POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Evaluating the performance of eBPF-based security software in a virtualized 5G cluster

Supervisors

Prof. Fulvio RISSO

Dr. David SOLDANI

Candidate

Daniel Calin PANAITE

July 2024

Summary

Since the arrival of 5G technology, there has been a shift towards virtualization and the use of containers instead of bare-metal machines. A significant factor enabling this shift is the use of Kubernetes to deploy and manage a large cluster of machines that together enable a 5G network to function.

While the cluster can run smoothly and without interruption, there is a severe need for strong security measures to prevent frequent attacks that could hinder the availability of our network. Tetragon is one of these security measures, providing robust observability and enforcement capabilities to enhance the security of our cluster. Through the use of eBPF, it is not only fast but also uses minimal resources to accomplish its goal.

By studying this tool, we can determine if it can be deployed within our 5G network and if it is capable of covering as many security use cases as possible. Initially, we focused on learning how to best use Tetragon and understanding how it works at a low level, leveraging eBPF and accessing Linux syscalls and the network layer for extensive monitoring. Later, we utilized Tetragon to create demonstrations that simulated its usage within our network, deciding whether it would be feasible to use it in a production environment.

After extensive research and testing, we found that while Tetragon has plenty of good use cases, there are just as many that require further support from machine learning to properly detect certain classes of events.

Acknowledgements

Voglio dedicare questo spazio a tutte quelle persone che, con il loro supporto, mi hanno aiutato in questo meraviglioso percorso accademico. Ringrazio il mio relatore, professor Fulvio Risso, per avermi aiutato e guidato nella tesi oltre ad avermi fornito questa stupenda opportunità.

Ringrazio anche Rakuten che mi ha accolto per questo progetto, soppratutto il mio supervisore aziendale David Soldani e i membri del team che mi hanno aiutato a svolgere questo lavoro, Hami Bour e Saber Jafarizadeh.

Non mancano di certo i ringraziamenti a quelle persone che mi sono state sempre vicine, sia nei momenti belli che in quelli più difficili. Grazie Mamma e Papà per essermi sempre stati vicino in questo lungo percorso universitario. Ringrazio anche al resto della mia famiglia che mi ha sempre supportato in tutte le mie scelte.

Desidero infine ringraziare i miei coinquilini nonché compagni di liceo, Alessandro e Andrea, con cui ho condiviso questi meravigliosi anni di universitá. Non possono mancare i ringraziamenti a tutti quei amici e compagni che mi hanno accompagnato lungo questo percorso.

Table of Contents

List of Tables

List of Figures

Acronyms

API
: Application Programming Interface

BPF
: Berkeley Packet Filter

CNI
: Container Network Interface

CPU
: Central Processing Unit

CR
: Custom Resource

CRD
: Custom Resource Definition

eBPF
: Extended Berkeley Packet Filter

FAM
: File Access Monitoring

FD
: File Descriptor

FIFO
: First-In-First-Out

FIM

File Integrity Monitoring

HPA

Horizontal Pod Autoscaler

HTTP

Hypertext Transfer Protocol

IP

Internet Protocol

JIT

Just-In-Time

JSON

JavaScript Object Notation

LIFO

Last-In-First-Out

NFV

Network Function Virtualization

RBAC

Role-Based Access Contro

REST

Representational State Transfer

SDN

Software-Defined Networking

SSL

Secure Sockets Layer

VM

Virtual Machine

VPA	Vertical Pod Autoscaler
YAML	Yet Another Markup Language

Chapter 1

Introduction

Since the first iteration of the Berkeley Packet Filter in 1992 many advancements have been made in improving the performance and usefulness of this tool. Ten years ago its modern counterpart eBPF was introduced, extending even further its capabilities of enhancing the Linux kernel to this day. Thanks to this technology, today we are able to more closely observe what is happening inside our systems from what files are being read without our knowledge to who is trying to access our computer from the outside, be it a regular user or a malicious one.

With the increase in cyber attacks in recent years there is an even more pressing need for tools like this to increase the security of servers where sensitive and important data is being stored. Other tools have existed for years that tried to guarantee the safety of a system but, because it was software running in user space, they were limited in terms of what they could accomplish. With eBPF having access to information directly within the kernel allows us to have even more control over our system.

Tetragon is a tool that leverages eBPF in order to observe and also enforce user-defined rules called policies within a Kubernetes cluster. They require significant knowledge of the Linux kernel and in the hands of an experienced security team most if not all malicious actors can be denied access to our system and sensitive data.

1.1 Current solution

In the evolving landscape of 5G networks, Sauron stands out as a sophisticated eBPF-based platform tailored for comprehensive observability, networking, and security monitoring. As our existing solution, Sauron integrates seamlessly within our Kubernetes cluster to monitor and secure the k8s nodes through the use of modules that when combined give the user a broad view of the entire system. For network observability it uses one Controller for the entire cluster and Node Agents for each node that belongs to it. The task of the Controller is to gather information from the Kubernetes API server and attach said data to each collected event inside the system. The collection of these events is handled inside the kernel by eBPF so the information from the API server is crucial to have a broader view of the event since more information on it can only be retrieved in user space.

The Node Agent gathers more information from the API server, this time of the Pods running on the node. It also loads the eBPF probes int other kernel and attaches it to XDP hooks, system calls and interfaces of each pod. The way in which these modules communicate with the kernel is through eBPF maps that allow data to be shared between the eBPF probes and the Node Agent running in user space.

Having described the architecture of the Sauron platform it is also important to properly illustrate how it works inside a Linux system. The Sauron Agent is installed as a native app and it employs the use of an eBPF program to collect the necessary data after the agent has loaded the program and attached it to the XDP hook. In the case of a router an eBPF program could be written to collect latency measurements and it also provides an interface to configure the desired measurements. All this requires the user to write their own eBPF program using a programming language such as C with the use of a library such as libbpf, increasing the complexity of the platform.

1.2 The challenge

While Sauron offers a robust solution for eBPF monitoring, developing and maintaining our own platform can be costly and resource-intensive. Fortunately, there are several alternatives available that provide comprehensive observability, security, and monitoring capabilities. Tetragon, for instance, leverages eBPF to deliver deep visibility into system calls and network activities, enabling real-time threat detection and policy enforcement. KubeArmor focuses on runtime security for Kubernetes, providing system hardening and behavior-based security controls. Tracee, an open-source runtime security and forensics tool, uses eBPF to detect and alert on suspicious activities at the kernel level. Lastly, Falco, a popular open-source runtime security project, uses eBPF to monitor system behavior and detect anomalies, offering extensive configurability and community support. These tools present viable alternatives to developing an in-house solution, offering rich features and reducing the overhead associated with building and maintaining custom infrastructure.

Recognizing the availability of numerous eBPF monitoring tools, we have decided to focus our evaluation on Tetragon. We will assess its performance, technical feasibility and business viability. Tetragon stands out due to its ease of use, flexibility, and the range of supported use cases:

- Usability: Tetragon is known for its straightforward installation and configuration process, which does not rely on additional components like Cilium. This simplicity makes it easier to deploy and manage in diverse environments, including Kubernetes, Docker, and plain Linux.
- Flexibility: One of the standout features of Tetragon is its remarkable flexibility. Users can create and customize policies to define specific monitoring and security parameters tailored to their unique requirements. This level of customization allows organizations to adapt Tetragon to a wide variety of use cases, ensuring that it meets their specific needs.
- Supported Use Cases: Tetragon excels in several critical use cases, making it a versatile tool for modern infrastructure monitoring and security. For real-time threat detection, Tetragon monitors system calls and network activities, identifying and responding to security threats as they occur. This proactive approach helps protect systems from potential breaches and malicious activities.

All these strengths make Tetragon the ideal choice for monitoring our 5G network cluster. It combines ease of use, which ensures a faster learning curve, with a comprehensive set of features that support all our use cases effectively.

1.3 Testing

To effectively evaluate the performance of our various eBPF monitoring tools, we have established a comprehensive benchmark designed to test how each tool performs against a standardized workload. This approach allows us to make direct comparisons and determine which tool best meets our needs. Our benchmark will include Tetragon, KubeArmor, and Tracee, as these tools are the most comparable in terms of usability and ease of use. By assessing these tools under the same conditions, we aim to identify their strengths and weaknesses, ensuring that our final choice is well-informed and tailored to our specific requirements. This evaluation will help us determine the most suitable solution for monitoring our 5G network cluster effectively.

1.4 Thesis overview

The thesis will expand on the current subject dividing it in the following chapters:

- **2 - Kubernetes**: describes what Kubernetes is and how this container orchestration tool helped in developing and researching this topic.
- **3 - Grafana**: an overview of a powerful software stack that helps in the creation of our observability platform by enabling data collection and visualization.
- **4 - Tetragon**: illustrates how Tetragon works, enabling powerful eBPF observability and enforcement.
- **5 - Implementations**: showcases our proposed solution for this thesis.
- **6 - Measurements**: contains a series of results gathered from benchmarks of our system running Tetragon and several other similar tools for comparison.
- **7 - Conclusion**: shows the results achieved so far and what could be possibly done in the future.

Chapter 2

Kubernetes

Kubernetes is a powerful open-source platform designed to automate deploying, scaling, and operating application containers across clusters of hosts [1]. Its name originates from Greek, meaning helmsman or pilot and is often abbreviated as K8s. It provides the infrastructure to build a truly container-centric development environment. Kubernetes orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This means it handles scheduling containers on a cluster, managing the workloads to ensure they run as the user intended, and scaling applications up or down based on demand, thus optimizing resource utilization.

2.1 History

The story of Kubernetes is fundamentally rooted in addressing the practical challenges of managing containerized applications at scale. Originating from Google, which has extensive experience in running production-grade containerized workloads, Kubernetes was officially introduced as an open-source project in mid-2014. The initial development was spearheaded by a team of Google engineers who had previously worked on Borg, Google's internal container orchestration system. Borg itself was pivotal in forming the design and functionality of Kubernetes, embedding years of Google's operational experience directly into the new platform.

The transition from Borg to Kubernetes was marked by significant enhancements aimed at addressing various user pain points identified in Borg over the years. While Borg was highly effective within Google, it was also recognized as complex and tightly integrated into Google's specific infrastructure, making it less ideal for open-source release. Kubernetes, in contrast, was designed to be more portable and flexible, capable of running across different environments, from private clouds to public clouds and hybrid systems [2].

Joining the initial team of Google developers were experts from Red Hat and CoreOS, who contributed additional insights and capabilities, broadening the scope and adaptability of Kubernetes. This collaboration underlined Kubernetes' commitment to community-led development, a principle that has remained at the core of its evolution. The input from a diverse set of contributors helped ensure that Kubernetes was not just a tool for large enterprises like Google but was accessible and useful for organizations of all sizes.

Since its inception, Kubernetes has rapidly evolved, supported by an ever-growing community of developers and users. It was donated to the Cloud Native Computing Foundation (CNCF) in 2015, a move that further cemented its position as a cornerstone of the cloud-native ecosystem. Under the CNCF, Kubernetes has flourished, becoming the leading orchestration tool that defines the cloud-native landscape, continually expanding its feature set and improving its robustness and efficiency.

This chapter on the history of Kubernetes not only highlights its origins and technical lineage but also sets the stage for understanding its impact on modern software development and deployment practices. The ongoing evolution of Kubernetes reflects its foundational goal of improving the scalability and manageability of containerized applications in a diverse array of environments.

2.2 From monolithic applications to containers

The evolution of application architecture from monolithic models to containerized environments represents a significant shift in how software is deployed and managed, greatly impacting development efficiency and operational flexibility [3].

Historically, monolithic applications were the standard approach in software development. In this model, all components of an application, ranging from input handling to data processing and UI rendering, were tightly integrated into a single, indistinguishable unit, usually deployed on a single physical server. This architecture simplified the development process in some ways but posed significant challenges. Resource allocation was a major issue, as all components shared the same underlying resources, leading to potential bottlenecks where one component could consume disproportionate resources, starving others.

The introduction of virtualization technology marked a pivotal shift, allowing multiple Virtual Machines (VMs) to run on a single physical server. Each VM operated as a distinct entity with its own full copy of an operating system, libraries, and application files, providing better isolation compared to traditional monolithic deployments. This meant that applications could be isolated from one another, enhancing security by preventing applications from accessing each other's data. Virtualization also improved resource utilization by allowing unused resources in one VM to be allocated to others that might need them more urgently. However, this approach also introduced overhead, as each VM required its own operating system and full set of resources, leading to underutilization of hardware capabilities.

Containers emerged as a more resource-efficient solution compared to VMs, enabling even greater scalability and flexibility. Unlike VMs, containers share the host operating system's kernel but can run isolated processes in user space. They provide lightweight execution environments that package applications and their dependencies together. This not only reduces overhead but also enhances portability across different computing environments. Containers support microservices architecture, allowing developers to decompose applications into smaller, loosely coupled services that can be developed, deployed, and scaled independently. This modularity enables teams to adopt different technologies for different services and to scale or update individual components without impacting others.

2.3 Kubernetes architecture

In the following chapter we will illustrate the architecture of Kubernetes and how each of its components interact with each other. The core of the architecture are its machines, divided between control plane and the nodes. Each node can either be a physical or virtual machine that runs the pods, composed of containers [4].

2.3.1 Control Plane

The Kubernetes control plane is the cornerstone of Kubernetes architecture, responsible for managing the cluster state and configuration. At its core, the control plane ensures that the containerized applications running on Kubernetes are in the desired state specified by the user.

Its components are:

- **API Server**: The Kubernetes API Server acts as the front end to the control plane, exposing the Kubernetes API and serving as the gateway through which all internal and external communications pass. It processes Representational State Transfer (REST) requests, validates them, executes the backend logic, and updates the corresponding objects in the etcd, ensuring that Kubernetes users can configure workloads and organizational units declaratively.

- **etcd**: A consistent and highly-available key value store used as the backing store for all cluster data. etcd stores and replicates the Kubernetes cluster state.

- **Scheduler**: The Scheduler watches for newly created Pods that have no assigned node, and selects a node for them to run on based on resource availability, constraints, affinity specifications and other factors. It is crucial for optimizing workload distribution and resource utilization across the cluster.

- **Controller Manager**: This component runs controller processes, which are background threads that handle routine tasks in the cluster. It watches the objects it manages and checks if the various elements are in their desired states via the API Server. If the states do not match it takes action and corrects any issues.
- **Cloud Controller Manager**: It is a control plane component that embeds cloud-specific logic. The cloud controller manager lets your cluster interact with specific API of your cloud provider. If you are running your own cluster it will not have this component.

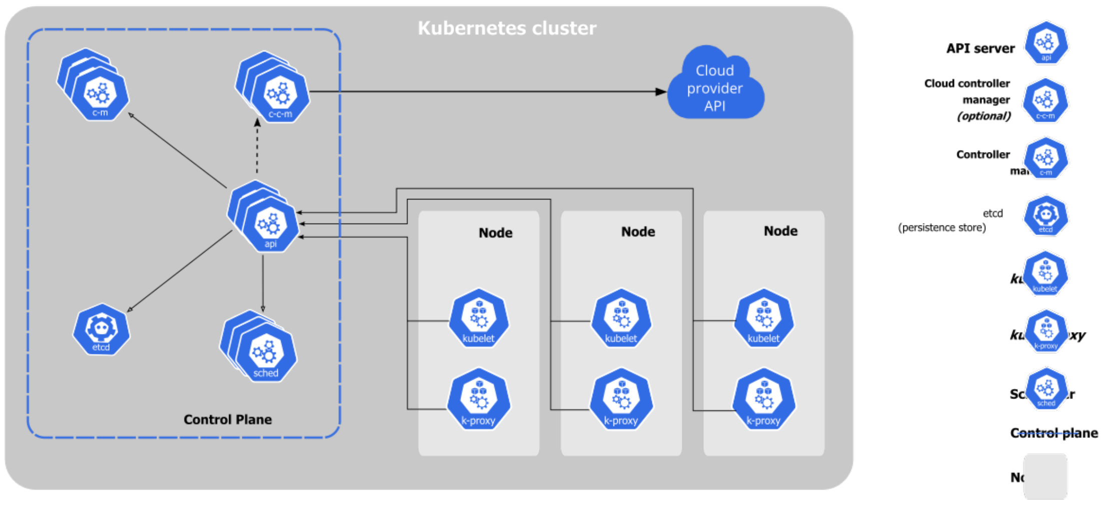


Figure 2.1: Kubernetes architecture

2.3.2 Nodes

Nodes are key elements of a cluster, consisting of any machine that is not a control plane. They are vital for a cluster to function since containers are put into pods in order to run on nodes.

There are several components that guarantee this functionality:

- **kubelet**: It is an agent that runs on each node and it makes sure that containers are running in a pod according to a PodSpec. A PodSpec is a YAML or JSON object that identifies specific settings for the container that is running within a pod.

- **kube-proxy**: This component is a network proxy that runs on each node and guarantees connectivity to the pods running within. While it is only part of the Kubernetes Service concept that we will discuss later it maintains network rules on nodes to allow network communication inside or outside the cluster.
- **Container runtime**: A fundamental component that allows Kubernetes to run containers and supports implementations such as containerd, CRI-O and others.

2.4 Kubernetes components

Having seen an overview of the Kubernetes architecture we will now delve deeper into understanding which core components are vital for a cluster to operate. Understanding what these components are is crucial for efficiently managing and scaling Kubernetes environments.

