C to end.
TIME     PID    COMM             T FILE
151      409    systemd-oomd     R proc/meminfo
151      409    systemd-oomd     R meminfo
321      983    gmain            R var/lib/snapd/desktop/desktop-directories
321      983    gmain            R lib/snapd/desktop/desktop-directories
321      983    gmain            R snapd/desktop/desktop-directories
321      983    gmain            R desktop/desktop-directories
322      983    gmain            R desktop-directories
322      983    gmain            R usr/local/share/desktop-directories
322      983    gmain            R local/share/desktop-directories
322      983    gmain            R share/desktop-directories
322      983    gmain            R desktop-directories
322      983    gmain            R usr/share/ubuntu/desktop-directories
322      983    gmain            R share/ubuntu/desktop-directories
322      983    gmain            R ubuntu/desktop-directories
322      983    gmain            R desktop-directories
323      983    gmain            R home/testavi/.local/share/desktop-directories
323      983    gmain            R testavi/.local/share/desktop-directories
323      983    gmain            R .local/share/desktop-directories
```

**Figure 4.5.** Snippet of dcsnoop.bt

### Metasploit Test

During the Metasploit attack, the capable.bt log indicated an increase in the CAP_SYS_ADMIN capability usage, which is a powerful capability often associated with exploits. Additionally, the opensnoop.bt log showed an increase in open() calls, particularly for files related to system processes like polkitd, which may indicate attempts to exploit the system. In the execsnoop logs, new processes, such as polkitd, were observed, suggesting that the exploit at-

```
Attaching 3 probes...
TIME(ms)   PID   ARGS
3143       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3152       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3153       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3156       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3156       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3156       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3156       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
3157       5423  dbus-launch --autolaunch=9483a8a770104ebea4897ad221000556 --binary-syntax --close-stderr
14051      5454  runc --root /var/run/docker/runtime-runc/moby --log
/run/containerd/io.containerd.runtime.v2.task/moby/46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4
e/log.json --log-format json --systemd-cgroup delete
46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4e
14073      5460  /usr/bin/containerd-shim-runc-v2 -namespace moby -address /run/containerd/containerd.sock -
publish-binary /usr/bin/containerd -id 46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4e -bundle
/run/containerd/io.containerd.runtime.v2.task/moby/46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4
e delete
14080      5466  runc --root /var/run/docker/runtime-runc/moby --log
/run/containerd/io.containerd.runtime.v2.task/moby/46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4
e/log.json --log-format json delete --force 46195d4cc051e5bbf6cd6f2227b10584532b301769fa7c1fcd0cc736cadb2e4e
```

**Figure 4.6.** Snippet of execsnoop.bt

```
Attaching 6 probes...
Tracing open syscalls... Hit Ctrl-C to end.
PID    COMM               FD ERR PATH
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /proc/meminfo
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
416    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/mem
```

**Figure 4.7.** Snippet of opensnoop.bt

tempts made during the attack were targeting this process. The results of the CVE exploits are given in the following table.

### OWASP ZAP Test

The OWASP ZAP logs revealed several new processes, including gmain, systemd-oomd, systemd-timesyn, systemd-network, kerneloops, and systemd-logind. These processes were likely introduced due to the OWASP ZAP session. Moreover, the log shows additional UIDs

```
Counting syscalls... Hit Ctrl-C to end.

Top 10 syscalls IDs:
@syscall[3]: 11499
@syscall[257]: 12272
@syscall[47]: 14064
@syscall[8]: 29157
@syscall[7]: 37943
@syscall[202]: 43244
@syscall[24]: 85708
@syscall[0]: 85718
@syscall[186]: 98973
@syscall[1]: 118944

Top 10 processes:
@process[java]: 13300
@process[systemd]: 14522
@process[gdbus]: 15899
@process[Thread-9]: 22644
@process[gnome-shell]: 35490
@process[C2 CompilerThre]: 36386
@process[ZAP-DownloadIns]: 58599
```

**Figure 4.8.** Snippet of syscount.bt

(100, 103, and 108) and a reduced set of capabilities compared to the control and Metasploit logs, with unique ones such as CAP_SYSLOG. The dcsnoop logs during the OWASP ZAP attack showed an increased number of connect and bind syscalls, indicating increased network activity as ZAP scans and probes the target system for vulnerabilities.

```
Attaching 2 probes...
Tracing tcp connections. Hit Ctrl-C to end.
TIME     PID      COMM             SADDR                                   SPORT  DADDR
DPORT
00:19:03 5926     python           127.0.0.1                               36682  127.0.0.1
40372
00:19:03 5926     python           127.0.0.1                               36684  127.0.0.1
40372
00:19:03 5926     python           127.0.0.1                               36692  127.0.0.1
40372
00:19:04 5926     python           127.0.0.1                               36708  127.0.0.1
40372
00:19:04 5926     python           127.0.0.1                               36710  127.0.0.1
40372
00:19:05 5926     python           127.0.0.1                               36720  127.0.0.1
40372
00:19:05 5926     python           127.0.0.1                               36736  127.0.0.1
40372
00:19:05 2536     Socket Thread    10.0.2.15                               33086  34.107.221.82
80
00:19:05 2536     Socket Thread    10.0.2.15                               33096  34.107.221.82
```

**Figure 4.9.** Snippet of tcpconnect.bt

**Table 4.1.** Exploits that were used to generate the system calls

| Exploits | Docker-sec (profile 1) | Bpftrace-gen (profile 2) |
|---|---|---|
| CVE-2013-2028 | Not Restricted | Restricted |
| CVE-2017-7529 | Not Restricted | Restricted |
| CVE-2019-9511 | Not Restricted | Restricted |
| CVE-2019-9513 | Not Restricted | Restricted |
| CVE-2014-0133 | Restricted | Restricted |
| CVE-2019-20372 | Restricted | Restricted |
| CVE-2017-7679 | Not Restricted | Restricted |
| CVE-2019-0211 | Not Restricted | Restricted |
| CVE-2016-2161 | Not Restricted | Restricted |
| CVE-2018-17199 | Not Restricted | Restricted |
| CVE-2014-3523 | Not Restricted | Restricted |
| CVE-2020-1927 | Not Restricted | Restricted |
| CVE-2017-12613 | Not Restricted | Restricted |
| CVE-2021-41773 | Not Restricted | Restricted |

### Comparing Control and Attack Logs

Comparing the control and attack logs highlights differences in the processes, capabilities, and UIDs observed during the events. The Metasploit logs show potential exploit attempts with increased usage of powerful capabilities, new processes, and open() calls targeting sen-

sitive files. The OWASP ZAP logs reflect the impact of the ZAP session on the system, with several new processes and increased network activity.

By analyzing these attack logs, we can identify potential vulnerabilities and security issues that may have been exploited or targeted by the attacker. This information can be used to strengthen the system's security posture by addressing these vulnerabilities and implementing additional security measures.

When comparing the control and attack logs, several key differences were observed that contributed to the generation of the AppArmor profile.

**Processes**

Control logs showed standard processes such as nginx, systemd-resolve, systemd-udevd, and sudo. In contrast, the attack logs displayed additional processes:

Metasploit logs: The introduction of the polkitd process, which is likely associated with the attempted exploit. OWASP ZAP logs: New processes like gmain, systemd-oomd, systemd-timesyn, systemd-network, kerneloops, and systemd-logind were observed, indicating increased activity during the ZAP session.

**Capabilities**

The control and Metasploit logs shared the same set of capabilities. However, the OWASP ZAP logs had a reduced set of capabilities, with some unique ones like CAP_SYSLOG. The Metasploit logs showed an increase in CAP_SYS_ADMIN usage, which is a powerful capability often associated with exploits.

**System Calls**

Comparing the syscalls in the control and attack logs:

The dcsnoop logs during the OWASP ZAP attack showed increased connect and bind syscalls, indicating heightened network activity. It also showed stat, mprotect and close syscalls, indicating repeated attempts. The opensnoop logs from the Metasploit attack revealed an increase in open() calls for sensitive files and system processes like polkitd.

### UIDs

The OWASP ZAP logs introduced multiple new UIDs (100, 103, and 108), which were likely related to the ZAP session.

### AppArmor Profile Generation Criteria

By analyzing the differences between the control and attack logs, we identified potential security concerns and vulnerabilities that were used to generate the AppArmor profiles. For example:

1. Restricting access to specific file paths observed in the opensnoop logs during the Metasploit attack.
2. Denying powerful capabilities like CAP_SYS_ADMIN, which showed increased usage in the Metasploit logs.
3. Limiting network access based on the required protocols and syscalls observed in the dcsnoop logs during the OWASP ZAP attack.
4. These adjustments to the AppArmor profiles aimed to mitigate the risks identified in the attack logs and enhance the overall security of the system.

The algorithm processes bpftrace logs to extract relevant information such as

- File System Acceses (/proc, /usr)
- Dcache lookups
- Syscalls (recv(), close())
- Shared Library calls (memcpy, memset in glibc)

Set appropriate rlimits such as:

- set rlimit data $\leq$100M, set rlimit nproc $\leq$10

## 4.3 Generated AppArmor Profiles

In this section, we discuss the various AppArmor profiles that were generated during the course of our analysis. These profiles aim to provide enhanced security for the Nginx container, based on different sources of information and analysis methods.


```
{"log":"/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
\n","stream":"stdout","time":"2023-03-04T04:54:21.894156775Z"}
{"log":"/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
\n","stream":"stdout","time":"2023-03-04T04:54:21.894194596Z"}
{"log":"/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
\n","stream":"stdout","time":"2023-03-04T04:54:21.89864936Z"}
{"log":"10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
\n","stream":"stdout","time":"2023-03-04T04:54:21.907483737Z"}
{"log":"10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
\n","stream":"stdout","time":"2023-03-04T04:54:21.923185324Z"}
{"log":"/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
\n","stream":"stdout","time":"2023-03-04T04:54:21.924294765Z"}
{"log":"/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
\n","stream":"stdout","time":"2023-03-04T04:54:21.93202447Z"}
{"log":"/docker-entrypoint.sh: Configuration complete; ready for start up
\n","stream":"stdout","time":"2023-03-04T04:54:21.933539655Z"}
{"log":"2023/03/04 04:54:21 [notice] 1#1: using the \"epoll\" event method
\n","stream":"stderr","time":"2023-03-04T04:54:21.945416265Z"}
{"log":"2023/03/04 04:54:21 [notice] 1#1: nginx/1.23.3
\n","stream":"stderr","time":"2023-03-04T04:54:21.945548014Z"}
{"log":"2023/03/04 04:54:21 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
```


**Figure 4.10.** Snippet of Apparmor complain logs on a ZAP session

Analyzing bpftrace logs for harmful syscalls and shared library calls like recv(), memcpy, and memset - applying rlimits and capability restrictions to prevent attacks.

Our experiments used bpftrace, Metasploit, OWASP ZAP, and ltrace. Because bpftrace can trace kernel and user-space events, we used it to measure system calls and shared library calls. OWASP ZAP simulated reconnaissance, while Metasploit tested containers. Finally, we attached ltrace to our AppArmor profile and Docker-sec's to compare their effectiveness.

We evaluated our security and identified areas for improvement by applying rlimits and capability restrictions and comparing our AppArmor profile to Docker-sec's generated one by tracing shared lib calls through ltrace for a different set of CVEs

### 4.3.1 Profile 1: Generated using AppArmor Complain Logs, Docker Documentation's Profile, and Docker-sec Default Profile

This profile is a combination of the base Nginx profile from Docker's documentation, the default Docker-sec profile, and additional rules generated from AppArmor complain logs. The profile includes additional restrictions on process memory and kernel access, as well as expanded rules for file access permissions. However, this profile does not incorporate information from bpftrace logs or container runtime analysis.

### 4.3.2 Profile 2: Generated using bpftrace Logs, and Additional Rules Based on Container Runtime Analysis

This profile extends the base Nginx profile from Docker's documentation with additional rules based on bpftrace logs and container runtime analysis. It is the most comprehensive and secure profile, as it incorporates information from multiple sources, including runtime behavior and bpftrace logs.

The CVE- list used to generated this is given above in the dataset for collection of the CVE's.