2.4.1 Resources

Resources in Kubernetes are mainly defined by CPU, memory, storage and bandwidth but there are others. These can be specified for each pod, service, deployment or other components and are separated into two: requests and limits.

The first, requests, are the amount of resources that are needed for the pod to operate while the limit is the maximum amount. The kubelet will distribute the system resources between pods keeping in mind these limits to ensure a smooth operation of the cluster.

2.4.2 Objects

Pods

There are several types of objects in a Kubernetes cluster, with the simplest being a Pod. It is composed of one or more containers that share storage and network resources and is the smallest unit inside a cluster. These highly coupled containers will be always scheduled on the same node since they belong to the same pod.

With the use of Linux namespaces the containers have both private and shared areas to facilitate sharing information.

Listing 2.1: Example of a Pod resource YAML

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: nginx
```

```yaml
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
```

Namespace

This object is only an abstraction and it serves to create multiple "logical clusters" where pods can ping each other and can be constrained by shared rules using Role-based access control(RBAC).

Deployment

Users do not manage pods directly but they create a deployment where a set of rules are specified. It can manage the version of a software and how many replicas of the pod will be created using ReplicaSets. Similar to a deployment are StatefulSets which are valuable for applications that need to have persistent storage.

Service

In order to access an application from outside the cluster it needs to be exposed using a service. This abstraction separates the execution of a pod from its exposed endpoint. Restarting and updating a pod will not have any impact on its service which provides connectivity to outside the cluster. An application can be exposed in different ways using services of various types: ClusterIP, NodePort, LoadBalancer, ExternalName.

Listing 2.2: Example of a Service resource YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Ingress

If a service needs to be reachable from the internet it requires an additional resource, called Ingress. While the cluster only has one external IP it is able to multiplex various services to it.

Listing 2.3: Example of a Ingress resource YAML

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

2.5 Kubernetes networking

Kubernetes networking can be complex but is crucial for ensuring that applications running in a Kubernetes cluster can communicate efficiently and securely. This chapter aims to simplify the fundamental concepts of Kubernetes networking for better comprehension and application.

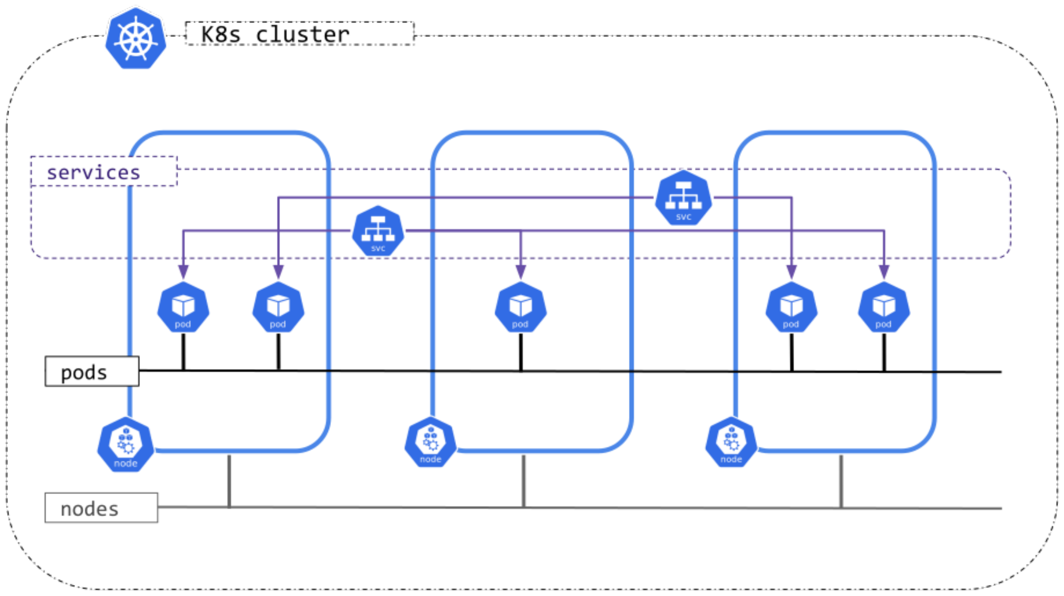


Figure 2.2: Kubernetes networking

Network Model

Kubernetes requires all pods to communicate with each other without the need for NAT (Network Address Translation). Every pod gets its own cluster-wide unique IP address, meaning you do not need to explicitly link pods or manage port conflicts.

Networking Components

Pod Networking: Each pod in a Kubernetes cluster is assigned a unique IP address. This isolation ensures that pods can communicate with each other transparently even between nodes and securely across a flat network space. Agents inside a node can also communicate freely with all pods on that node. Service Networking: Kubernetes Services are an abstraction which defines a logical set of Pods and a policy by which to access them. This abstraction allows for decoupling work definition from the pods and lets you expose an application running inside the pods to be reachable from outside the cluster or only for consumption inside it. Ingress: Managing external access to the services within a cluster, typically HTTP, Ingress allows for load balancing, SSL termination, and name-based virtual hosting.

Common Networking Solutions

Several networking solutions can implement the required Kubernetes model:

CNI (Container Network Interface) plugins: These are used in Kubernetes to connect pod networks to the underlying host network. Popular plugins include Cilium [5], Flannel [6], and Weave [7]. Kube-proxy: A network proxy which reflects services as defined in the Kubernetes API on each node. It can operate in various modes such as iptables or IPVS, managing routing and load-balancing for traffic heading to the services.

Network Policies

Network policies in Kubernetes allow you to control the traffic between pods. You can define rules that specify which pods can communicate with each other and which ports are open on those pods. This control is crucial for enforcing a secure multi-tenant environment and reducing the risk of internal threats.

Challenges and Best Practices

While Kubernetes networking provides numerous benefits, it also presents challenges such as network security, complexity in configuration, and managing high availability. Best practices include:

- Regularly updating network policies.
- Using namespace segregations to enhance security.
- Monitoring and logging network traffic for anomalies.

2.6 Custom Resources

Kubernetes offers powerful extensibility through custom resources (CRs), custom resource definitions (CRDs), and custom controllers. This chapter delves into how these components work together to expand Kubernetes' functionality, enabling the creation of domain-specific extensions that integrate seamlessly with the core Kubernetes API.

Custom Resource Definitions (CRDs)

CRDs are a pivotal feature in Kubernetes that allow users to define new types of resources. By creating a CRD, you define a new kind of resource with a name and schema that the Kubernetes API server can handle. This new resource behaves like standard Kubernetes objects, allowing you to create, view, and manage it via the Kubernetes API.

- Definition Process: The process involves specifying the resource's name, scope (namespaced or cluster-wide), and schema, which details the structure and validation rules of the API.
- Uses: CRDs are commonly used to create operational abstractions for complex systems, such as managing a database cluster or a network configuration.

Custom Resources

Custom resources are instances of CRDs. They allow developers to store and retrieve structured data. Unlike traditional Kubernetes objects such as Pods or Services, custom resources store configurations and states specific to user-defined applications.

- Manipulation: Users can manipulate these resources using standard Kubernetes tools like kubectl.
- Examples: Examples include configuring application-specific settings, maintaining records of deployed artifacts, or storing metadata that interacts with custom controllers.

Custom Controllers

Custom controllers are programs that watch the state of your Kubernetes resources and perform actions to drive the current state towards the desired state. They are key to implementing advanced functionality and handling the lifecycle of custom resources.

- Controller Pattern: Custom controllers use the observer pattern to watch for changes in CRs and execute defined business logic to adjust resources as needed.
- Functionality: They can manage anything from complex application workflows to simple tasks, like updating a status field in a CR.

Operators

Operators are custom controllers designed to manage specific applications or services. They encapsulate operational knowledge and can automate complex tasks.

- Operator SDK: The Operator SDK is a popular tool that simplifies the creation of custom controllers and CRDs, providing templates and management capabilities.
- Life-Cycle Management: Operators handle the full lifecycle of an application, from deployment to scaling and updates.

2.7 Security

Security in Kubernetes is paramount, especially as applications and their configurations become increasingly complex. This chapter focuses on three core components that play critical roles in Kubernetes security: Secrets, Service Accounts, and Role-Based Access Control (RBAC). Understanding and implementing these features correctly is essential for maintaining a secure Kubernetes environment.

Secrets

Secrets in Kubernetes are used to store and manage sensitive information such as passwords, OAuth tokens, and SSH keys. Keeping such information secure and out of application code is crucial for maintaining the integrity and security of applications. Secrets can be mounted as data volumes or exposed as environment variables to be used by pods in a cluster, thereby minimizing the risks associated with exposing sensitive data. It is recommended to encrypt secrets at rest and to limit access to them using appropriate RBAC policies.

Service Accounts

Service accounts are special user accounts that can be assigned to applications and services running on Kubernetes. They help in managing access to the Kubernetes API, allowing applications to interact with the Kubernetes master securely. They facilitate the automated, secure interaction between Kubernetes pods and the Kubernetes API. The best practice for security accounts is to grant only the minimal permissions necessary to perform the intended tasks.

Role-Based Access Control (RBAC)

RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In Kubernetes, RBAC allows administrators to regulate access to Kubernetes resources and namespaces based on the roles assigned to users and processes [8].

- Roles and RoleBindings: Roles define permissions on resources, and RoleBindings assign these roles to specific users, groups, or service accounts within a particular namespace.

- ClusterRoles and ClusterRoleBindings: For permissions that span across a whole cluster, ClusterRoles and ClusterRoleBindings are used. They are critical for managing cluster-wide permissions.

2.8 Workload Scaling

Kubernetes workload scaling is a fundamental aspect of managing containerized applications in modern cloud environments. As applications experience fluctuating demand, the ability to dynamically adjust resource allocation ensures optimal performance, cost efficiency, and reliability. Scaling in Kubernetes involves automatically increasing or decreasing the number of pod replicas based on current workload demands, a process managed by the Kubernetes control plane using various built-in mechanisms such as vertical and horizontal scaling.

2.8.1 Vertical and Horizontal Scaling

There are two primary types of scaling in Kubernetes: horizontal and vertical [9]. Horizontal scaling, also known as scaling out, involves adding more pod replicas to distribute the load evenly. This method is particularly effective for stateless applications where multiple instances can run independently. Vertical scaling, or scaling up, entails increasing the resource limits of existing pods, making each instance more powerful. This approach is often used for stateful applications that require higher computational power or memory.

Kubernetes offers several tools to facilitate workload scaling. The Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pod replicas based on CPU utilization or other select metrics. The Vertical Pod Autoscaler (VPA) recommends or automatically adjusts the resource requests and limits of containers within pods. Additionally, if available, the Cluster Autoscaler can adjust the number of nodes in a cluster, ensuring that there are enough resources to meet the demands of the scaled workloads.

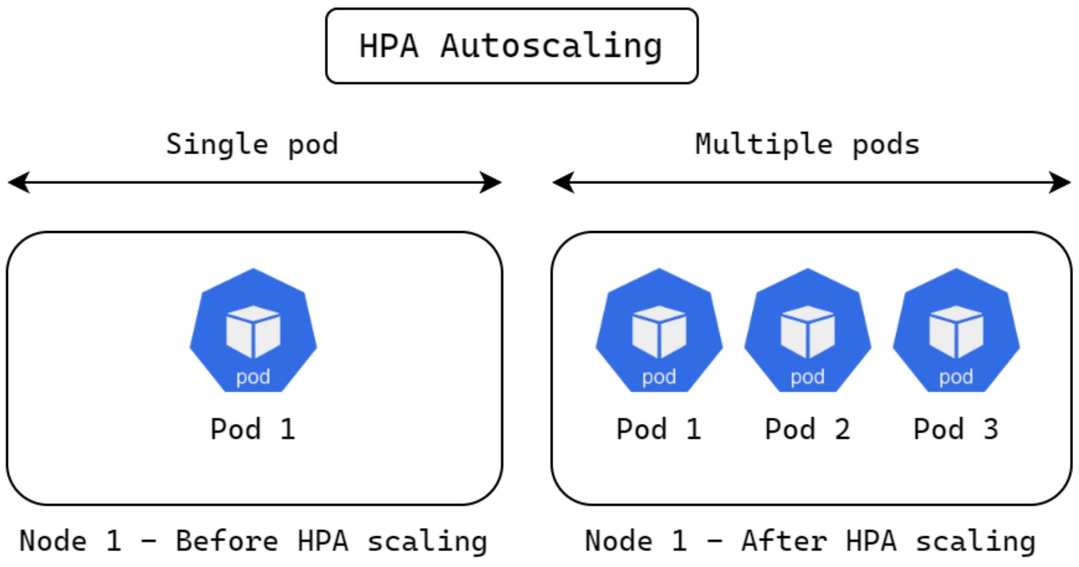


Figure 2.3: Horizontal Pod Autoscaling

Deployments and ReplicaSets are essential Kubernetes objects that play a pivotal role in the autoscaling of applications. A Deployment manages the desired state of an application by defining the number of replicas and their configuration, while the underlying ReplicaSet ensures that the specified number of pod replicas are maintained. This setup provides fault tolerance and load distribution. In the context of autoscaling, the HPA dynamically interacts with Deployments and ReplicaSets to adjust the number of replicas based on real-time performance metrics. By continuously monitoring these metrics, HPA can scale the number of pod replicas up or down to meet the current demand, ensuring optimal resource utilization.

Chapter 3

eBPF

The Extended Berkeley Packet Filter (eBPF) is a revolutionary technology born within the Linux kernel that enable the safe execution of sandboxed programs in a priviledged context [10]. It allows to safely extend the capabilities of the kernel without changing the source code or loading additional kernel modules. Traditionally, the operating system has always been slower to evolve due to its need to be reliable and secure. Due to its priviledged access to more sensitive information it is the best place to implement observability, security and network functionality. Thanks to eBPF however we can expand the capabilities of the operating system even in spite of its slow evolution, molding it to our needs.

3.1 History

At its inception in 1992, the Berkeley Packet Filter (BPF) was designed as a network tap and packet filter which permitted the capture of network packets at the operating system level with the goal of increasing the speed of such operations. It allowed for direct access to raw link-layer packets and a user level program could apply filters on which packets it wanted to recieve. The program needs to be written for an in-kernel virtual machine and its code is interpreted and compiled to machine code by a just-in-time (JIT) mechanism.

Over the years, BPF evolved and its applications expanded beyond packet filtering. With the introduction of extended BPF (eBPF) around 2014, it transitioned into a more general execution engine within the Linux kernel. eBPF enhanced the capabilities of BPF by supporting a broader set of functionalities like performance monitoring, network traffic management, and security enforcement, making it an integral part of modern Linux systems.

3.2 Architecture

In this chapter we are going to go over the core components and mechanics that make it possible for an eBPF program to function. We will explore elements such as the eBPF virtual machine (VM), the rigorous safety checks enforced by the verifier, the efficiency gains offered by just-in-time compilation, and the dynamic interaction between user space and kernel space facilitated by eBPF maps and helper functions.

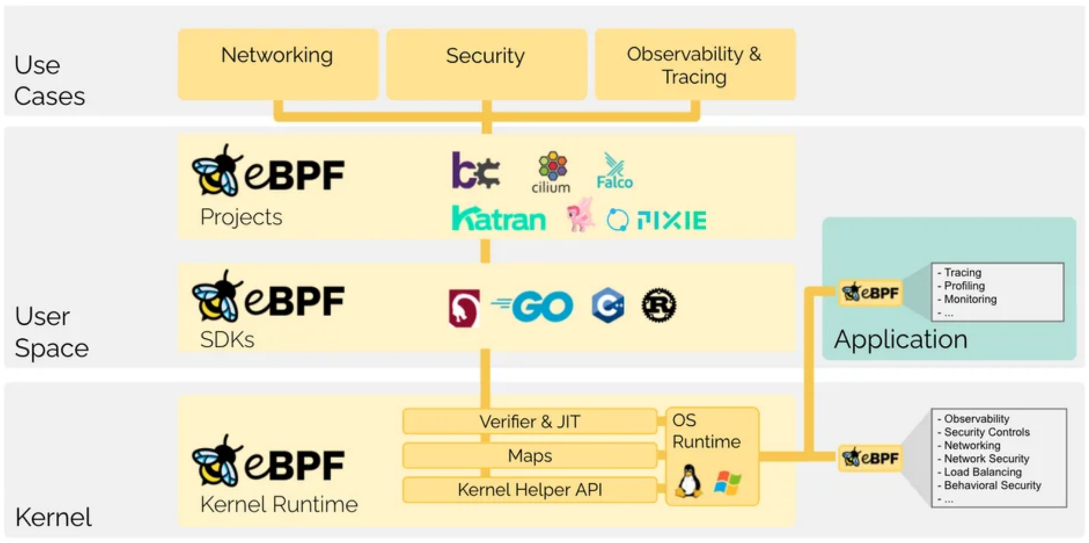


Figure 3.1: eBPF Architecture [10]

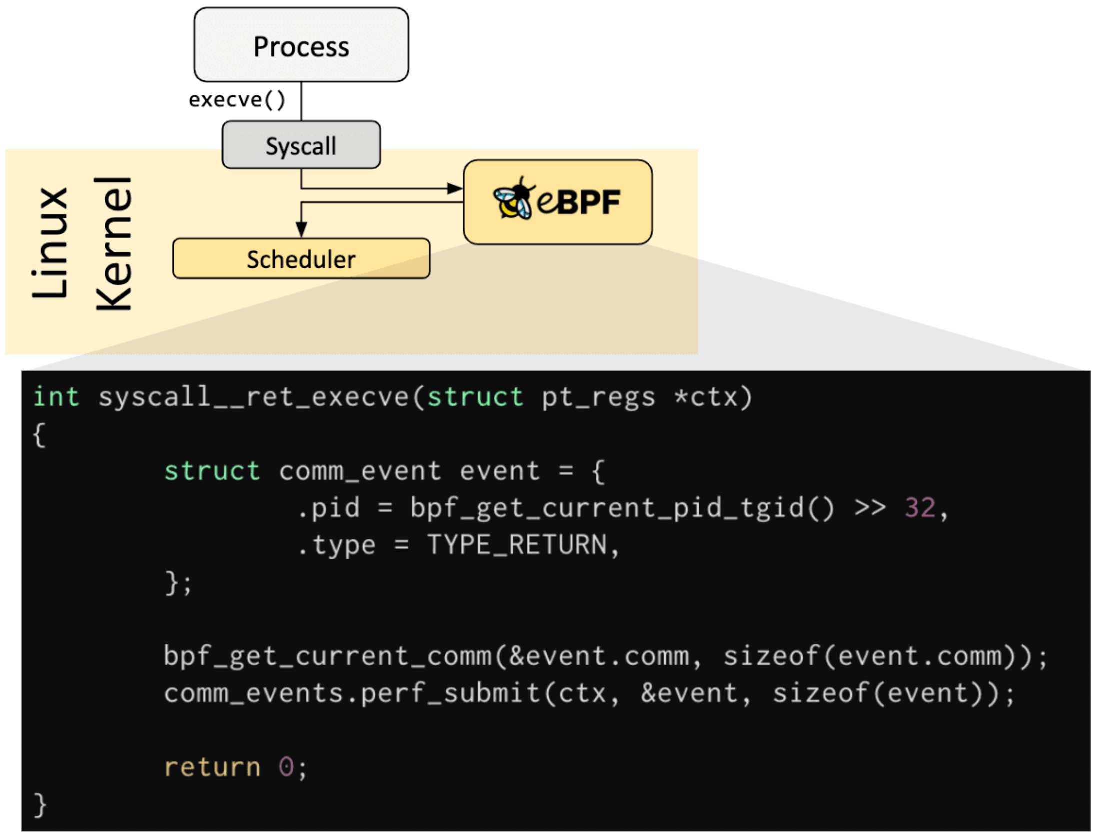
3.2.1 eBPF Program Exection

eBPF is a powerful technology that allows code execution within the Linux kernel with minimal performance overhead, enabling deep observability and security functionalities [11]. One of the key features of eBPF is its ability to attach to various hooks and probes within the kernel, such as system call hooks and kprobes, to monitor and respond to a wide range of events.

System call hooks are crucial for monitoring and modifying the behavior of system calls, which are the primary interface between user applications and the kernel. By attaching eBPF programs to system call hooks, administrators can intercept, log, and even alter the behavior of these calls. This capability is particularly useful for security monitoring, auditing, and performance analysis. For example, an eBPF program can be attached to the open system call to log every file access attempt, capturing details like the file name, user ID, and process ID. This real-time visibility into file operations can help detect unauthorized access and potential breaches.

Kernel probes (kprobes) provide a mechanism to dynamically insert probes into running kernel code, allowing eBPF programs to execute when specified kernel functions are called. Kretprobes, a variant of kprobes, are used to attach to the return point of a kernel function, enabling the capture of function return values. These probes are invaluable for tracing and debugging, as they allow developers to monitor the execution flow and internal state of the kernel.

Combining system call hooks and kprobes with eBPF programs offers a flexible and powerful approach to system monitoring and security. eBPF programs can be written in C, compiled into bytecode, and then loaded into the kernel using tools like bcc (BPF Compiler Collection) or libbpf. Once loaded, these programs run in a sandboxed environment, ensuring they do not compromise kernel stability or security.



```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Figure 3.2: eBPF Execution for predefined hooks [10]

3.2.2 eBPF Virtual Machine

The eBPF virtual machine is not a virtual machine in the traditional sense, like those used for emulating entire operating systems. Instead, it is a lightweight, sandboxed execution environment within the Linux kernel that runs eBPF programs. These programs are loaded into the kernel at runtime and execute in response to events such as network packets arriving or syscalls being made. Since it is running inside the kernel it has no performance overhead for syscalls and context switching between kernel and user space.

While all operations done by eBPF programs can be achieved by normal kernel modules it is extremely dangerous to directly execute in-kernel programs. This could potentially cause lock-ups, memory corruption, crash processes, cause security vulnerabilities and other unwanted effects. Thus, running fast JIT compiled kernel code using a safe environment such as a VM is a much better option in terms of security and kernel safety.

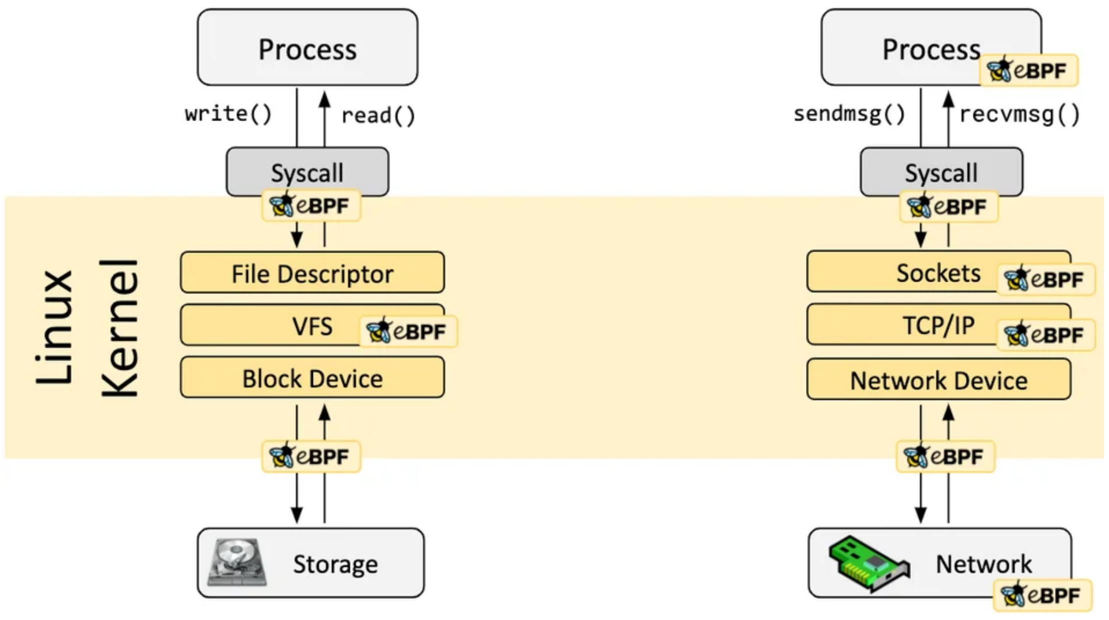


Figure 3.3: eBPF Execution for probes [10]

The eBPF VM however has some limitations:

- It is not Turing complete. No loops are allowed so all eBPF programs are guaranteed to always complete and never hang, however there is an exception. Starting with Linux kernel version 5.3 bounded loops are now possible within eBPF programs. With this change the previous limit of eBPF programs having only 4096 instructions has been increased to one million.

- All memory access is bounded and type-checked and only a MOV assembly instruction can change the type of a register.
- There are no null dereferences and the main function takes only a single argument.

Thanks to these kind of restrictions the eBPF program can be quickly verified and loaded into the kernel after its instructions are parsed into a directed acyclic graph (DAG).

3.2.3 Verifier

The eBPF verifier plays a crucial role in the security and stability of the Linux kernel by ensuring that all eBPF programs loaded into the kernel are safe and reliable. This chapter delves into the intricate workings of the eBPF verifier, its importance in the eBPF ecosystem, and the mechanisms it employs to maintain system integrity. The primary function of the eBPF verifier is to analyze and certify the safety of eBPF programs before they are allowed to run within the kernel. This process is essential because it prevents potentially harmful code, whether malicious or inadvertently dangerous, from executing operations that could destabilize or compromise the kernel.

How it works

The verification process begins with static code analysis, a thorough examination that inspects the eBPF bytecode. This scrutiny involves a series of checks designed to affirm that the program adheres to strict operational guidelines. These checks include validating memory accesses to ensure they are within safe bounds, confirming that the control flow of the program does not contain any loops that might lead to indefinite execution, and verifying that all operations are performed on initialized and legitimate data. Additionally, the verifier ensures that each program has a clear and safe path to termination, preventing it from hanging the kernel. Despite its critical role, the verifier faces significant challenges, primarily due to the complexity and power of eBPF programs. Analyzing these dynamic programs requires a balance between thoroughness and efficiency to maintain performance standards. Developers must also navigate the intricacies of the verifier when writing or debugging eBPF code, as the detailed analysis can sometimes reject valid code due to overly stringent checks or edge cases that are difficult to anticipate.

Ongoing developments in the verifier aim to expand its capabilities and improve its efficiency, accommodating the increasing sophistication of eBPF programs. Ultimately the eBPF verifier is more than just a gatekeeper; it is a sophisticated tool that plays a foundational role in the security and stability of the Linux kernel.

As eBPF continues to evolve, so too will the verifier, adapting to new challenges and ensuring that eBPF remains a safe and powerful tool for kernel-level programming.

3.2.4 JIT Compiler

One of the key components that enhance the performance of the eBPF framework within the Linux kernel is the Just-In-Time (JIT) compiler. This chapter delves into the JIT compiler's role in the eBPF architecture, explaining how it works, its benefits, and the technical challenges it addresses.

The eBPF JIT compiler is crucial for translating eBPF bytecode into native machine code executable directly by the host CPU. This translation occurs at runtime and is designed to optimize the performance of eBPF programs, making them as efficient as possible. By converting bytecode to optimized machine code, the JIT compiler reduces the execution overhead that would typically be associated with interpreted or non-native code. The process begins when an eBPF program, verified for safety and correctness by the eBPF verifier, is passed to the JIT compiler. Here, the compiler takes the generic eBPF bytecode and translates it into architecture-specific instructions. This task requires a deep integration with both the eBPF system and the underlying hardware architecture, as the generated code must be highly optimized and tailored to the specific capabilities and features of the processor.

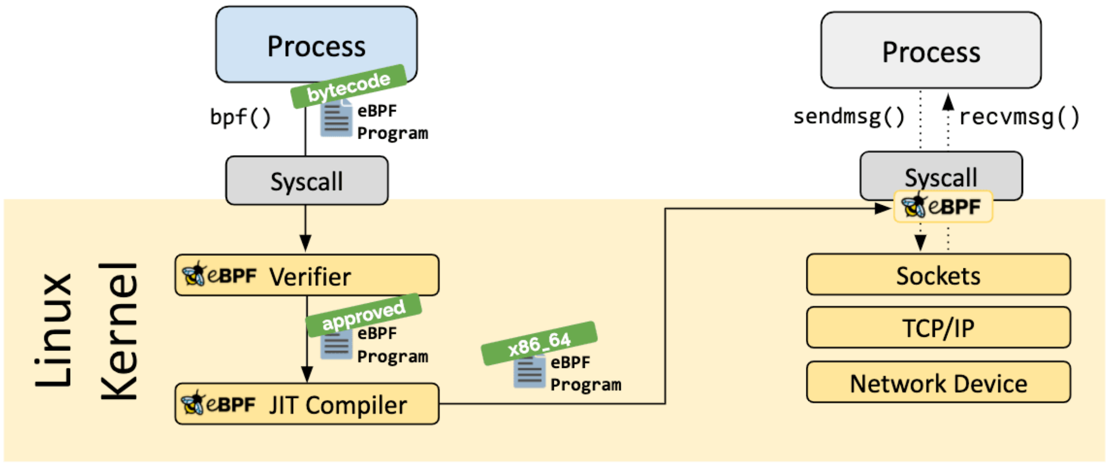


Figure 3.4: Verifier and JIT Compiler chain [10]

Performance

The primary advantage of JIT compilation in eBPF is the significant boost in performance. Native machine code executes much faster than interpreted bytecode because it is directly processed by the CPU without the need for additional translation. Moreover, JIT-compiled eBPF programs can leverage CPU-specific optimizations, further enhancing their efficiency and speed. This capability is particularly important for time-sensitive tasks such as network packet filtering and system monitoring.

Challenges

Implementing a JIT compiler for eBPF is not without challenges. The compiler must ensure that the translated code is not only fast but also safe and secure. It must prevent any potential security vulnerabilities that could arise from lower-level machine code operations. Furthermore, maintaining portability across different hardware architectures complicates the JIT compiler's development, requiring careful design to accommodate diverse CPU features and instruction sets.

Additionally, the JIT compilation process itself must be fast enough not to negate the benefits of increased execution speed. This requires a balance between the time spent compiling the code and the runtime performance gains.

3.2.5 eBPF Maps

eBPF maps are a critical component of the eBPF architecture, providing the essential functionality of maintaining state across function calls and between different eBPF programs. This chapter explores the role of eBPF maps, the various types available, their functionalities, and the mechanics of their integration within the Linux kernel.

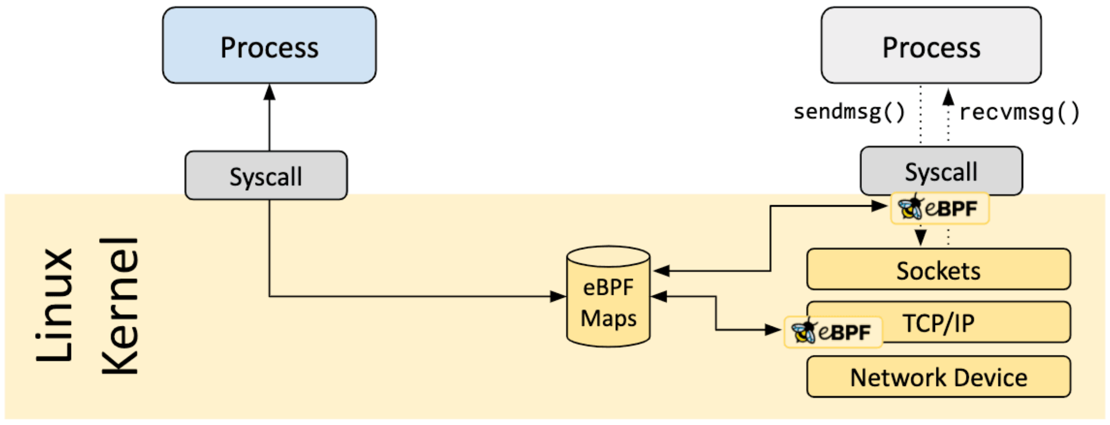


Figure 3.5: eBPF Maps [10]

What are eBPF Maps

eBPF maps are data structures that store data in key-value pairs and are accessible both from user space and from within eBPF programs running in kernel space. They act as the primary mechanism for eBPF programs to retain state information between function executions, share data between different programs, or communicate with user-space applications. This capability significantly extends the usefulness of eBPF programs beyond stateless processing.

Types of eBPF Maps

The flexibility of eBPF maps derives primarily from the variety of map types designed to support different use cases such as:

- **Hash Maps**: Store entries in a key-hash structure, allowing for efficient lookups, inserts, and deletions.
- **Array Maps**: Use a simple array structure where each element is accessed by a direct index, providing very fast access times.
- **Per-CPU Maps**: Designed for scalability, these maps reduce contention by maintaining separate data instances for each CPU.
- **LRU Maps**: Automatically evict least recently used items when the map reaches its capacity, useful for cache management.
- **Queue and Stack Maps**: Implement First-in-First-out (FIFO) and Last-in-First-out (LIFO) data structures, respectively, which are useful for various network and processing tasks.

How eBPF Maps Work

eBPF maps are created via API calls from user space and can be manipulated by eBPF programs at runtime. When an eBPF program is loaded into the kernel, it can access predefined map handles that refer to these data structures. This design allows eBPF programs to interact seamlessly with preallocated storage space without needing to perform any memory management tasks, which are handled by the kernel.

While eBPF maps provide powerful functionalities, they also introduce challenges such as managing synchronization between multiple programs and handling map lifecycle correctly without leaking resources. Developers must be aware of these issues and utilize available tools and practices to address them.

3.2.6 Helper Functions

eBPF helper functions are a critical aspect of the eBPF ecosystem, enabling eBPF programs to interact with the Linux kernel in powerful ways. These functions provide eBPF programs access to kernel services and data structures, enhancing their capabilities beyond mere packet filtering or data observation. This chapter explores the different types of helper functions as well as guidelines for their effective utilization.

Types

eBPF helper functions can be categorized based on the functionalities they provide. Some common categories include:

- **Map Management Functions**: Functions that facilitate operations on eBPF maps such as retrieving data from a map or adding or modifying map entries.
- **Packet Operations**: Functions that manipulate packet data, crucial for network monitoring and modification tasks.
- **System Information Retrieval**: Functions that fetch system data like the current time or system configuration details, enabling eBPF programs to make context-aware decisions.
- **Security and Auditing Functions**: Functions that help implement security policies or log events, essential for building security-focused eBPF applications.