```
    # include < tunables / global >
@{PROC}=/proc/
profile bpftracenginx flags=(attach_disconnected,
mediate_deleted) {
# include < abstractions / base >

# Network access
network inet tcp,
network inet udp,
network inet icmp,

deny network raw,
deny network packet,
```

```
# General permissions
file,
umount,

# Deny write access to various directories
deny /bin/** wl,
deny /boot/** wl,
deny /dev/** wl,
deny /etc/** wl,
deny /home/** wl,

deny /lib/** wl,
deny /lib64/** wl,
deny /media/** wl,
deny /mnt/** wl,
deny /opt/** wl,
deny /proc/** wl,
deny /root/** wl,
deny /sbin/** wl,
deny /srv/** wl,
deny /tmp/** wl,
deny /sys/** wl,
deny /usr/** wl,

# Audit write attempts
audit /** w,

# Allow write access to specific files
```

```
  /var/run/nginx.pid w,

  # Allow execution of Nginx
  /usr/sbin/nginx ix,

  # Deny specific binaries
  deny /bin/dash mrwklx,
  deny /bin/sh mrwklx,
  deny /usr/bin/top mrwklx,

  # Capabilities
  capability chown,
  capability dac_override,
  capability setuid,
  capability setgid,
  capability net_bind_service,
  deny capability net_raw,
  deny capability sys_admin,
  deny capability sys_ptrace,
  deny capability sys_module,
  deny capability perfmon,
  deny capability sys_rawio,
  deny capability sys_time,
  deny capability ipc_lock,
  deny capability ipc_owner,
  deny capability sys_boot,
  deny capability sys_nice,
  deny capability sys_resource,
  deny capability setfcap,
```

```
deny capability syslog,
deny capability mknod,

# Deny write access to @{PROC}
deny @{PROC}/* w,
deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s]
[^1-9][^0-9][^0-9][^0-9]*}/** w,
deny @{PROC}/sys/[^k]** w,
deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,
deny @{PROC}/sysrq-trigger rwklx,
deny @{PROC}/mem rwklx,
deny @{PROC}/kmem rwklx,
deny @{PROC}/kcore rwklx,

# Deny mount operation
deny mount,

# Deny access to specific sys directories
deny /sys/firmware/** rwklx,
deny /sys/kernel/security/** rwklx,



# Restrict access to sensitive directories
deny /etc/ssh/** r,
/etc/passwd r,
/etc/shadow r,
/etc/group r,
deny /root/** r,
```

```
# Additional rules based on the opensnoop.bt
information and comparison
deny /proc/meminfo r,
deny /sys/fs/cgroup/user.slice/user-1000.slice/
user@1000.service/mem/** r,

#set the rlimits
set rlimit memlock <= 100M
set rlimit data < 10
}
```

Here is a breakdown of the profile:

1. Global rules and abstractions:

```
#include <tunables/global>
@{PROC}=/proc/
#include <abstractions/base>
```

These lines include global tunables and common abstractions to set up the basic rules for the profile. It also defines the PROC global variable.

2. Network Access:

```
network inet tcp,
network inet udp,
network inet icmp,

deny network raw,
deny network packet,
```

These rules allow common network protocols (TCP, UDP, ICMP) while denying raw and packet-level access. These restrictions are general security best practices and not derived from specific bpftrace logs.

3. Filesystem and mount access:

```
file,
umount,
```

These rules allow general file access and unmounting of filesystems.

4. Deny rules for specific directories:

```
deny /bin/** wl,
deny /boot/** wl,
deny /dev/** wl,
deny /etc/** wl,
deny /home/** wl,
deny /lib/** wl,
deny /lib64/** wl,
deny /media/** wl,
deny /mnt/** wl,
deny /opt/** wl,
deny /proc/** wl,
deny /root/** wl,
deny /sbin/** wl,
deny /srv/** wl,
deny /tmp/** wl,
deny /sys/** wl,
deny /usr/** wl,
```

These rules deny write access to various system directories. This is a general security best practice to prevent unauthorized modifications.

5. Audit and specific file access:

```
audit /** w,
/var/run/nginx.pid w,
```

These rules audit any write attempts and allow write access to the Nginx process ID file.

6. Nginx execution:

```
/usr/sbin/nginx ix,
```

This rule allows the execution of the Nginx binary.

7. Deny rules for potentially harmful binaries:

```
deny /bin/dash mrwklx,
deny /bin/sh mrwklx,
deny /usr/bin/top mrwklx,
```

These rules deny access to specific potentially harmful binaries.

8. Capabilities:

```
capability chown,
capability dac_override,
capability setuid,
capability setgid,
capability net_bind_service,


deny capability sys_admin,
deny capability sys_ptrace
```

These rules grant necessary capabilities for Nginx to function correctly. Sys_admin and sys_ptrace were denied on the basis of bpftrace logs.

9. Deny rules from {Proc}:

```
deny @{PROC}/* w,
deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s],[^1-9][^0-9]
[^0-9][^0-9]*}/** w,
deny @{PROC}/sys/[^k]** w,
deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,
deny @{PROC}/sysrq-trigger rwklx,
deny @{PROC}/mem rwklx,
deny @{PROC}/kmem rwklx,
deny @{PROC}/kcore rwklx,
```

These rules deny write access and certain read access to the @PROC filesystem, which is a general security best practice.

10. Deny mount and explicitly deny firmware and kernel access again:

```
deny mount,


deny /sys/firmware/** rwklx,
deny /sys/kernel/** rwklx,
```

11. Restrict access to sensitive directories:

```
deny /etc/ssh/** r,
/etc/passwd r,
/etc/shadow r,
/etc/group r,
deny /root/** r,
```

These rules restrict access to sensitive directories and files, which is a general security best practice. They are not derived from bpftrace logs but help to strengthen the security of the profile. The others need to be enabled.

12. Restrict repeatedness and limiting the calls:

```
 #set the rlimits
set rlimit memlock <= 100M
set rlimit data < 10
set rlimit nproc < 10
```

These rules set the rlimits to prevent the repeatedness and make the profile more effective in the face of exploit. The data rlimit sets limits for the data input byte for the nginx server. The memlock rlimit restricts memory resource usage. the nproc rlimit restricts the no. of processes that can be forked.

Overall, this updated AppArmor profile takes into consideration the information obtained from the opensnoop logs, bpftrace logs, and other logs provided. It allows for required

capabilities and file access permissions while denying potentially harmful capabilities and access to sensitive system directories. The profile aims to provide a secure environment for Nginx to operate in, based on the observed behavior from the logs.

### 4.3.3 Comparison of the Generated Profiles

We compared the generated profiles based on file access permissions, network access rules, and additional capabilities. Profile 2 emerged as the most comprehensive and secure AppArmor profile due to the incorporation of bpftrace logs and container runtime analysis. This profile provides better access control and restrictions tailored to the specific runtime behavior of the container.

## 4.4 Evaluation Results

The CVE List that is tested for obtaining the evaluation is:

**Table 4.2.** Comparison with docker-sec

| CVE | docker-sec generated profile | bpftrace generated profile |
|---|---|---|
| 2019-11510 | Restricted | Restricted |
| 2019-11043 | Restricted | Restricted |
| 2021-3156 | Not Restricted | Restricted |
| 2022-22965 | Restricted | Restricted |
| 2022-31137 | Not Restricted | Restricted |
| 2010-2263 | Not Restricted | Restricted |
| 2021-41773 | Not Restricted | Restricted |
| 2007-3010 | Restricted | Restricted |
| 2012-2329 | Not Restricted | Restricted |
| 2002-0392 | Not Restricted | Restricted |

The Ltrace output with PID, the library calls, and the data input values have been given:

```
Ltrace output:
2452 posix_memalign(0x7ffd7fa8e660, 16, 4096, 616) = 0
2452 memcpy(0x55c1a1df74c8, "host", 4)
= 0x55c1a1df74c8
```

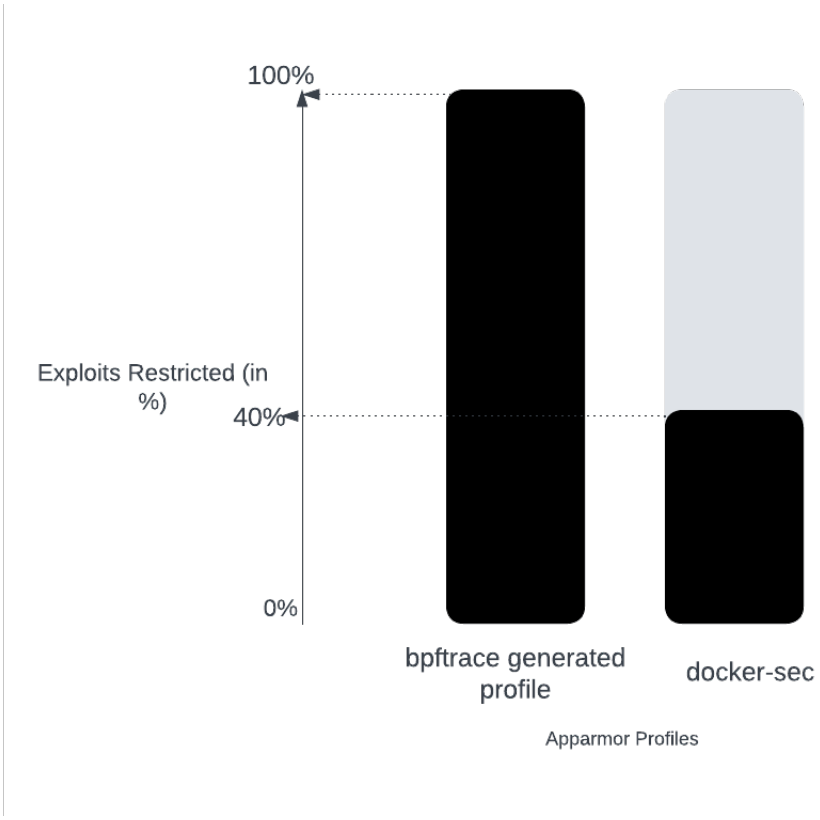


**Figure 4.11.** bpftrace generated profile vs docker-sec generated profile

```
2452 memcpy(0x55c1a1df74dc, "transfer-encoding", 17)
= 0x55c1a1df74dc
2452 memcpy(0x55c1a1df74ed, "/usr/share/nginx/html", 21)
= 0x55c1a1df74ed
2452 memcpy(0x55c1a1df7502, "/", 1)
= 0x55c1a1df7502
2452 memcpy(0x55c1a1df7503, "index.html\0", 11)
= 0x55c1a1df7503
2452 __xstat64(1, "/usr/share/nginx/html/index.html"...,
0x7ffd7fa8e340) = 0
2452 memset(0x55c1a1df7228, '\0', 480)
= 0x55c1a1df7228
```

```
2452 memcpy(0x55c1a1df750e, "/usr/share/nginx/html", 21)
= 0x55c1a1df750e
2452 memcpy(0x55c1a1df7523, "/index.html", 11)
= 0x55c1a1df7523
2452 open64("/usr/share/nginx/html/index.html"...,
2048, 00) = 3
2452 __fxstat64(1, 3, 0x7ffd7fa8e0e0)
= 0
2452 memset(0x55c1a1df7560, '\0', 80)
= 0x55c1a1df7560
2452 memset(0x55c1a1df75b0, '\0', 24)
= 0x55c1a1df75b0
2452 __memcpy_chk(0x7ffd7fa8d970,
0x55c1a1509dd0, 19, 2048) = 0x7ffd7fa8d970
2452 memcpy(0x7ffd7fa8d985, "error", 5)
= 0x7ffd7fa8d985
2452 syscall(186, 0x55c1a14b3142, 0x55c1a14b132d, 0) = 9
2452 memcpy(0x7ffd7fa8d98c, "9", 1)
= 0x7ffd7fa8d98c
2452 memcpy(0x7ffd7fa8d98e, "9", 1)
= 0x7ffd7fa8d98e
2452 memcpy(0x7ffd7fa8d992, "2", 1)
= 0x7ffd7fa8d992
2452 memcpy(0x7ffd7fa8d9be, "172.17.0.3", 10)
= 0x7ffd7fa8d9be
2452 memcpy(0x7ffd7fa8d9d2, "localhost", 9)
= 0x7ffd7fa8d9d2
2452 memcpy(0x7ffd7fa8d9e7, "GET / HTTP/1.1", 14)
= 0x7ffd7fa8d9e7
```

```
2452 memcpy(0x7ffd7fa8d9ff, "MvdCmgLPtVFcgyzk", 16)
= 0x7ffd7fa8d9ff
2452 write(4, "2023/04/05 17:10:16 [error] 9#9:"..., 161)
= 161
2452 memset(0x55c1a1df75c8, '\0', 80)
= 0x55c1a1df75c8
2452 memcpy(0x55c1a1df7621, "400 Bad Request", 15)
= 0x55c1a1df7621
2452 memcpy(0x55c1a1df764e,
"Wed, 05 Apr 2023 17:10:16 GMT", 29) = 0x55c1a1df764e
2452 memcpy(0x55c1a1df767b, "text/html", 9)
= 0x55c1a1df767b
2452 memcpy(0x55c1a1df7696, "157", 3)
= 0x55c1a1df7696
2452 memset(0x55c1a1df76d8, '\0', 80)
= 0x55c1a1df76d8
2452 memset(0x55c1a1dd80a0, '\0', 80)
= 0x55c1a1dd80a0
2452 memset(0x55c1a1dd80f0, '\0', 128)
= 0x55c1a1dd80f0
2452 writev(12, 0x7ffd7fa8dd70, 3, 0x55c1a14eff60) = 309
2452 shutdown(12, 1, 0x80000000, 0x642dab78)
= 0
2452 recv(12, 0x7ffd7fa8d5c0, 4096, 0)
= 4096
2452 recv(12, 0x7ffd7fa8d5c0, 4096, 0)
= 3
2452 epoll_wait(8, 0x55c1a1de9280, 512, 5000)
= 2
```

```
2452 gettimeofday(0x7ffd7fa8e630, 0)
= 0
2452 clock_gettime(1, 0x7ffd7fa8e680, 344, 0)
= 0
2452 accept4(6, 0x7ffd7fa8e640, 0x7ffd7fa8e63c, 2048)
= 11
2452 posix_memalign(0x7ffd7fa8e5c0, 16, 512, 0) = 0
2452 memcpy(0x55c1a1ddb020,
"\002\0\261\221\254\021\0\003\0\0\0\0\0\0\0",16)
= 0x55c1a1ddb020
2452 memcpy(0x55c1a1ddb080, "172", 3)
= 0x55c1a1ddb080
2452 memcpy(0x55c1a1ddb084, "17", 2)
= 0x55c1a1ddb084
2452 memcpy(0x55c1a1ddb087, "0", 1)
= 0x55c1a1ddb087
2452 memcpy(0x55c1a1ddb089, "3", 1)
= 0x55c1a1ddb089
2452 memset(0x55c1a1ddb090, '\0', 64)
= 0x55c1a1ddb090
2452 epoll_ctl(8, 1, 11, 0x7ffd7fa8e58c)
= 0
2452 recv(12, 0x7ffd7fa8d6c0, 4096, 0)
= 0
2452 memcpy(0x55c1a1dd81b0, "172.17.0.3", 10)
= 0x55c1a1dd81b0
2452 memcpy(0x55c1a1dd81c0,
"05/Apr/2023:17:10:16 +0000", 26) = 0x55c1a1dd81c0
2452 memcpy(0x55c1a1dd81dd, "GET / HTTP/1.1", 14)
```

```
= 0x55c1a1dd81dd
2452 memcpy(0x55c1a1dd81ed, "400", 3)
= 0x55c1a1dd81ed
2452 memcpy(0x55c1a1dd81f1, "157", 3)
= 0x55c1a1dd81f1
2452 write(5, "172.17.0.3 - - [05/Apr/2023:17:1"..., 81)
= 81
2452 close(3)
= 0
2452 free(0x55c1a1df6730)
= <void>
2452 free(0x55c1a1dd7c20)
= <void>
2452 close(12)
= 0
2452 free(0x55c1a1dc10b0)
= <void>
2452 free(0x55c1a1dc06a0)
= <void>
2452 epoll_wait(8, 0x55c1a1de9280, 512, 0xea60) = 1
2452 gettimeofday(0x7ffd7fa8e630, 0)
= 0
2452 clock_gettime(1, 0x7ffd7fa8e680, 358, 0)
= 0
2452 memset(0x55c1a1ddb0e8, '\0', 80)
= 0x55c1a1ddb0e8
2452 malloc(1024)
= 0x55c1a1dc10b0
```