How they work

eBPF helper functions are invoked within eBPF programs using a specific helper call instruction. The eBPF program passes parameters to the helper function, which then executes a predefined operation and returns a result. This mechanism is designed to be safe and efficient, with strict type and access checks performed by the eBPF verifier to ensure that all helper function calls do not compromise system stability or security.

Best Practices

When utilizing eBPF helper functions, it's essential to employ them judiciously, particularly in performance-sensitive areas, as excessive use can lead to overhead. Developers should be mindful of the security implications of each function, especially when handling user-supplied data, to prevent potential vulnerabilities. Additionally, compatibility is a key consideration; not all helper functions are supported across different kernel versions, so it's important to verify their availability and consider alternative strategies if necessary. By adhering to these best practices, developers can ensure that their use of eBPF helper functions is both effective and secure.

Despite their utility, eBPF helper functions come with limitations. The kernel strictly controls the availability of these functions to prevent abuse or unintended operations that could destabilize the system. Additionally, the evolving nature of the eBPF landscape means that available helper functions can change between kernel versions, posing challenges for developers in maintaining forward compatibility.

3.2.7 User Space Tools

The development and management of eBPF programs rely heavily on robust user space tools that provide capabilities ranging from program loading and manipulation to performance analysis and debugging. This chapter explores the landscape of user space tools designed for eBPF, detailing how they enhance the functionality and usability of eBPF within Linux systems.

User space tools for eBPF are critical for bridging the gap between complex kernel-level operations and user-friendly interfaces. These tools enable developers to write, deploy, debug, and manage eBPF programs efficiently. The toolset ranges from compilers that transform high-level code into eBPF bytecode to loaders that insert the bytecode into the kernel, and libraries that facilitate interaction between user applications and eBPF programs running in the kernel.

Notable Tools

The eBPF ecosystem is supported by a variety of tools developed by the community and supported by various organizations. Key among these is the **BPF Compiler Collection (BCC)**, which provides a set of tools and libraries for creating eBPF programs in C and includes utilities to attach these programs to hooks in the kernel. BCC simplifies the process of developing eBPF programs by abstracting much of the complexity involved in direct eBPF API calls.

Another significant tool is **bpftrace**, designed for tracing and dynamically analyzing Linux kernel behavior. It allows for quick and easy scripting of eBPF code using a high-level language, which is particularly useful for diagnostics and troubleshooting. This tool supports a variety of powerful capabilities, including custom probe insertion and data aggregation.

These user space tools not only make eBPF more accessible but also enhance its power. They provide essential capabilities such as performance monitoring, debugging, and automated management of eBPF programs. For example, tools like BCC and bpftrace come equipped with numerous pre-built scripts that can be used to analyze system performance, track down performance bottlenecks, and monitor system events without needing to write complex eBPF code from scratch.

Libraries

Within the ecosystem of user space tools for eBPF, libraries play a pivotal role in simplifying the development and operational processes associated with eBPF programs. Libraries such as **libbpf**, part of the Linux kernel project, provide essential abstractions and functionalities that streamline the interaction between eBPF programs and user applications. Libbpf, in particular, facilitates the loading and management of eBPF programs and maps, offering a straightforward API that helps developers avoid the intricacies of lower-level eBPF system calls. This enables more robust and less error-prone applications, as developers can focus on program logic rather than kernel-specific details. As eBPF continues to mature, these libraries are regularly updated to support the latest features, ensuring that developers have the tools needed to leverage eBPF's full capabilities effectively.

Chapter 4

Monitoring Tools

In the dynamic landscape of modern software operations, monitoring tools play an indispensable role in ensuring applications run smoothly and efficiently. This chapter focuses on a suite of essential monitoring tools such as Grafana, Loki, Prometheus, and Promtail, each designed to address specific facets of monitoring and data visualization. These tools work in concert to provide developers and system administrators with deep insights into their systems' health, performance, and logs. We will explore how Grafana allows for powerful data visualization across various data sources, how Prometheus efficiently collects and stores metrics, how Loki enhances log aggregation and analysis, and how Promtail facilitates the gathering of logs for processing by Loki. Together, these tools form a robust monitoring stack that empowers teams to detect anomalies, understand trends, and make data-driven decisions to optimize application performance and reliability.

4.1 Grafana

Grafana was initially developed in 2014 by Torkel Ödegaard as an open-source project aimed at providing a powerful, flexible platform for visualizing time-series data [12]. Originally conceived as a front-end for Graphite, it quickly expanded to support a wide range of data sources, including Prometheus, InfluxDB, and Elasticsearch, among others. Grafana's intuitive interface and robust visualization capabilities attracted a growing community of users and contributors, leading to rapid enhancements and feature additions. Over the years, Grafana Labs was established to drive the development and commercialization of Grafana, while still maintaining its open-source roots. Today, Grafana stands as one of the most popular and versatile data visualization tools, widely used across various industries for monitoring and analytics.

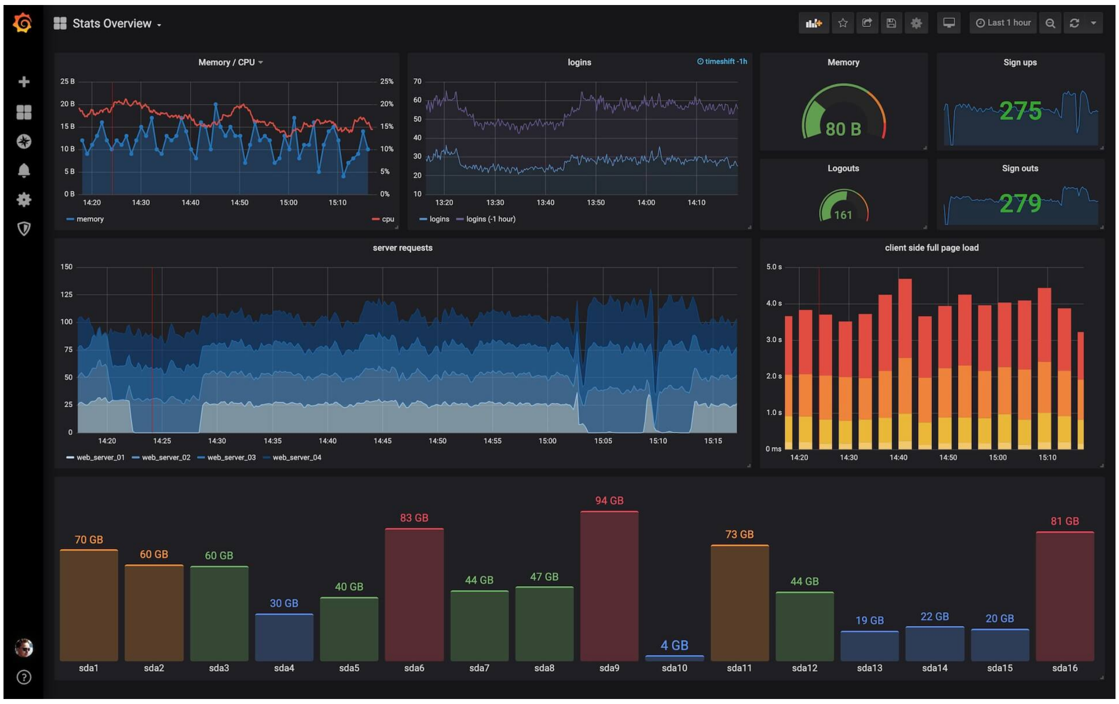


Figure 4.1: Example of a Grafana dashboard

It is an open-source analytics and interactive visualization web application that provides rich ways to chart and monitor data in real time. Widely recognized for its versatility and user-friendly interface, Grafana is the tool of choice for data scientists, IT administrators, and DevOps engineers seeking to understand complex datasets. This chapter will introduce Grafana, outline how it operates, and delve into its broad capabilities, which allow users to turn their time-series data into beautiful graphs and visualizations that highlight trends, spikes, and drops.

Grafana stands out in the realm of data visualization and monitoring for its extensive capabilities, catering to a diverse range of needs and preferences across industries. Its versatility is highlighted by its support for a wide array of data sources such as Prometheus, Elasticsearch, MySQL, and more, allowing users to pull in data from various environments and view it through a singular, cohesive interface. Grafana's strength lies in its robust visualization options which include line charts, bar graphs, histograms, and geospatial maps, among others, each customizable with a range of display options to suit specific user requirements. Users can enhance dashboards with interactive features like variable selectors for real-time analysis and comparative studies across different metrics or time periods.

The platform's alerting functionality is highly advanced, capable of sending notifications through multiple channels based on triggers set within the data, which is crucial for maintaining operational continuity and quick responsiveness to potential issues. Annotations in Grafana allow users to mark specific events directly on graphs, providing context for spikes or drops in data, which is especially useful during post-mortem analysis or incident reviews. Additionally, the integration of plugins extends Grafana's functionality, enabling the inclusion of new data sources, new monitoring options, or even custom applications tailored to the user's operational landscape. This holistic approach to data interaction not only simplifies the monitoring tasks but also empowers teams to derive actionable insights effectively, making Grafana an invaluable tool in data-driven decision-making processes.

- **Data Sources**: Grafana's strength lies in its extensive support for various data sources. Users can connect to multiple sources simultaneously, allowing complex data overlays and correlation.
- **Dashboards and Panels**: Users design dashboards that contain panels; each panel can show data from different sources through graphs, charts, and tables. These dashboards are highly interactive and can be customized to suit the specific monitoring needs of any team.
- **Alerting System**: Grafana includes a powerful alerting system that notifies teams of potential issues before they become critical. This system allows users to define alert rules for the metrics they are visualizing, ensuring rapid response times to performance anomalies or system failures.

4.2 Grafana Loki

Similarly to how Grafana is an open-source dashboard platform, there are other OSS tools made by their team that are tightly integrated in the Grafana ecosystem. One of these tools is Loki, a project developed by Grafana Labs, distinctly different from Grafana itself, designed specifically for aggregating and querying logs from various sources [13]. While Grafana focuses on metrics visualization and analysis, Loki provides a solution to manage and analyze log data, complementing Grafana's capabilities by integrating seamlessly with it for comprehensive observability.

Grafana is a broad platform used for data visualization across various metrics, allowing users to create dynamic dashboards to track performance, resources, and more. In contrast, Loki is specialized; its primary function is to handle logs, which are textual records of events occurring within systems, making it a vital tool in debugging and tracing system behavior.

Loki is designed to be cost-effective and highly efficient, offering a simpler approach to log processing that differs significantly from other log-aggregation products like Elasticsearch. Instead of indexing the content of the logs, as is common with many other logging systems, Loki indexes only the metadata of the logs. This approach results in a more lightweight system that requires less storage and computational power, reducing cost and increasing the speed of queries.

Architecture

Loki's architecture draws significant inspiration from Prometheus, particularly appealing to those already familiar with the Prometheus query language, which streamlines the learning curve. Logs are collected through various agents such as Promtail, Grafana Agent, or Fluentd, which are responsible for gathering logs from different parts of a system and forwarding them to Loki. This approach not only simplifies the ingestion process but also optimizes the management of logs by categorizing them efficiently.

In Loki, logs are meticulously structured into streams that are tagged with a set of key-value pairs, a method influenced by the labeling system used in Prometheus. This label-based approach facilitates structured querying of logs without the need for a full-text search index, significantly reducing the overhead associated with log data management. Each stream represents a specific source or type of log, allowing users to retrieve and analyze log data based on these labels, which is particularly useful in environments with large volumes of log data.

The storage model of Loki is designed for efficiency and scalability. Logs are segmented into chunks, which are then compressed to minimize storage space and enhance the speed of data retrieval. When a query is executed, Loki identifies and fetches the relevant compressed chunks, decompresses them, and performs a linear scan to locate the requested log entries. This process is remarkably efficient because the initial filtering by labels drastically reduces the number of logs to be scanned, ensuring quick retrieval times even over large datasets.

One of Loki's standout features is its seamless integration with Grafana, which provides a unified platform for monitoring and visual analysis. Once logs are ingested, they can be queried using LogQL, a powerful and flexible query language developed by Grafana Labs. LogQL allows for intricate querying and manipulation of log data, enabling users to perform detailed analysis and aggregation directly within their Grafana dashboards. This integration allows for the correlation of log data with metrics, offering a comprehensive view of system health and behavior. The ability to visualize logs alongside metrics in Grafana not only enhances the overall observability but also significantly improves the capabilities for debugging and diagnosing issues across systems.

4.3 Promtail

Promtail was developed by Grafana Labs as a complementary component to Loki, its log aggregation system, to address the need for efficient log collection and forwarding [14]. The project began shortly after the introduction of Loki in late 2018, aiming to provide a seamless way to ship logs from various sources to Loki. Promtail was designed to work similarly to Prometheus's service discovery mechanisms, allowing it to dynamically locate and collect logs from a variety of sources, including Kubernetes pods, systemd journals, and standalone log files. As an open-source project, Promtail has rapidly evolved, integrating feedback from the community and incorporating features to better handle diverse logging environments. Its development has been guided by the same principles that drive Grafana and Loki: simplicity, efficiency, and ease of use. Today, Promtail is a critical tool in the Grafana Labs ecosystem, providing a robust solution for log collection and forwarding in modern cloud-native environments.

Listing 4.1: Example of a Promtail configuration file

```
server:
    http_listen_port: 9080
    grpc_listen_port: 0
positions:
    filename: /var/lib/promtail/positions.yaml
clients:
    - url: http://localhost:3100/loki/api/v1/push
scrape_configs:
    - job_name: system
      static_configs:
        - targets:
            - localhost
          labels:
            job: varlogs
            __path__: /var/log/*log

    - job_name: nginx
      static_configs:
        - targets:
            - localhost
          labels:
            job: nginx
            __path__: /var/log/nginx/*.log
```

How it works

Promtail is an agent that is deployed directly on the servers where logs are generated. Its primary function is to collect logs, enrich them with metadata, and forward them to Loki for storage and analysis. It is crucial for ensuring that logs are not only collected in real-time but are also structured in a way that makes them easily queryable within Loki. By tailing log files, monitoring them for new entries and capturing these entries as they occur, Promtail acts as the critical first step in the log processing pipeline.

Promtail's operation can be divided into several key activities, each integral to its role as a log collector:

- **Service Discovery**: Leveraging techniques similar to those used in Prometheus, Promtail employs service discovery to automatically detect and monitor log file paths across hosts in the environment. This feature ensures that all relevant logs are captured without manual intervention.
- **Log Tailing**: Once log paths are identified, Promtail begins tailing these files, which involves reading new log entries as they are appended to the logs.
- **Metadata Enrichment**: As logs are collected, Promtail attaches labels to each log entry. These labels, which can include data such as the source file, host name, and other contextual information, are crucial for organizing logs in Loki and facilitate efficient querying and analysis.

Configuration

Configuring Promtail is a crucial step that determines how effectively it can collect and process logs. This configuration is managed through a detailed configuration file, where administrators specify various parameters that guide Promtail's behavior. Within this file, users define jobs, which are essentially instructions on where Promtail should look for logs and how it should handle them. Each job contains specifications about the log paths, the frequency of log scraping, and the methods for log extraction. These jobs allow Promtail to systematically manage different sources of logs, ensuring that all necessary data is captured.

Moreover, the configuration file outlines how logs should be processed through a series of pipeline stages. As logs are collected, they undergo various transformations: initially, they are parsed, which involves breaking down the raw log data into a structured format that Loki can more easily query. Following parsing, logs are labeled with metadata such as the source file's path or the host's name, which aids in their categorization and retrieval in Loki. Lastly, a filtering process is applied to ensure that only relevant logs are forwarded to Loki, optimizing resource use and processing time.

These configuration steps are vital not just for operational efficiency but also for tailoring Promtail's functionality to the specific needs of an environment. By adjusting the configuration file, administrators can fine-tune Promtail's performance, from modifying scrape intervals to enhance real-time data collection, to tweaking parsing rules to better structure the data, or refining labels for more effective data segmentation. This flexibility allows Promtail to be a powerful and adaptable tool in any logging architecture.

Advanced Features

Promtail's advanced features extend its capabilities beyond basic log collection and processing, making it a versatile tool suited for complex and scalable logging architectures. One of the standout features of Promtail is its support for multi-tenancy. This capability is crucial for organizations that manage log data across multiple divisions or for service providers who handle logs from various customers. Through multi-tenancy, Promtail can segregate logs based on predefined rules, ensuring that data from different sources remains separate and secure, while still being processed through a single Promtail instance. This segregation is typically managed through labeling, where logs are tagged with tenant-specific identifiers that facilitate appropriate routing and storage in Loki.

Another advanced feature is Promtail's dynamic file watching. This allows Promtail to adapt to changes in log directories by automatically detecting new files or changes within existing files without needing manual reconfiguration. This feature is particularly useful in environments where log outputs are volatile and can change frequently, such as in temporary job logs or rapidly evolving application logs.

Promtail also supports various backoff strategies and load balancing, which enhance its robustness in high-throughput environments. These features help manage the load on both the Promtail agent and the Loki server, ensuring stable performance even under intense data ingestion scenarios. The backoff strategies prevent Promtail from overwhelming the network and the Loki server with too many requests simultaneously, especially during periods of high log generation.

These advanced features make Promtail not just a log collection tool but a comprehensive log management solution that can scale to meet the needs of large-scale, diverse, and dynamic logging environments. This adaptability ensures that as an organization's logging requirements grow and change, Promtail remains an effective and integral part of their observability infrastructure.

4.4 Prometheus

Prometheus was created in 2012 by former Google engineers at SoundCloud [15], inspired by their experiences with Google's internal monitoring tools [16]. The goal was to develop a robust, scalable solution tailored to the needs of modern cloud-native environments. Released as an open-source project under the Apache

2.0 license, Prometheus quickly gained traction due to its powerful query language, PromQL, and its emphasis on multidimensional data collection and monitoring. Its architecture, featuring a pull-based model and time-series database, set it apart from existing solutions. In 2016, Prometheus joined the Cloud Native Computing Foundation (CNCF) as the second hosted project after Kubernetes, which further accelerated its adoption and development. Today, Prometheus is a cornerstone of cloud-native monitoring, widely used by organizations to gain insights into their applications and infrastructure.

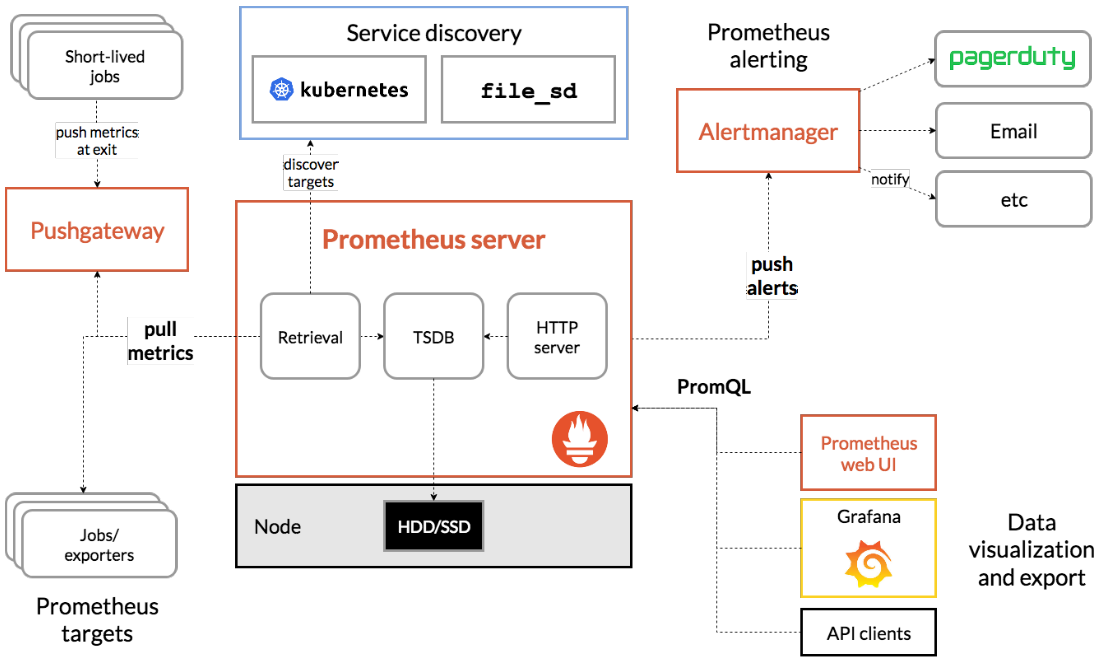


Figure 4.2: Prometheus architecture [16]

Architecture

At its core, Prometheus consists of several components that work together to gather, store, and use time-series data effectively:

- **Prometheus Server**: The heart of the system, the Prometheus server handles the retrieval and storage of data. It scrapes metrics from configured targets at specified intervals, evaluates rule expressions, displays results, and triggers alerts if certain conditions are met.
- **Storage**: Prometheus stores time-series data in a local disk in an efficient custom format, making the retrieval and real-time monitoring feasible even under high load.

- **Alertmanager**: This component manages alerts sent by the Prometheus server and takes care of deduplicating, grouping, and routing them to the correct receiver such as email, PagerDuty, or OpsGenie. It also ensures alert silencing and inhibition logic, which are crucial for managing a large number of alerts.
- **Push Gateway**: For supporting short-lived jobs, Prometheus includes a Push Gateway, which allows ephemeral and batch jobs to expose their metrics to Prometheus.

Prometheus excels in its core functionalities, centered around the efficient collection, storage, and processing of time-series data. At its heart lies a powerful data model that uses metric names coupled with sets of key-value pairs, known as labels, to uniquely identify each time series. This design allows for flexible and rich data queries through Prometheus's own query language, PromQL. PromQL can perform complex data retrieval and computation, making it possible to extract meaningful insights and trends from real-time data. Additionally, while Prometheus itself doesn't provide extensive visualization capabilities, it seamlessly integrates with tools like Grafana to enable sophisticated data visualization. This combination supports a wide array of monitoring scenarios, from tracking system performance metrics such as CPU and memory usage to understanding more granular application-specific metrics.

Usage

Prometheus is extensively used across various domains to address multiple monitoring needs:

- **Infrastructure Monitoring**: Prometheus can monitor a wide range of system metrics including memory, CPU, disk usage, and network statistics, providing a detailed view of the system health.
- **Application Monitoring**: By exposing custom metrics from applications, Prometheus can track application performance and usage, crucial for detecting anomalies and ensuring optimal performance.
- **Dynamic Service Discovery**: As services are dynamically added or removed in a modern cloud setup, Prometheus automatically discovers new services and starts collecting metrics without manual intervention.

Challenges

While Prometheus offers powerful monitoring capabilities, it faces several challenges, particularly in managing data storage and ensuring high availability. As Prometheus typically stores substantial volumes of time-series data locally, efficiently managing disk space and implementing effective data retention policies are crucial to prevent storage overload and maintain performance. Furthermore, Prometheus does not natively support clustering, which complicates efforts to achieve high availability. Users often have to rely on additional tools like Thanos or Cortex to create a more resilient monitoring infrastructure. These challenges require careful planning and configuration to ensure that Prometheus remains scalable and reliable in larger, more dynamic environments.

Chapter 5

Tetragon

Tetragon is an advanced observability and security platform designed for monitoring and managing modern cloud-native environments, particularly those orchestrated by Kubernetes [17]. It leverages eBPF technology to provide deep visibility into network traffic, application behavior, and system performance. Tetragon's capabilities are particularly valuable in complex and dynamic environments like 5G networks, where rapid changes and high data throughput necessitate robust monitoring solutions.

The main focus of Tetragon is detecting and reacting to security-significant events. Being able to monitor its host system is extremely valuable when the main objective is having a broad overview of the machine and what is happening inside it. The main type of events Tetragon can detect are:

- Process execution events
- System call activity
- I/O activity like network and file access

Another important feature of this eBPF framework is the ability to be Kubernetes-aware. Since it can be installed inside a K8s cluster it communicates with its API and can obtain useful information such as namespaces, pods, services and so on. By doing this the metrics that are generates in relation to security events are rich with metadata to help better understand what is happening inside the system we are trying to monitor.

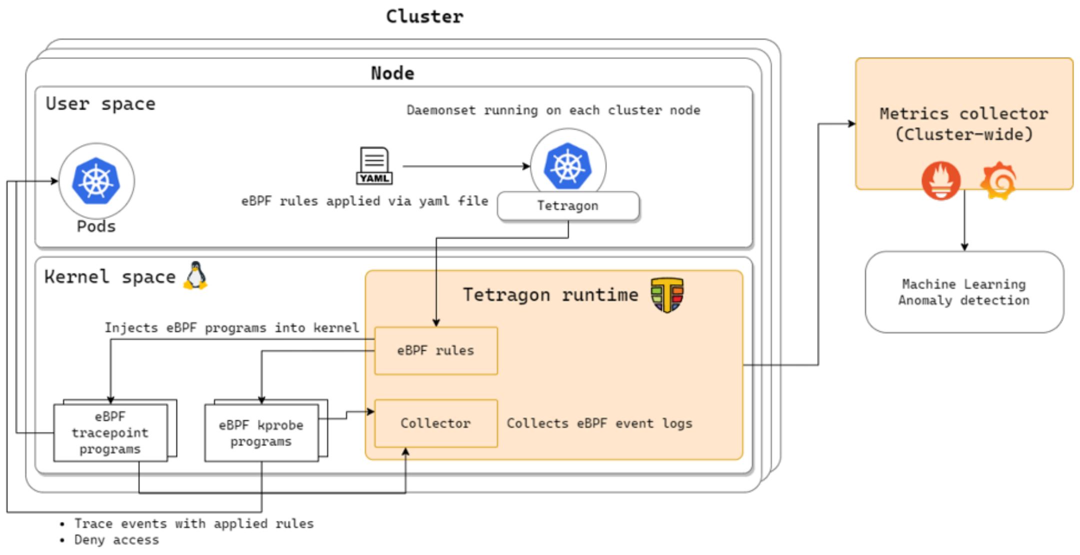

Cluster
Node
User space
Daemonset running on each cluster node
Pods
YAML
eBPF rules applied via yaml file
Tetragon
Kernel space
Tetragon runtime
Injects eBPF programs into kernel
eBPF rules
eBPF tracepoint programs
eBPF kprobe programs
Collector
Collects eBPF event logs
• Trace events with applied rules
• Deny access
Metrics collector (Cluster-wide)
Machine Learning Anomaly detection

Figure 5.1: Schema of Tetragon's architecture

History

Tetragon began as an ambitious project aimed at addressing the growing need for comprehensive observability and security in cloud-native environments. Initially conceived by a group of open-source enthusiasts and security experts, the project sought to leverage the emerging eBPF technology to provide deep visibility into system operations with minimal performance overhead. In its early stages, Tetragon focused on developing a robust framework for monitoring and analyzing kernel-level events, which required extensive collaboration with the Linux kernel community. The project's early development involved creating efficient eBPF programs, integrating them with user-space components, and ensuring seamless compatibility with Kubernetes. As Tetragon matured, it attracted contributions from a broader community, incorporating advanced features such as real-time policy enforcement, detailed logging, and integration with popular monitoring tools like Prometheus. This collaborative and iterative development process has positioned Tetragon as a powerful and flexible tool for enhancing security and observability in modern IT infrastructures.

Real Time Monitoring

Tetragon's real-time eBPF monitoring leverages the power of eBPF technology to provide deep, immediate visibility into system operations and network activities. By attaching eBPF programs to various kernel-level events, Tetragon can capture granular telemetry data on system calls, network packets, and resource usage with minimal performance overhead. This real-time monitoring capability enables administrators to detect and respond to anomalies and performance issues as they occur, ensuring that applications and infrastructure maintain optimal performance and security. The continuous stream of detailed insights provided by eBPF allows for proactive management, quick troubleshooting, and enhanced operational efficiency in dynamic, cloud-native environments like Kubernetes-orchestrated 5G networks. Moreover, Tetragon has the ability to filter security events either by file, socket, binary, ecc. but also by K8s objects such as namespaces.

Flexibility

Tetragon exemplifies flexibility through its support for user-defined tracing policies, empowering administrators to tailor monitoring and observability to their specific needs. These policies enable users to define what events and metrics to trace, allowing for a customized and granular approach to data collection. By writing tracing policies, users can specify conditions and contexts under which eBPF programs should be attached to kernel events, ensuring that only relevant data is captured and analyzed. This level of customization is particularly advantageous in diverse environments where different applications and services may have unique monitoring requirements. For instance, in a large 5G virtualized network managed by Kubernetes, various network slices, virtual network functions (VNFs), and containerized network functions (CNFs) can be monitored according to their specific performance and security needs. User-defined tracing policies allow for the dynamic adjustment of monitoring parameters as the network evolves, providing the flexibility to scale and adapt without compromising on observability. Moreover, these policies can be updated in real-time, enabling rapid response to emerging issues or changes in the network environment. This flexibility ensures that Tetragon remains a versatile tool, capable of providing deep insights and comprehensive monitoring tailored to the unique characteristics of any cloud-native infrastructure.

eBPF Kernel Aware

Tetragon, utilizing eBPF technology, gains comprehensive access to the Linux kernel state, enabling it to integrate this kernel-level data with Kubernetes context and user-defined policies. This integration allows for the creation of real-time rules enforced by the kernel, facilitating the annotation and enforcement of various system components such as process namespaces, capabilities, and socket associations. For instance, Tetragon can map process file descriptors to filenames, providing detailed visibility and control over system interactions. A practical application of this capability is the enforcement of security policies: when an application attempts to change its privileges, a predefined policy can trigger an alert or terminate the process before the syscall completes, preventing potential execution of unauthorized actions. This proactive approach to monitoring and enforcement significantly enhances the security and reliability of cloud-native environments.

5.1 Tracing Policies

Tracing policies are a core feature of Tetragon, enabling administrators to define and customize the specific events and conditions they wish to monitor within their infrastructure. These policies allow for fine-grained control over the data collected by Tetragon's eBPF programs, ensuring that monitoring is both relevant and efficient. By defining tracing policies, users can focus on the most critical aspects of their environment, tailoring Tetragon's capabilities to meet the unique needs of their applications and systems.

5.1.1 Defining Tracing Policies

Tetragon tracing policies are defined using a flexible and expressive policy language. These policies specify the conditions under which eBPF programs should be attached to kernel events, as well as the type of data to be collected. Tracing policies can be crafted to monitor a wide range of activities, including system calls, network traffic, file accesses, and process executions. The policy language supports a variety of matching criteria, such as process names, user IDs, and network addresses, allowing for precise targeting of monitoring activities.

For example, a tracing policy might specify that all read and write operations on a sensitive file should be monitored, or that all network connections to a particular port should be logged. By allowing such specific definitions, tracing policies ensure that Tetragon collects only the most relevant data, minimizing performance overhead and enhancing the efficiency of monitoring efforts.

5.1.2 Implementation Steps

Implementing tracing policies in Tetragon involves several steps:

- **Policy Creation**: Administrators create tracing policies using the policy language, defining the specific events and conditions they wish to monitor. This includes specifying the kernel events to attach eBPF programs to and the data to be collected.

- **Policy Deployment**: Once created, tracing policies are deployed to the Tetragon platform. This process involves using the *kubectl* cli tool to load the policies into the Tetragon system, where they are translated into eBPF programs that are dynamically inserted into the kernel.

- **Policy Management**: Administrators can manage tracing policies with the same tool that was used to depoy policies, *kubectl*. This includes enabling, disabling, and updating policies as needed. Policies can be modified in real-time, allowing for dynamic adjustment to changing conditions within the monitored environment.

5.1.3 Tracing Policies Advantages

The benefits of tracing policies in Tetragon are plenty, significantly enhancing the platform's monitoring and observability capabilities. Firstly, tracing policies provide customizable monitoring, enabling administrators to tailor data collection to their specific needs by focusing on the most relevant events and conditions. This ensures that monitoring efforts are not only effective but also efficient, as the precise criteria for data collection minimize performance overhead. Additionally, the ability to update tracing policies in real-time allows for dynamic adjustments in response to changing conditions within the monitored environment, ensuring continuous and adaptive monitoring. This flexibility is crucial for maintaining high levels of security and performance in rapidly evolving infrastructures. Tracing policies also play a pivotal role in enhancing security by enabling the enforcement of security measures directly within the kernel. For instance, administrators can define policies to terminate unauthorized processes or log suspicious activities, thereby proactively mitigating risks and maintaining system integrity. Furthermore, the detailed logs and audit trails generated by tracing policies support compliance with regulatory requirements, facilitating auditing processes and ensuring that organizations meet stringent data protection and security standards. Overall, the use of tracing policies in Tetragon provides a powerful, adaptable, and efficient means of maintaining observability, security, and compliance in modern IT environments.

5.2 Events

Tetragon's security events are pivotal for maintaining and enhancing the security posture of cloud-native environments. These events are generated by monitoring and analyzing a wide array of system activities, including system calls, network traffic, file access, and process executions. When Tetragon detects behaviors that deviate from normal patterns or match predefined threat signatures, it generates security events that provide detailed insights into these anomalies. These events are accompanied by comprehensive contextual information, such as timestamps, involved processes, and associated system resources, enabling administrators to quickly understand and respond to potential threats. Through detailed logging and alerting mechanisms, these security events also facilitate compliance with regulatory requirements and support forensic investigations, ensuring that security incidents are thoroughly documented and analyzed.

5.2.1 Visualization

Tetragon exposes its security events through two primary channels: gRPC endpoints [18] and JSON logs, providing flexible and accessible methods for administrators to integrate and utilize this critical information.

- **gRPC endpoint**: The gRPC endpoint is a robust interface that allows for real-time, programmatic access to Tetragon's security events. This endpoint facilitates seamless integration with various monitoring and incident response systems, enabling automated workflows and immediate reactions to detected anomalies. By leveraging gRPC, administrators can subscribe to a stream of security events, ensuring they receive prompt notifications of any suspicious activities. This real-time data flow supports quick decision-making and enhances the overall responsiveness of the security infrastructure. Additionally, the gRPC interface can be customized to filter specific types of events, ensuring that only the most relevant information is transmitted to the connected systems.

- **JSON logs**: In parallel, Tetragon generates detailed JSON logs for its security events. These logs provide a structured and easily readable format that can be ingested by various log management and analysis tools. JSON logs are ideal for environments that rely on centralized logging systems, as they can be easily indexed and queried for specific security events. The rich data contained within these logs includes comprehensive contextual information such as event timestamps, process IDs, user IDs, and the nature of the detected anomaly. This detailed logging not only supports real-time monitoring but also enables historical analysis, compliance reporting, and forensic investigations. By storing security events in JSON format, Tetragon ensures that all necessary information is preserved for thorough examination and future reference.