```
2452 recv(11, 0x55c1a1dc10b0, 1024, 0)
= 1024
2452 ioctl(11, 21531, 0x55c1a1e073ec)
= 0
2452 posix_memalign(0x7ffd7fa8e630,
16, 4096, 0x7f30447505f7) = 0
2452 memset(0x55c1a1df6780, '\0', 1384)
= 0x55c1a1df6780
2452 memset(0x55c1a1df7228, '\0', 480)
= 0x55c1a1df7228
2452 memset(0x55c1a1df7408, '\0', 192)
= 0x55c1a1df7408
2452 posix_memalign(0x7ffd7fa8e660, 16, 4096, 616) = 0
2452 memcpy(0x55c1a1df74c8, "host", 4)
= 0x55c1a1df74c8
2452 memcpy(0x55c1a1df74dc, "transfer-encoding", 17)
= 0x55c1a1df74dc
2452 memcpy(0x55c1a1df74ed, "/usr/share/nginx/html", 21)
= 0x55c1a1df74ed
2452 write(4, "2023/04/05 17:10:16 [error] 9#9:"..., 161)
= 161
```

This is a snippet of the logs generated by nginx, when cve:2013-2028 is attempted repeatedly, without the use of our apparmor profile. We can observe repeated calls to shared libraries, which were subsequently reduced when our profile was applied.

On the other hand, the container when secured with our apparmor profile which limited the calls, we could not see any memset() or memcpy() or close(0 or xstat() calls by the nginx container.

Here is a comparison of the three AppArmor profiles:

Profile 1: Generated using AppArmor complain logs, Docker documentation's profile, and Docker-sec default profile

This profile is a combination of the base Nginx profile from Docker's documentation, the default Docker-sec profile, and additional rules generated from AppArmor complain logs. The profile includes additional restrictions on process memory and kernel access, as well as expanded rules for file access permissions. However, this profile does not incorporate information from bpftrace logs or container runtime analysis.

Profile 2: Generated using bpftrace logs, Docker documentation's profile, and additional rules based on container runtime analysis

This profile extends the base Nginx profile from Docker's documentation with additional rules based on bpftrace logs and container runtime analysis. It is the most comprehensive and secure profile, as it incorporates information from multiple sources, including runtime behavior and bpftrace logs.

Comparison of the two profiles:

File access permissions:

Profile 1 adds file access permissions from the complain logs and Docker-sec default profile. Profile 2 further refines file access permissions based on bpftrace logs and container runtime analysis.

Network access rules:

Both profiles share the same basic network access rules, including allowing TCP, UDP, and ICMP, while denying raw and packet network access. Additional capabilities:

Profile 1 denies some capabilities, but allows most of them according to the Docker-sec default profile. Profile 2 further refines the capabilities by considering bpftrace logs and container runtime analysis.

It follows the experimentation results by denying the specific capabilties in the attacks by denying them which are corresponding to the specific syscalls as given below:

- deny ptrace = deny capability sys_ptrace
- deny perf_event_open = deny capability sys_perf_event_open
- deny process_vm_readv = deny capability sys_process_vm_readv

- deny process_vm_writev = deny capability sys_process_vm_writev
- deny create_module = deny capability sys_module
- deny init_module = deny capability sys_module
- deny delete_module = deny capability sys_module
- deny ioperm = deny capability sys_rawio
- deny iopl = deny capability sys_rawio
- deny socketcall = deny capability sys_socket
- deny reboot = deny capability sys_reboot
- deny settimeofday = deny capability sys_time
- deny adjtimex = deny capability sys_time
- deny clock_settime = deny capability sys_time
- deny modify_ldt = deny capability sys_module

These subsequently help prevent all the cve modules attempted, and also provide ltrace evalution which shows a 40% more success when exploited in docker web servers in terms of CVEs prevented. They show a reduction of shared library calls (memset, memcpy), as evidenced by the ltrace output.

In conclusion, Profile 2 offers the most comprehensive and secure AppArmor profile due to the incorporation of bpftrace logs and container runtime analysis, providing better access control and restrictions tailored to the specific runtime behavior of the container.

### 4.5 Summary

In chapter 4, we focused on generating and evaluating AppArmor profiles for Nginx container and httpd container using various sources of information and analysis methods.

Profile 1: Generated using docker-sec which was trained on nginx and apache httpd containers.

Profile 2: Generated using bpftrace logs, Docker documentation's profile, and additional rules based on container runtime analysis, further refining file access permissions, capabilities, and restrictions based on bpftrace logs and container runtime analysis.

Profile 2 emerged as the most comprehensive and secure AppArmor profile due to the incorporation of bpftrace logs and container runtime analysis, providing better access control and resource restrictions as defined by rlimits tailored to the specific runtime behavior of the container, in our case web servers like nginx and apache.

A comparison of the two profiles was conducted based on file access permissions, network access rules, and additional capabilities. The evaluation results demonstrated the advantages of Profile 2 over the other profiles in terms of access control and security.

To measure we used:

- ltrace: ltrace is a tool that can be used to trace the execution of a program and log all system calls. This can be useful for finding vulnerabilities that are caused by incorrect system calls.

In the ltrace output we saw that the amount of memcpy and memset calls along with others such as ioctl calls, and recv() calls were negligible in profile 2, as compared to profile 1, it being generated by docker-sec.

# 5. CONCLUSION AND FUTURE WORK

## 5.1 Conclusion

In this study, we investigated the innovative application of AppArmor profiles to enhance the security of containers, specifically focusing on an Nginx and Apache containers acting as web servers. We explored the generation of AppArmor profiles using various sources of information and analysis methods.