5.2.2 Event Filtering and Redacting

Tetragon offers advanced event filtering and redacting capabilities to ensure that administrators receive relevant security events while protecting sensitive information. Event filtering allows users to specify criteria that determine which events are captured and transmitted, reducing noise and focusing on critical incidents. This customization ensures that monitoring efforts are efficient and that administrators can quickly identify and respond to significant threats. In addition to filtering, Tetragon provides robust mechanisms for redacting sensitive information from events before they are logged or transmitted. This feature is crucial for maintaining data privacy and compliance with regulations, as it prevents the exposure of confidential data such as personally identifiable information (PII) or proprietary business details. By combining event filtering with sensitive information redaction, Tetragon delivers precise, secure, and compliant monitoring solutions that cater to the specific needs and policies of any organization.

5.2.3 Metrics

Tetragon enhances its observability capabilities by exporting metrics to Prometheus. This integration allows Tetragon to provide detailed, real-time metrics on system performance, security events, and resource utilization directly to Prometheus, enabling comprehensive monitoring and analysis. Administrators can leverage Prometheus' powerful querying language to analyze these metrics, create custom dashboards, and set up alerts based on specific conditions. By exporting metrics to Prometheus, Tetragon ensures that organizations can seamlessly incorporate its rich telemetry data into their existing monitoring infrastructure, facilitating proactive system management, rapid incident response, and continuous performance optimization. This integration not only enriches the overall observability ecosystem but also enhances the ability to maintain robust and resilient cloud-native environments.

5.3 Monitoring

Tetragon, as an eBPF platform, operates on various elements of the Linux kernel, enabling comprehensive monitoring of binaries, files, network sockets, and more. This capability allows Tetragon to provide detailed visibility and control over a wide range of system components, ensuring robust observability and security in complex environments.

5.3.1 Execution Monitoring

Execution monitoring is a critical aspect of system observability, enabling the tracking and analysis of program executions across diverse environments. Tetragon leverages eBPF technology to provide robust execution monitoring capabilities, making it an invaluable tool for administrators managing Kubernetes clusters, virtual machines, and bare-metal systems. By capturing and analyzing execution data in real-time, Tetragon enhances visibility, security, and performance management across complex, distributed infrastructures.

Kubernetes Monitoring

Kubernetes clusters present unique challenges for execution monitoring due to their dynamic nature and orchestration of numerous containerized applications. Tetragon addresses these challenges through several key features:

- **Pod-Level Monitoring**: Tetragon tracks the execution of processes within individual pods, providing detailed visibility into the behavior of containerized applications. This includes monitoring system calls, resource usage, and interactions between containers.

- **Namespace and Capability Annotations**: Tetragon annotates processes with their corresponding namespaces and capabilities, enabling precise monitoring and control. This helps in identifying and managing privilege escalations and other security-related events.

- **Integration with Kubernetes APIs**: Tetragon integrates with Kubernetes APIs to enhance its monitoring capabilities. This integration allows Tetragon to correlate execution data with Kubernetes objects such as pods, services, and deployments, providing a comprehensive view of the cluster's state.

Virtual Machine Monitoring

Virtual machines add another layer of complexity to execution monitoring due to their abstraction from the underlying hardware. Tetragon effectively addresses these challenges by leveraging eBPF to monitor executions at the kernel level. This allows for detailed insights into the behavior of applications running within VMs, including tracking system calls, process executions, and resource usage.

Tetragon's ability to provide cross-VM visibility is particularly beneficial, as it enables administrators to track and correlate execution events across multiple virtual machines, offering a cohesive view of application behavior and interactions. Additionally, Tetragon monitors resource allocation and usage within VMs, helping administrators to optimize performance and ensure efficient resource utilization. By integrating these capabilities, Tetragon ensures that the complexities associated with virtualized environments are effectively managed, maintaining high levels of observability and control.

Bare-metal System Monitoring

Bare-metal systems, while simpler in some respects than virtualized environments, still require robust execution monitoring to ensure security and performance. Tetragon provides comprehensive monitoring capabilities for bare-metal systems:

- **Direct Hardware Interaction**: Tetragon monitors executions that interact directly with hardware, capturing detailed telemetry on system calls and process behavior. This is crucial for understanding the performance and security implications of applications running on bare-metal systems.
- **Enhanced Security Monitoring**: Tetragon enhances security monitoring on bare-metal systems by detecting and responding to anomalous behavior. This includes identifying unauthorized access attempts and privilege escalations in real-time.
- **Performance Optimization**: By tracking resource usage and process interactions, Tetragon helps administrators optimize the performance of applications running on bare-metal systems, ensuring efficient utilization of hardware resources.

Listing 5.1: Process exec log example

```
"process_exec": {
    "process": {
        "exec_id": "Z2tlLWpvaG4tNjMyLWRlZmF1",
        "pid": 52699,
        "uid": 0,
        "cwd": "/",
        "binary": "/usr/bin/curl",
        "arguments": "https://ebpf.io/applications/#
   tetragon",
        "flags": "execve rootcwd",
        "start_time": "2023-10-06T22:03:57.700327580Z",
        "auid": 4294967295,
        "pod": {
            "namespace": "default",
            "name": "xwing",
            "container": {
                "name": "spaceship",
                "image": {
                    "name": "docker.io/tgraf/netperf:latest
   "
                },
                "start_time": "2023-10-06T21:52:41Z",
                "pid": 49
            },
            "pod_labels": {
                    "app.kubernetes.io/name": "xwing",
                    "class": "xwing",
                    "org": "alliance"
               },
               "workload": "xwing"
          },
},
"node_name": "gke-john-632-default-pool-7041cac0-9s95",
"time": "2023-10-06T22:03:57.700326678Z"
```

Tetragon's execution monitoring capabilities, powered by eBPF technology, provide unparalleled visibility into the behavior of applications across various environments such as Kubernetes clusters, virtual machines, and bare-metal systems.

5.3.2 File Access Monitoring

File access monitoring (FAM) is a crucial aspect of system security and performance management, providing insights into how files are accessed, modified, and utilized within an environment. Tetragon, leveraging eBPF technology, offers robust file access monitoring capabilities that are essential for administrators managing complex infrastructures. By capturing detailed telemetry on file operations in real-time, Tetragon enhances visibility, security, and compliance across diverse environments.

eBPF allows Tetragon to attach small, efficient programs to various kernel events related to file operations, such as open, read, write, and close. These programs capture detailed information about file access patterns, which is then processed and analyzed in user space. This approach ensures minimal performance overhead while providing granular visibility into file activities. Moreover, Tetragon leverages eBPF technology to access the Linux file struct, enabling it to gather detailed information about files directly from the system. An example of the information that can be obtained are file paths, permissions and ownership.

This approach however has a few downsides since it uses the path to match what file is being accessed. Doing this can lead to certain events not being tracked as the same file is, for example, accessed via a hard link. Even if this type of links are created only by users with elevated permissions it is still a flaw we should keep in mind. To solve this problem completely though we can rely on the usage of inode numbers, which uniquely identify a file within the file system [19].

5.3.3 File Integrity Monitoring

Complementary to file access monitoring is the concept of file integrity monitoring (FIM) that helps us guarantee the integrity and content of our file system. In addition to FAM which checks if certain files have been accessed, FIM on the other hand checks the computes hashes of those files. This makes sure that since the last check the contents of the file have not been modified or deleted.

To accomplish this goal we can rely on Linux's Integrity Measurement Architecture (IMA-measurement) [20], which provides a framework for maintaining the integrity of files on a system. Similarly to policies made for FAM where we track syscalls such as open, here we can track the *bprm_check_security* hook and use the IMA *ima_file_hash operator* inside our tracing policy to generate events with the computed file hash.

5.3.4 Network Monitoring

Tetragon's network monitoring capabilities leverage the power of eBPF technology to provide deep, real-time visibility into network traffic and interactions within modern cloud-native environments. By attaching eBPF programs to various network-related kernel events, Tetragon can capture detailed telemetry on packet flows, connection states, and network errors without significant performance overhead. For example we can easily track tcp connections using kprobes within Tetragon, specifically tcp_connect which hooks into the tcp_v4_connect kernel function to track incoming and outgoing connections. This allows administrators to monitor network activities at a granular level, facilitating the detection of anomalous behavior, performance bottlenecks, and security threats. Tetragon's integration with Kubernetes and other orchestration platforms ensures that network monitoring is context-aware, correlating network events with specific pods, services, and deployments. This comprehensive approach to network monitoring enhances the security, reliability, and performance of complex infrastructures, making Tetragon an indispensable tool for managing dynamic and distributed systems.

5.4 Policy Enforcement

Beyond monitoring, Tetragon's unique strength lies in its ability to enforce policies directly within the kernel. By integrating user-defined policies and Kubernetes context, Tetragon can dynamically respond to specific events and conditions, enforcing security and operational rules at the system level. For example, it can trigger alerts, log activities, or even terminate processes based on predefined policies. For example, by issuing a SIGKILL signal to the process that has triggered an event, it can stop its execution before it goes any further. This dual capability of monitoring and policy enforcement allows Tetragon to not only detect and analyze issues as they occur but also to proactively mitigate risks and maintain compliance with security and operational standards. The combination of real-time observability and automated policy enforcement makes Tetragon an indispensable tool for managing and securing modern, complex infrastructures.

Tetragon employs two primary mechanisms for policy enforcement: overriding return values and sending signals. These mechanisms allow Tetragon to dynamically influence system behavior in response to specific conditions, providing administrators with powerful tools to maintain security and operational integrity.

5.4.1 Overriding Return Values

One of Tetragon's key policy enforcement capabilities is the ability to override return values of system calls. When a monitored event meets certain predefined conditions, Tetragon can alter the return value of the associated system call. This mechanism effectively prevents unauthorized or undesirable actions from being executed by manipulating the outcome of critical system operations. For example, if an application attempts to access a sensitive file or change its privileges in a way that violates security policies, Tetragon can override the return value to indicate failure, thereby blocking the action. This approach ensures that potential security threats are mitigated in real-time, enhancing the protection of the system against malicious or erroneous activities.

5.4.2 Signals

In addition to overriding return values, Tetragon can enforce policies by sending signals to processes. Signals are a fundamental mechanism in Unix-like operating systems for handling asynchronous events. Tetragon leverages this capability to enforce immediate and decisive actions on processes that violate predefined policies. For instance, if a process exhibits behavior that matches a security threat, such as executing unauthorized commands or accessing restricted resources, Tetragon can send a signal to terminate the process (e.g., SIGKILL) or to trigger a specific action (e.g., SIGTERM). This ability to send signals provides a flexible and powerful means to control process behavior, enabling administrators to swiftly respond to potential threats and maintain system stability and security.

The integration of these enforcement mechanisms within Tetragon's monitoring framework ensures that policy enforcement is tightly coupled with real-time observability. Administrators can define complex policies that leverage both overriding return values and sending signals, tailoring responses to the specific needs and security requirements of their environment. The flexibility of Tetragon's policy enforcement mechanisms allows for fine-grained control over system operations, making it possible to enforce nuanced security measures and operational rules that are critical for maintaining the integrity and reliability of cloud-native infrastructures.

Chapter 6

Implementation

In this chapter, I will discuss my research project involving Tetragon and its potential to enhance observability and enforcement in a large, virtualized 5G network using Kubernetes. We will explore how Tetragon can collect crucial data from both the underlying system and the Kubernetes cluster it operates within. Additionally, we will examine how this tool can centralize and organize the gathered data, making it easier to visualize and understand what is happening in our network. This centralized approach aims to provide a clearer and more comprehensive view of network activities, thereby improving our ability to monitor and manage the 5G infrastructure effectively.

The main challenge we faced in monitoring a large network of hundreds of machines, typical in a 5G network, was centralizing all our observability data. If we used traditional eBPF programs manually deployed on each machine, we would be overwhelmed by the sheer volume of logs generated. Instead, by using Tetragon, we demonstrate how this process can be streamlined, making it much simpler and more efficient.

6.1 Proposed solution

In light of the previous chapter, we have undertaken the task of setting up a workflow for Tetragon to gain a comprehensive understanding of its inner workings and to explore the process of writing custom policies tailored to our specific needs. This section details the steps we followed, the challenges we encountered, and the solutions we implemented to effectively integrate Tetragon into our 5G network environment.

6.1.1 Studying the use cases

Before starting to work on the actual implementation, we took considerable time to thoroughly brainstorm and analyze the potential use cases for our solution. Recognizing the complexity and unique requirements of our 5G network environment, we aimed to ensure that our efforts with Tetragon would address the most critical and impactful scenarios. With the guidance and expertise of the rest of the team, we managed to form a comprehensive list of use cases that would benefit significantly from Tetragon's advanced monitoring and policy enforcement capabilities.

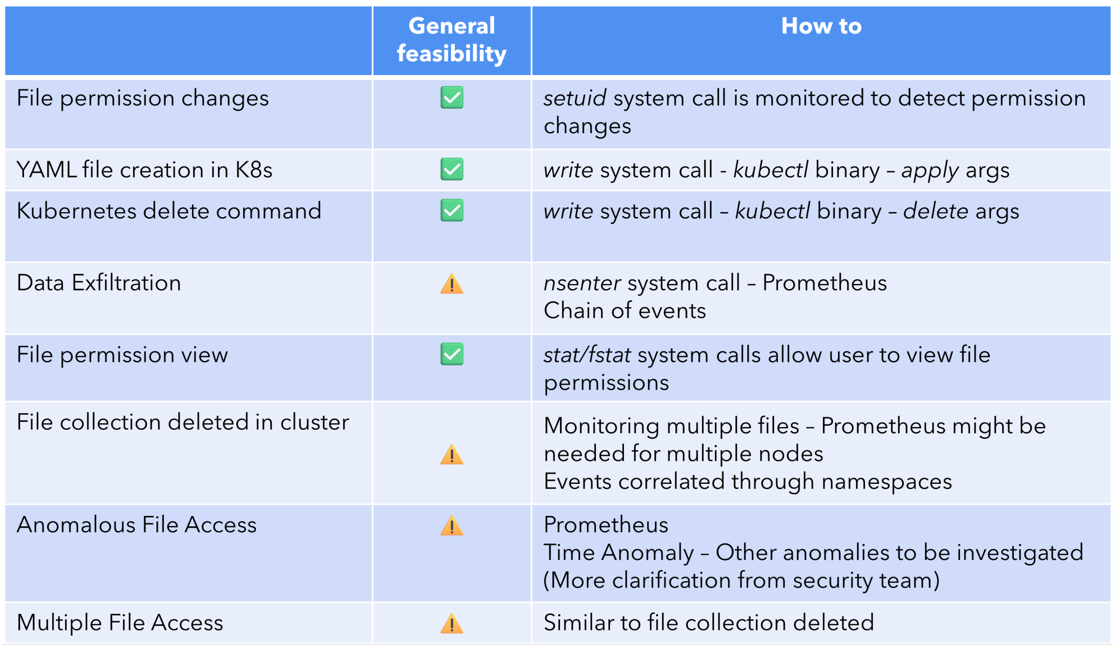
Starting from basic use cases that simply monitor files being accessed or modified, we expanded our scope to include more nuanced and complex scenarios.

- Initially, our focus was on establishing a baseline of file access activities, capturing details such as which files were being accessed, by whom, and when. This foundational monitoring allowed us to understand normal usage patterns and detect any deviations that might indicate unusual behavior.

- As we delved deeper, we recognized the importance of identifying more sophisticated threats, such as malicious access attempts to the nodes in our cluster. This required us to define policies that could detect access to sensitive files at unusual times, which is often a red flag for potential security incidents. For instance, legitimate users typically access critical files during regular working hours, so any attempts to access these files late at night or during weekends would trigger an alert.

- Moreover, we identified the need to monitor for patterns indicative of ill-intentioned actors, such as attempts to access a large number of files in a short period. This behavior is characteristic of certain types of attacks, such as data exfiltration or ransomware activities, where an attacker tries to read or encrypt as many files as possible before being detected. To address this, we implemented more complex policies that could recognize these patterns and respond accordingly.

- In addition to monitoring file access, we extended our use cases to include the detection of unauthorized file modifications. This involved tracking changes to critical configuration files, where unauthorized modifications could lead to significant disruptions or security breaches. By setting up alerts for unexpected changes, we could quickly identify and respond to potential threats.



	General feasibility	How to
File permission changes	✓	*setuid* system call is monitored to detect permission changes
YAML file creation in K8s	✓	*write* system call - *kubectl* binary – *apply* args
Kubernetes delete command	✓	*write* system call - *kubectl* binary – *delete* args
Data Exfiltration	⚠	*nsenter* system call – Prometheus Chain of events
File permission view	✓	*stat/fstat* system calls allow user to view file permissions
File collection deleted in cluster	⚠	Monitoring multiple files – Prometheus might be needed for multiple nodes Events correlated through namespaces
Anomalous File Access	⚠	Prometheus Time Anomaly – Other anomalies to be investigated (More clarification from security team)
Multiple File Access	⚠	Similar to file collection deleted

Figure 6.1: Tetragon use cases

After identifying all the previously stated use cases, we began to strategize on how we could implement them using Tetragon and assess their feasibility within the platform. This process involved a detailed examination of Tetragon's capabilities, including its ability to handle complex monitoring requirements and enforce sophisticated security policies. We reviewed the documentation, experimented with different configurations, and consulted with the team to understand the practical limits and strengths of the platform. Following careful consideration and initial testing, we established two main groups of use cases:

- **Supported by Tetragon**: Use cases such as the simple detection of file modifications or permission changes using system calls like setuid can be entirely realized with Tetragon. This involves configuring Tetragon to monitor specific system calls that alter file attributes or permissions, thereby providing real-time alerts whenever such actions occur. For instance, if a critical configuration file's permissions are modified unexpectedly, Tetragon can detect this change and trigger an alert or block the system call, allowing administrators to investigate and respond swiftly. Furthermore, tracking the execution of certain sensitive commands, such as kubectl, is also feasible using Tetragon policies. By setting up policies to monitor the execution of these commands, Tetragon can provide visibility into administrative actions that might impact the Kubernetes cluster's state. This is particularly important in environments where the security and integrity of cluster operations need to be tightly controlled. For example, if an unauthorized user attempts to execute kubectl commands to modify cluster configurations or deploy resources, Tetragon can detect this activity and take predefined actions, such as logging the event, alerting the security team, or even blocking the command execution.

- **Needing additional support**: Whenever there is a need for further data processing, such as in the case of anomaly detection, additional tools become essential to augment Tetragon's capabilities. Anomaly detection often requires sophisticated analysis and correlation of large volumes of data to identify patterns that deviate from normal behavior, which are indicative of potential security threats or malicious activities. Tetragon excels at collecting detailed telemetry and enforcing real-time policies, but the subsequent data processing and analysis typically necessitate the integration of complementary tools.

6.1.2 Studying System Calls

After thoroughly assessing our use cases, the next crucial step was to determine which system calls to monitor using our policies to achieve our goal. This phase involved a detailed analysis of the activities and events that were most critical to our infrastructure's security and operational integrity. System calls, being the primary interface between user applications and the kernel, provide a rich source of information about the behavior and state of the system. By selectively monitoring specific system calls, we could gather precise and actionable data to meet our security and observability objectives.

To begin, we identified the primary goals of our monitoring efforts. These goals included detecting unauthorized access to sensitive files, tracking changes to critical system configurations, monitoring the execution of key administrative commands, and observing network-related activities that could indicate security breaches or misconfigurations. Each of these goals required a tailored approach to system call monitoring, ensuring that we captured relevant data without overwhelming the system with excessive logging.

File Monitoring

We then mapped these goals to specific system calls that would provide the necessary insights. For example, to monitor file access and modifications, we focused on system calls such as *open, openat*, *read* and *write*. These calls allowed us to detect when sensitive files were accessed or altered, providing early warning of potential security incidents. Another particularly useful syscall is the kprobe *fd_install*, which plays a crucial role in our monitoring strategy by enabling us to detect when a file has been loaded into memory and is prepared for access by the open syscall.

This syscall is invoked before any program can interact with the file, giving us a valuable early point of intervention. By monitoring fd_install, we can proactively track which files are being prepared for access, allowing us to implement security measures such as denying access or logging attempts to open files.

Additionally, we utilized the *FollowFD* action provided by Tetragon, which creates an eBPF map to track monitored files by their File Descriptor (FD). In the Linux OS, each file is assigned a unique FD, acting as an identifier. This feature allowed us to efficiently monitor file activities, ensuring that we could track access and modifications accurately by associating these actions with their respective FDs. This capability was crucial for maintaining precise oversight over sensitive files, enhancing our ability to detect and respond to unauthorized file operations effectively.

Permission Changes

Monitoring system calls related to permission changes is critical for maintaining the security and integrity of a system. System calls like *chmod, chown* and *setfacl* are used to modify file permissions, ownership, and access control lists, respectively. These modifications can significantly impact the security posture of the system by altering who can read, write, or execute files. By tracking these syscalls, we can gain valuable insights into changes made to file permissions, allowing us to detect potentially unauthorized or malicious activities promptly.

Using eBPF-based tools like Tetragon, we can attach probes to these syscalls to capture detailed information whenever they are invoked. For instance, when a chmod syscall is executed, the probe can record the target file, the new permission settings, the user who initiated the change, and the exact time of the modification. This data is then logged and analyzed to ensure that all permission changes are legitimate and comply with security policies.

6.1.3 Testing Use Cases

After the initial study of the use cases for Tetragon, we embarked on a testing phase to evaluate its functionality within a controlled environment. To do this, we set up a smaller cluster consisting of just three nodes as testing on a large cluster that mimicks a 5G network would not be feasible. This scaled-down cluster provided a manageable yet sufficiently complex setting to observe how Tetragon operates and interacts with both the underlying system and the cluster as a whole. Our goal was to understand Tetragon's capabilities in a real-world scenario, focusing on its observability and security features.

We began by deploying Tetragon on our three-node cluster, carefully configuring it to monitor various system and network activities. This setup allowed us to observe how Tetragon collects and processes data from multiple sources, including system calls, network traffic, and file system events. By examining these interactions, we aimed to assess Tetragon's ability to provide deep visibility into system behaviors and its efficiency in identifying potential security threats.

File Monitoring

To thoroughly assess Tetragon's capabilities, we initiated a series of tests focusing on its file monitoring features. We began by deploying specific file monitoring policies aimed at detecting any reads and writes to sensitive files within our three-node cluster. These policies were carefully crafted to trigger alerts whenever critical system files, such as configuration files or security logs, were accessed or modified. Each event was logged with detailed context, allowing us to analyze the nature and potential impact of the file operations. The results of these tests demonstrated Tetragon's robust capability to enhance our security posture by providing precise and timely alerts on sensitive file activities, proving it to be an invaluable tool for maintaining the integrity and security of our systems.

Listing 6.1: Example of a file access policy tracking the 'passwd' file

```yaml
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: file-activity
spec:
  kprobes:
    - call: fd_install
      syscall: false
      args:
        - index: 0
          returnCopy: false
          type: int
        - index: 1
          returnCopy: false
          type: file
      selectors:
        - matchActions:
            - action: FollowFD
              argFd: 0
              argName: 1
          matchArgs:
            - index: 1
```

```
                  operator: Equal
                  values:
                    - /etc/passwd
```

Permission Monitoring

Afterwards, we focused on implementing policies designed to track file permission changes, particularly targeting sensitive system files. This effort was crucial in enhancing our system's security posture, ensuring that any unauthorized or suspicious modifications to file permissions could be detected and prevented in real time. By monitoring specific syscalls associated with permission-changing commands such as *chmod* we were able to gain comprehensive visibility into all permission alterations occurring within the system. Leveraging Tetragon's use of eBPF technology allowed us to intercept these syscalls at an early stage, even before they were fully executed. This preemptive monitoring capability meant that we could detect attempts to change file permissions as soon as the syscall was invoked, rather than waiting for the action to complete. As a result, we had the opportunity to block potentially harmful changes before they could impact the system, effectively neutralizing threats in their nascent stages. For instance, if a malicious actor attempted to escalate their privileges by changing the permissions of a sensitive configuration file or a critical system binary, our monitoring policies would immediately detect this activity. Tetragon would then use eBPF to log the attempt and block the syscall, thereby preventing the change from taking place. This not only thwarted the immediate threat but also generated detailed logs that could be used for forensic analysis and future threat assessments. Furthermore, this approach provided us with a robust audit trail of all permission changes. Every invocation of a permission-changing syscall was recorded, along with relevant details such as the user who initiated the command, the file targeted, the old and new permissions, and the timestamp of the event. This data was invaluable for maintaining compliance with security policies and regulations, as it allowed us to demonstrate control over critical system configurations.

Listing 6.2: Example of a log file generated from tracking the 'sys_chmodat' syscall

```
{
  "process_tracepoint": {
    "process": {
      "exec_id": "
    aXQtc3VzaGktMToyMjkzMDAxMTI3NDU3MDE6NTIxNDM3",
      "pid": 521437,
      "uid": 1000,
      "cwd": "/home/sushi",
      "binary": "/usr/bin/chmod",
      "arguments": "700 testing",
      "flags": "execve clone",
      "start_time": "2024-02-05T17:13:31.166002420Z",
      "auid": 4294967295,
      "parent_exec_id": "
  aXQtc3VzaGktMToyMDU0MjE5MjAwMDAwMDA6Mjc4NTc=",
      "refcnt": 1,
      "tid": 521437
    },
    "subsys": "syscalls",
    "event": "sys_enter_fchmodat",
    "args": [
      {
        "long_arg": "268"
      }
    ],
    "policy_name": "syscall-fchmodat",
    "action": "KPROBE_ACTION_POST"
  },
  "node_name": "cluster-node-1",
  "time": "2024-02-05T17:13:31.166278805Z"
}
```

Kubectl Monitoring

We also implemented a policy to monitor the execution of the kubectl command, which is crucial for managing Kubernetes clusters. This policy was specifically designed to log the attributes and parameters passed with each kubectl invocation, allowing us to track and analyze all changes made to the cluster in detail. By capturing this data, we were able to audit administrative actions comprehensively, ensuring that any modifications to the cluster configuration were both intentional and authorized.

This detailed logging provided visibility into command usage patterns, which was instrumental in detecting potential misconfigurations or unauthorized attempts to alter cluster resources. For instance, we could see who executed kubectl commands, what changes were attempted, and the context of these changes. This level of scrutiny helped us to quickly identify and respond to any suspicious activity that could compromise the cluster's security or stability.

Moreover, this monitoring policy allowed us to create a historical record of all administrative actions, which is valuable for both operational oversight and compliance purposes. By analyzing this log data, we could improve our operational practices and ensure that our cluster management policies were being followed correctly. Overall, this policy significantly enhanced our ability to maintain the security and integrity of our Kubernetes environment, providing us with the necessary tools to oversee and safeguard our infrastructure effectively.

Prometheus Integration

Furthermore, we observed the native integration of Tetragon with Prometheus, which was a significant aspect of our evaluation. Tetragon exposes its own metrics endpoint, allowing Prometheus to collect and monitor various metrics seamlessly. This integration enabled us to capture both system-level metrics and policy-related metrics, providing a comprehensive overview of our system's behavior and security posture. The metrics endpoint revealed valuable information, giving us insights into potential security threats or policy violations.

For instance, we could quantify the number of times sensitive file monitoring policies were activated, indicating attempts to read or write to critical system files. Additionally, metrics on binary executions helped us track how often certain binaries were called, which is essential for detecting unusual or malicious activities. This real-time data was instrumental in understanding the effectiveness of our deployed policies and the overall security landscape of our infrastructure. By leveraging Prometheus' powerful querying and alerting capabilities, we could set up custom alerts based on these metrics, ensuring immediate response to any suspicious activities. This integration showcased how Tetragon enhances our monitoring capabilities, providing detailed, actionable insights that are crucial for maintaining a secure and well-monitored system environment.

6.1.4 Log Gathering

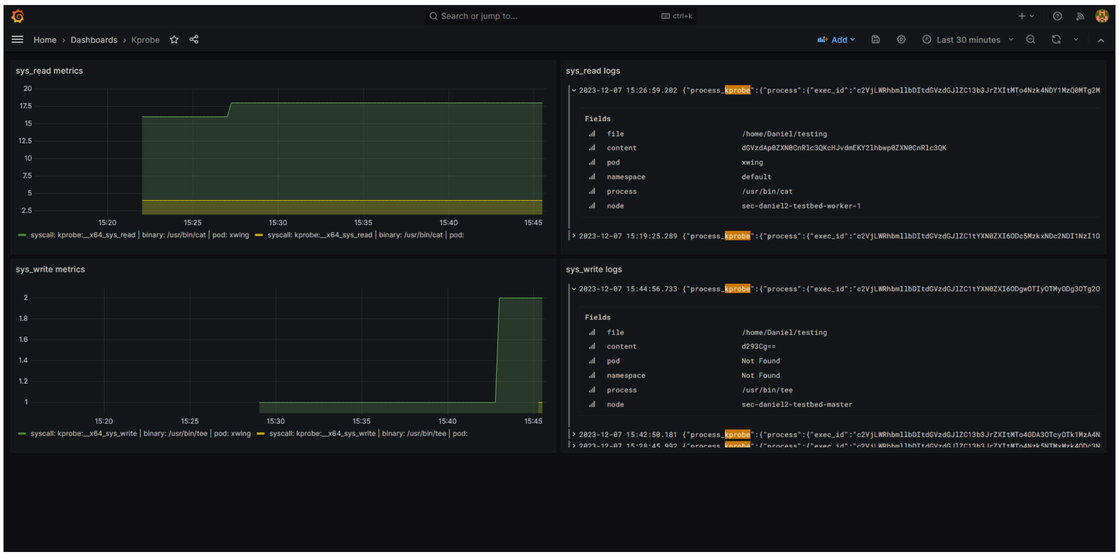
After defining and testing our use cases, we needed to obtain the logs that Tetragon was generating to study and analyze them for a better understanding of what is going on in our system and cluster. There are two primary methods for obtaining this data, each with its own advantages and disadvantages. The Prometheus endpoint that Tetragon exposes provides basic metrics on the events that triggered in our system, such as counts of occurrences and timestamps. However, it does not offer detailed information about each individual event. Conversely, the log files that Tetragon stores in the system in JSON format offer a much richer source of data. These logs contain comprehensive details about the triggered events, including the name of the node where the event occurred, the namespace, pod, binary, and other pertinent information.

Collecting and storing these JSON logs requires setting up an effective log management system. We considered using centralized logging solutions like Loki, integrated with Promtail, to forward these logs to a centralized location. This setup would enable us to use powerful querying and visualization tools such as Grafana to sift through the data, correlate events, and gain deeper insights into our system's behavior. Centralized logging would also facilitate long-term storage and archival of logs, ensuring that historical data is readily available for trend analysis and forensic investigations, thereby enhancing our ability to monitor and secure our infrastructure effectively.

Setting up Promtail and Loki

Since Tetragon is running inside our cluster, we need to set up Promtail so that it is deployed on every single one of our nodes. This requires configuring a deployment YAML file, which specifies the deployment details and ensures that a DaemonSet is created across all nodes. A DaemonSet ensures that Promtail runs on each node, providing it with the necessary access to Tetragon's log paths. This setup allows Promtail to collect log data efficiently and tail the log files, preventing duplicate entries by ensuring it only processes new log entries. To implement this, we carefully configured the YAML file to include all necessary permissions and paths for Promtail. Additionally, we configured Promtail to handle log rotation and ensure continuous log monitoring without data loss. Once the YAML configuration was complete, we applied it to our cluster, creating the Promtail DaemonSet and verifying its correct deployment on all nodes. After successfully deploying Promtail, we needed an instance of Loki to receive the aggregated log data. Setting up Loki involved creating another YAML file to define its deployment configuration. This file specified the creation of pods for Loki, a service endpoint for Promtail to send data to, and a persistent volume for storing the collected log data. The persistent volume configuration was crucial to ensure that Loki could store large amounts of log data reliably over time, facilitating long-term analysis and historical log retention. We configured Loki to listen on a specific port for incoming data from Promtail, ensuring smooth data flow between the two components. Additionally, we set up the necessary authentication and authorization mechanisms to secure the log data during transit. Once the Loki configuration was applied to the cluster, we verified its connectivity with Promtail, ensuring that logs were being forwarded and stored correctly. This end-to-end setup enabled a robust logging infrastructure, allowing us to monitor, analyze, and store Tetragon logs efficiently.