Profile 2, which was generated using BPFtrace logs, Docker documentation's profile, and additional rules based on container runtime analysis, emerged as the most restrictive and secure AppArmor profile as evidenced by the ltrace output. It incorporated information from multiple sources and was tailored to the specific runtime behavior of the container, providing better access control and restrictions as well as incorporating rlmits to prevent dos type attacks. This innovation of utilizing bpftrace logs to generate AppArmor profiles addresses the security concerns associated with web server containers, such as Nginx, and improves the overall container security posture.

## 5.2 Using bpftrace to generate apparmor profiles

Bpftrace is a high-level tracing language and runtime for eBPF (Extended Berkeley Packet Filter). eBPF is a low-level virtual machine running inside the Linux kernel, which can be used for a wide range of purposes, including networking, security, and performance analysis. bpftrace provides a more user-friendly and expressive interface to write eBPF programs, making it easier to create and run eBPF tracing tools.

eBPF allows you to write programs that can be attached to various parts of the kernel, such as system calls, kernel functions, and networking events. These programs can collect data, filter packets, or enforce security policies. eBPF programs are highly efficient and have minimal impact on system performance.

This can be used as was used in our custom bpftrace script to attach uprobe to a tracepoint and trace the memory resource utlization levels as well.

In conclusion, the innovative use of bpftrace in generating AppArmor profiles lies in its ability to provide fine-grained analysis, tailored security policies, and real-time monitoring of container behavior. This approach ensures a more secure environment for containerized applications and contributes to the overall security of container deployments.

### 5.3 Future Work

While this study demonstrates the advantages of using AppArmor profiles generated from eBPF logs and container runtime analysis, there are several avenues for future research:

1. Automation of profile generation: Developing a tool or framework that automatically generates AppArmor profiles based on eBPF logs and runtime analysis could streamline the process and reduce the manual effort required to create secure profiles.

2. Evaluation with other container technologies: Testing the generated AppArmor profiles with other container technologies, such as containerd or CRI-O, would help assess the generalizability of the proposed approach.

3. Integration with other security mechanisms: Investigating the combination of AppArmor profiles with other container security mechanisms, such as seccomp or SELinux, could lead to a more comprehensive and robust security solution.

4. Evaluation with various containerized applications: Conducting a similar analysis for different containerized applications would help to assess the effectiveness of the proposed approach across a broader range of scenarios.

5. Real-world attack scenarios: Testing the generated AppArmor profiles in real-world attack scenarios could provide valuable insights into their effectiveness in mitigating specific threats and vulnerabilities.

In conclusion, this study demonstrates the potential of using AppArmor profiles, generated from various sources and analysis methods, to enhance container security. The results show that profiles generated using eBPF logs and container runtime analysis are more comprehensive and secure. The innovation of utilizing bpftrace logs to generate AppArmor

profiles can effectively address web server container security concerns. Future research can build on these findings to further improve the security of containerized applications and develop automated tools to facilitate the generation of AppArmor profiles.

# REFERENCES


Aich, R. (2021). *Efficient audit data collection for linux* (Doctoral dissertation). Stony Brook University.

Bao, A. C. (2023). Is docker secure for web server. https://www.alibabacloud.com/tech-news/web-server/gipubz2iq7-is-docker-secure-for-web-server#:~:text=What%5C%20is%5C%20Docker%5C%3F-,Docker%5C%20is%5C%20an%5C%20open%5C%2Dsource%5C%20platform%5C%20that%5C%20enables%5C%20developers%5C%20to,scalability%5C%2C%5C%20flexibility%5C%2C%5C%20and%5C%20security.

Bélair, M., Laniepce, S., & Menaud, J.-M. (2019). Leveraging kernel security mechanisms to improve container security: A survey. *Proceedings of the 14th international conference on availability, reliability and security*, 1–6.

Bentaleb, O., Belloum, A. S., Sebaa, A., & El-Maouhab, A. (2022). Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*, *78*(1), 1144–1181.

Bhardwaj, A., & Krishna, C. R. (2021). Virtualization in cloud computing: Moving from hypervisor to containerizationa survey. *Arabian Journal for Science and Engineering*, *46*(9), 8585–8601.

Brady, K., Moon, S., Nguyen, T., & Coffman, J. (2020). Docker container security in cloud computing. *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 0975–0980.

Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv:1501.02967*.

Chelladhurai, J., Chelliah, P. R., & Kumar, S. A. (2016). Securing docker containers from denial of service (dos) attacks. *2016 IEEE International Conference on Services Computing (SCC)*, 856–859.

Combe, T., Martin, A., & Di Pietro, R. (2016). To docker or not to docker: A security perspective. *IEEE Cloud Computing*, *3*(5), 54–62.

Cowan, C. (2007). Securing linux systems with apparmor - linux australia. http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/198.pdf

docs.docker.com. (2023). Apparmor security profiles for docker. https://docs.docker.com/engine/security/apparmor/

Gao, X., Gu, Z., Li, Z., Jamjoom, H., & Wang, C. (2019). Houdini's escape: Breaking the resource rein of linux control groups. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 1073–1086.

Goniwada, S. R. (2022). Containerization and virtualization. In *Cloud native architecture and design: A handbook for modern day architecture and design with enterprise-grade examples* (pp. 573–617). Apress. https://doi.org/10.1007/978-1-4842-7226-8_16

Gregg, B. (2015). Linux performance testing tools. https://brendangregg.com/linuxperf.html

Jader, O. H., Zeebaree, S., & Zebari, R. R. (2019). A state of art survey for web server performance measurement and load balancing mechanisms. *International Journal of Scientific & Technology Research*, *8*(12), 535–543.

Jain, V., Singh, B., & Choudhary, N. (2022). Audit and analysis of docker tools for vulnerability detection and tasks execution in secure environment. *International Conference on Emerging Technologies in Computer Engineering*, 654–665.

Kaiser, S., Haq, M. S., Tosun, A. S., & Korkmaz, T. (2022). Container technologies for arm architecture: A comprehensive survey of the state-of-the-art. *IEEE access: practical innovations, open solutions*, *10*, 84853–84881. https://doi.org/10.1109/access.2022.3197151

Kaur, P., Josan, J. K., & Neeru, N. (2022a). Performance analysis of docker containerization and virtualization. *Proceedings of Third International Conference on Communication, Computing and Electronics Systems*, 863–877.

Kaur, P., Josan, J. K., & Neeru, N. (2022b). Performance analysis of docker containerization and virtualization. In V. Bindhu, J. M. R. S. Tavares, & K.-L. Du (Eds.), *Proceedings of third international conference on communication, computing and electronics systems* (pp. 863–877). Springer Singapore.

Kelly, C., Pitropakis, N., Mylonas, A., McKeown, S., & Buchanan, W. J. (2021). A comparative analysis of honeypots on different cloud platforms. *Sensors (Basel)*, *21*(7), 2433.

Kim, S., Kim, B. J., & Lee, D. H. (2021). Prof-gen: Practical study on system call whitelist generation for container attack surface reduction. *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 278–287.

Kithulwatta, W. M. C. J. T., Jayasena, K. P. N., Kumara, B. T. G. S., & Rathnayaka, R. M. K. T. (2022). Performance evaluation of docker-based apache and nginx web server. *2022 3rd International Conference for Emerging Technology (INCET)*, 1–6. https://doi.org/10.1109/INCET54531.2022.9824303

Leahy, D., & Thorpe, C. (2022). Zero trust container architecture (ztca): A framework for applying zero trust principals to docker containers. *International Conference on Cyber Warfare and Security*, *17*(1), 111–120.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., & Zhou, Q. (2018). A measurement study on linux container security: Attacks and countermeasures. *Proceedings of the 34th Annual Computer Security Applications Conference*, 418–429.

Loukidis-Andreou, F., Giannakopoulos, I., Doka, K., & Koziris, N. (2018). Docker-sec: A fully automated container security enhancement mechanism. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 1561–1564.

Manu, A., Patel, J. K., Akhtar, S., Agrawal, V., & Murthy, K. B. S. (2016). A study, analysis and deep dive on cloud paas security in terms of docker container security. *2016 international conference on circuit, power and computing technologies (ICCPCT)*, 1–13.

Martin, A., Raponi, S., Combe, T., & Di Pietro, R. (2018). Docker ecosystem–vulnerability analysis. *Computer Communications*, *122*, 30–43.

Morabito, R., Kjällman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. *2015 IEEE International Conference on cloud engineering*, 386–393.

MP, A. R., Kumar, A., Pai, S. J., & Gopal, A. (2016). Enhancing security of docker using linux hardening techniques. *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 94–99.

Nurwarsito, H., & Sejahtera, V. B. (2020). Implementation of dynamic web server based on operating system-level virtualization using docker stack. *2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 33–38. https://doi.org/10.1109/ICITEE49829.2020.9271710

Overview. (2022). https://docs.docker.com/get-started/

Preeth, E., Mulerickal, F. J. P., Paul, B., & Sastri, Y. (2015). Evaluation of docker containers based on hardware utilization. *2015 international conference on control communication & computing India (ICCC)*, 697–700.

Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, *17*(3), 228.

Rankin, K., & Hill, B. M. (2014). *The official ubuntu server book*. Pearson Education.

Saraiva de Sousa, N. F., Lachos Perez, D. A., Rosa, R. V., Santos, M. A. S., & Esteve Rothenberg, C. (2019). Network service orchestration: A survey. *Comput. Commun.*, *142-143*, 69–94.

Schreuders, Z. C., McGill, T., & Payne, C. (2011). Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm. *ACM Trans. Inf. Syst. Secur.*, *14*(2). https://doi.org/10.1145/2019599.2019604

Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE Access*, *7*, 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732

Tomar, A., Jeena, D., Mishra, P., & Bisht, R. (2020). Docker security: A threat model, attack taxonomy and real-time attack scenario of dos. *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 150–155.

Tunde-Onadele, O., He, J., Dai, T., & Gu, X. (2019). A study on container vulnerability exploit detection. *2019 IEEE International Conference on Cloud Engineering (IC2E)*, 121–127.

Wenhao, J., & Zheng, L. (2020). Vulnerability analysis and security research of docker container. *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 354–357.

Wist, K., Helsem, M., & Gligoroski, D. (2021). Vulnerability analysis of 2500 docker hub images. In *Advances in security, networks, and internet of things* (pp. 307–327). Springer.

Yasrab, R. (2018). Mitigating docker security issues. *arXiv preprint arXiv:1804.05039*.

Zhimin, G., Zhuo, L., Jiaxuan, F., Xiangqun, W., Gui, Y., Nuannuan, L., & Cen, C. (2021). Research on security hardening technology of container image file configuration for power intelligent iot terminal. *2021 IEEE Sustainable Power and Energy Conference (iSPEC)*, 4227–4232. https://doi.org/10.1109/iSPEC53008.2021.9735992

Zhu, H., & Gehrmann, C. (2021). Lic-sec: An enhanced apparmor docker security profile generator. *Journal of Information Security and Applications*, *61*, 102924.

Zhu, H., & Gehrmann, C. (2022). Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 129–137.