Setting up Grafana


Search or jump to...
ctrl+k
Home › Dashboards › Kprobe
Add
Last 30 minutes
sys_read metrics
20
17.5
15
12.5
10
7.5
5
2.5
15:20
15:25
15:30
15:35
15:40
15:45
syscall: kprobe:__x64_sys_read | binary: /usr/bin/cat | pod: xwing
syscall: kprobe:__x64_sys_read | binary: /usr/bin/cat | pod:
sys_write metrics
2
1.8
1.6
1.4
1.2
1
15:20
15:25
15:30
15:35
15:40
15:45
syscall: kprobe:__x64_sys_write | binary: /usr/bin/tee | pod: xwing
syscall: kprobe:__x64_sys_write | binary: /usr/bin/tee | pod:
sys_read logs
sys_write logs

```
2023-12-07 15:26:59.202 {"process_kprobe":{"process":{"exec_id":"c2VjLWRhbmllbDItdGVzdGJlZC13b3JrZXItMTo4Nzk4NDY1MzQ0MTg2M

Fields
   file          /home/Daniel/testing
   content       dGVzdAp0ZXN0CnRlc3QKcHJvdmEKY2lhbwp0ZXN0CnRlc3QK
   pod           xwing
   namespace     default
   process       /usr/bin/cat
   node          sec-daniel2-testbed-worker-1

2023-12-07 15:19:25.289 {"process_kprobe":{"process":{"exec_id":"c2VjLWRhbmllbDItdGVzdGJlZC1tYXN0ZXI6ODc5MzkxNDc2NDI1NzI1O
```

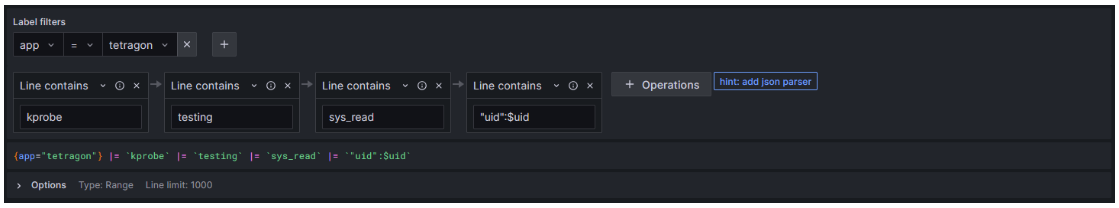
```
2023-12-07 15:44:56.733 {"process_kprobe":{"process":{"exec_id":"c2VjLWRhbmllbDItdGVzdGJlZC1tYXN0ZXI6ODgwOTIyOTMyODg3OTg2O

Fields
   file          /home/Daniel/testing
   content       d293Cg==
   pod           Not Found
   namespace     Not Found
   process       /usr/bin/tee
   node          sec-daniel2-testbed-master

2023-12-07 15:42:50.181 {"process_kprobe":{"process":{"exec_id":"c2VjLWRhbmllbDItdGVzdGJlZC13b3JrZXItMTo4ODA3OTcyOTk1MzA4N
2023-12-07 15:28:45.992 {"process_kprobe":{"process":{"exec_id":"c2VjLWRhbmllbDItdGVzdGJlZC13b3JrZXItMTo4Nzk5NTMxMzk4ODc3N
```

Figure 6.2: Dashboard configured with events gathered in our system

Similarly to Loki, setting up Grafana involves deploying several key components within our Kubernetes cluster. We begin by creating a deployment YAML file for Grafana, which specifies the creation of pods, a service, and a persistent volume. The pods will run the Grafana instances, the service will expose Grafana to other components within the cluster, and the persistent volume will store our dashboards, configurations, and connections to logging tools such as Loki. In addition to these components, we also need to set up an ingress resource. The ingress resource is critical because it allows external access to the Grafana frontend, enabling us to interact with Grafana's powerful visualization and dashboard capabilities from outside the cluster. The ingress configuration includes specifying the domain or subdomain through which Grafana will be accessible, as well as setting up TLS for secure communication.

After verifying that Grafana is running correctly and accessible via the specified ingress endpoint, the next step is to configure Grafana to work with Loki. This involves logging into the Grafana interface and adding Loki as a data source. We provide the Loki endpoint defined during the Loki setup process, along with any necessary authentication credentials. With Loki configured as a data source in Grafana, we proceed to create dashboards tailored to our monitoring needs. These dashboards will be designed to display the log data collected by Promtail and stored in Loki, allowing us to visualize trends, identify anomalies, and gain insights into system behavior. Grafana's rich visualization options, including graphs, charts, and tables, enable us to create detailed and interactive dashboards that provide a comprehensive view of our system's health and performance.

Since Promtail gathers logs in raw JSON format, it's crucial to understand the structure of these logs to properly parse them using LogQL, the query language for Loki. This knowledge allows us to extract the specific data we need to display in our Grafana dashboards effectively. For logs generated from a File Monitoring event, we focus on parsing key pieces of information such as the file name, the binary used to access the file, the system call invoked, the user ID that performed the access, and the node where the file is stored. Understanding these elements is essential because it helps us filter and visualize the most relevant data points, providing clear insights into file access patterns and potential security incidents. For instance, knowing which binary accessed a file can help identify if a non-standard or malicious program is trying to read sensitive information. Similarly, tracking user IDs (UIDs) allows us to monitor and audit user activities, ensuring compliance with security policies. Promtail also logs the timestamp of each event, which is invaluable for creating time series data. This temporal information allows us to plot log entries on a timeline, helping to visualize changes in log volume and identify when specific events occurred. By leveraging Grafana's powerful visualization capabilities, we can create graphs that show spikes or trends in file access activity, making it easier to spot anomalies or unusual behavior.



```
{app="tetragon"} |= `kprobe` |= `testing` |= `sys_read` |= `"uid":$uid`
```

Figure 6.3: Example of a logql filter query to extract the logs containing the 'sys_read' syscall

6.1.5 Log Usage

In this chapter, we will evaluate whether the data provided by Tetragon is adequate for creating effective observability dashboards and alerts. This involves analyzing the depth and accuracy of the collected data to ensure it meets our monitoring needs and supports proactive system management and security measures.

Initially, we attempted to use the data provided by the Prometheus endpoint to create our observability dashboards and alerts. However, it quickly became apparent that this approach was insufficient for our needs. The Prometheus endpoint was limited in its capabilities, offering only basic timestamp information for each event. While this allowed us to know that an event had occurred, it did not provide any of the detailed context necessary for thorough analysis and actionable insights. For example, the endpoint did not include crucial details such as the specific file that was accessed, the binary responsible for the access, the system call invoked, the user ID involved, or the node where the event occurred. Without this granular data, we were unable to fully understand the nature of the events or their potential impact on our system. This lack of detailed information meant that we could not effectively track the precise activities occurring within our infrastructure, making it difficult to identify patterns, diagnose issues, or respond to potential security threats.

Recognizing these limitations, we realized that we needed a more comprehensive solution to achieve our observability goals. We turned our attention to the JSON log files generated by Tetragon, which contained the detailed information we required. These logs provided a wealth of data, including file names, binaries, system calls, user IDs, and node information, all of which were essential for creating meaningful and actionable dashboards and alerts. By switching our focus to these detailed log files, we were able to parse and analyze the data more effectively. This approach allowed us to extract the necessary information to build robust monitoring tools, capable of providing deep insights into system behavior and security events. Through careful parsing and visualization of this data, we ensured that our observability efforts were comprehensive and precise, enabling proactive system management and enhanced security monitoring.

Despite our decision to use JSON logs to gather detailed event information, we faced another challenge: filtering and grouping these events to create comprehensive graphs and visualizations tailored to each of our use cases. The raw JSON logs provided us with a rich dataset, but the sheer volume and complexity of the data required effective tools and methods for parsing and analysis. To address this, we turned to Loki LogQL, a powerful query language designed specifically for querying logs stored in Loki. LogQL allows us to filter and group log events based on their JSON properties, making it possible to extract precisely the data we need for each specific use case. However, before we could fully leverage LogQL, we needed to familiarize ourselves with its syntax and capabilities.

As we became more proficient with LogQL, we were able to create increasingly sophisticated queries that combined multiple filters and groupings. This allowed us to build detailed and dynamic dashboards in Grafana, where we could visualize the data through various chart types, including line graphs, bar charts, and heatmaps. These visualizations made it easier to identify trends, detect anomalies, and understand the overall behavior of our system. Despite our decision to use JSON logs to gather detailed event information, we faced another challenge: filtering and grouping these events to create comprehensive graphs and visualizations tailored to each of our use cases. The raw JSON logs provided us with a rich dataset, but the sheer volume and complexity of the data required effective tools and methods

for parsing and analysis.

To address this, we turned to Loki LogQL, a powerful query language designed specifically for querying logs stored in Loki. LogQL allows us to filter and group log events based on their JSON properties, making it possible to extract precisely the data we need for each specific use case. However, before we could fully leverage LogQL, we needed to familiarize ourselves with its syntax and capabilities.

We began by exploring the fundamental concepts of LogQL, such as log streams, labels, and filter expressions. Log streams in Loki are categorized by labels, which are key-value pairs that describe the log entries. By understanding how to utilize these labels effectively, we could begin to craft queries that would filter logs based on various JSON properties, such as the name of the file accessed, the binary used, the system call invoked, the user ID, and the node where the event occurred.

For instance, to track file access events, we crafted LogQL queries that filtered logs by the filename property, allowing us to group events based on the specific files being accessed. Similarly, by filtering on the binary property, we could identify which executables were responsible for accessing certain files, providing insights into application behavior and potential security threats. The user id property enabled us to monitor and audit user activities, ensuring compliance with security policies and detecting unauthorized access attempts.

One of the significant challenges we faced was managing the different UIDs that appeared in the generated logs. In the Linux operating system, UIDs are assigned to users to distinguish between different accounts. While this system works well on individual machines, it poses significant challenges in a distributed environment like a cluster. The same UID can correspond to different users on different nodes, and vice versa, making it nearly impossible to map UIDs to specific individuals without a centralized identity management system. This limitation significantly impacts our ability to perform accurate user attribution for security and compliance purposes. In a large cluster with dozens or even hundreds of machines, this task becomes exceptionally complex. Each node in the cluster might have different users and corresponding UIDs, making it difficult to consistently identify the individuals behind specific actions, even when the UID is available in the Tetragon event logs. This variability complicates efforts to track and correlate activities across multiple nodes, hindering our ability to maintain a clear and cohesive audit trail. Despite this challenge, we found that tracking the UID 0 can be particularly useful. Tracking UID 0, which is always associated with the root user who has all the privileges inside the system, is critical. Monitoring these UID helps us identify actions that could affect system integrity and security, such as changes to critical files, installation of software, and other administrative tasks.

6.2 Evaluating Feasability

In light of our work understanding and working with Tetragon, we discovered a few key elements that helped us evaluate this tool and how it can be integrated in our observability suite for our 5G network.

Firstly, considering our proposed use cases, we have determined that implementing a solution using Tetragon could be feasible, especially for monitoring critical security events. Tetragon's capability to immediately log any intrusions or malicious access attempts is particularly valuable in the context of a 5G network, where the speed and volume of data transactions are significantly higher. The real-time visibility provided by Tetragon can help safeguard sensitive operations and data exchanges that are critical in a 5G environment. This data includes detailed logs of system calls, file accesses, network interactions, and user activities, all of which are invaluable for identifying patterns associated with security threats.

A significant concern regarding the deployment of Tetragon in our large 5G network is the effective separation and identification of user access. In a 5G environment, where multiple users and devices are constantly interacting with the system, accurately tracking user activities is essential for security and operational management. While Tetragon does provide the UID responsible for specific events, it lacks the capability to cross-reference these UIDs with usernames by accessing the local machine's 'passwd' file. This limitation poses a considerable challenge for our use case. In a large-scale 5G network comprising thousands of machines, the inability to directly link UIDs to usernames severely hampers our ability to identify and trace the sources of security incidents quickly and efficiently. The necessity to manually sift through different machines to match UIDs with usernames is not only time-consuming but also impractical, especially during critical situations where timely responses are crucial. This shortcoming could lead to significant delays in identifying the perpetrators of malicious activities, potentially allowing security breaches to escalate unchecked. The lack of direct UID-to-username resolution in Tetragon means that we might miss out on crucial contextual information needed to accurately attribute actions to specific users. This is particularly problematic in a 5G network where user activities can vary widely and be highly dynamic. Without the ability to easily map UIDs to usernames, we lose an important layer of visibility and accountability, making it difficult to enforce user-specific policies or track user behavior effectively. To mitigate this issue, we would need to implement additional systems or scripts to automate the process of resolving UIDs to usernames across the cluster. This could involve setting up a centralized database or directory service that maintains a consistent mapping of UIDs to usernames for all machines in the cluster. However, this approach introduces additional complexity and potential points of failure, and it requires ongoing maintenance to ensure the mappings remain up-to-date and accurate.

Another significant concern is the visualization and management of the vast amounts of data generated by logs and events from thousands of machines in our 5G network. The current setup with Tetragon involves deploying a DaemonSet on each node, which makes log collection more complex. Ideally, having a single endpoint for all generated logs would streamline the process, reducing the overhead associated with managing log data across numerous nodes. However, Tetragon's design necessitates the deployment of a Promtail agent on each node, as it creates independent log files per node. This decentralized logging system requires us to gather and organize data from multiple sources, adding to the complexity of our monitoring setup. Once the logs are aggregated, we use Grafana to create centralized dashboards for visualization, which somewhat mitigates the complexity. Grafana's powerful visualization capabilities make it easier to analyze and interpret the collected data once the log gathering infrastructure is in place. Despite this, the lack of built-in automation within Tetragon presents a challenge. There is no straightforward way to set up thresholds or alerts that trigger based on the occurrence of specific events directly within Tetragon. As a result, we must rely on Grafana to handle these alerting mechanisms. This introduces latency between the occurrence of an event and the alert reaching the system administrators. Such delays can be critical in a high-speed 5G environment where timely responses to security incidents are paramount. The current workflow involves logs being generated and stored by Tetragon, collected by Promtail, sent to Loki, and finally visualized and monitored in Grafana. Any latency in this pipeline can hinder our ability to react swiftly to potential threats. An ideal solution would integrate more proactive monitoring capabilities directly within Tetragon, leveraging eBPF to detect and respond to malicious activities before they can cause harm. By enabling real-time thresholds and alerts at the eBPF level, we could significantly enhance our observability and response times. This would allow us to detect and mitigate threats almost instantaneously, improving the overall security posture of our 5G infrastructure. While Grafana provides an effective platform for visualizing and analyzing log data, the lack of direct automation and real-time alerting in Tetragon creates a gap in our monitoring strategy. Addressing this issue would involve enhancing Tetragon's capabilities to include real-time detection and alerting mechanisms, reducing reliance on external tools and minimizing latency. This integration would ensure that we can maintain robust and responsive security measures across our extensive 5G network, safeguarding it against potential threats more efficiently.

Chapter 7

Measurements

In this chapter we will evaluate the performance of Tetragon while it is deployed inside our testing cluster. We wanted to compare the performance of different eBPF based tools, measuring their impact on the system they are installed on while they are running and a heavier workload is being executed on the tested node.

7.1 Benchmark

To ensure a common testing environment we used the same node for all of our tests and made no changes to the cluster or the underlying system in between our different tests. For the actual testing we used two different tools, one for gathering information on the system load and one for generating a heavy workload. We used *sar* for gathering the information and *stress-ng* for the workload, simulating around 20000 process forks every second, saturating one out of the 8 cores of the CPU.

7.1.1 Tables

Starting with our first test we executed the benchmark under three different conditions: our system baseline resource consumption, while running the benchmark without Tetragon and lastly executing the benchmark with Tetragon running.

The other two eBPF tools that we have chosen for these tests are similar products that have achieved a similar popularity as Tetragon: KubeArmor and Tracee. We have repeated the same tests for both of them, gathering the following results.

Tetragon	KubeArmor	Tracee
91.63	98.36	100.00
95.52	96.59	100.00
94.02	98.35	100.00
86.30	98.23	100.00
88.21	98.61	99.88
93.87	97.00	99.89
90.66	98.49	100.00
93.42	98.11	99.88
92.18	98.24	100.00
93.24	97.10	100.00
93.16	98.60	100.00
91.39	94.87	99.88
87.40	97.85	100.00
88.48	98.35	100.00
92.68	97.47	99.88
93.58	97.73	100.00
93.92	98.48	100.00
92.86	96.98	100.00
90.41	98.74	100.00
93.89	97.97	99.88
88.82	98.61	100.00
94.04	97.58	100.00
92.26	98.35	99.90
88.49	97.37	100.00
95.35	97.98	100.00
94.25	98.61	99.88
91.06	98.49	99.88
92.37	98.23	100.00
85.17	97.98	99.88
91.98	98.00	100.00

Table 7.1: Idle benchmark results

	Tetragon	KubeArmor	Tracee
	83.29	85.64	87.8
	78.21	85.61	87.04
	77.71	84.99	87.47
	81.27	85.62	87.12
	79.47	86.13	86.9
	82.54	85.34	87.01
	84.32	85.55	87.04
	82.44	85.48	87.47
	83.52	85.64	86.61
	81.91	85.71	86.25
	82.01	85.77	85.48
	77.99	85.23	86.5
	79.44	84.66	86.81
	82.17	84.91	87.44
	81.86	85.93	86.59
	82.41	85.05	87.47
	83.33	85.57	86.43
	83.16	85.82	86.68
	83.06	85.30	87.12
	82.81	84.73	87.15
	79.62	86.17	86.68
	82.99	86.04	87.14
	83.38	84.88	87.17
	80.15	84.46	86.79
	81.74	86.33	86.72
	82.89	87.26	86.59
	84.04	90.04	87.26
	64.01	85.88	87.23
	80.51	85.39	86.7
	83.61	84.99	87.14
Delta:	9.65	12.52	13.14

Table 7.2: Benchmark running with eBPF tool not installed results (Delta compared to table 7.1)

	Tetragon	KubeArmor	Tracee
	76.99	82.61	81.61
	59.44	84.65	81.55
	77.27	85.25	81.63
	78.43	85.70	81.82
	78.02	84.54	81.90
	74.71	84.05	82.41
	78.18	83.48	82.99
	76.53	85.08	82.35
	78.47	84.99	81.12
	75.45	83.82	82.84
	78.31	84.08	82.82
	78.08	85.41	83.20
	75.92	84.78	81.30
	73.09	84.70	82.65
	77.52	85.03	81.95
	78.82	84.20	81.76
	79.67	83.63	82.82
	78.63	85.04	81.45
	78.90	85.61	83.01
	76.09	85.41	82.07
	75.42	83.23	82.24
	75.29	85.39	81.95
	77.59	84.89	82.69
	76.06	85.41	82.45
	76.58	86.10	82.65
	78.51	84.18	82.16
	80.08	83.63	81.95
	77.86	84.28	82.82
	78.79	85.51	82.71
	79.30	85.15	82.05
Delta:	5.04	2.09	4.78

Table 7.3: Benchmark running with eBPF tools installed and running (Delta compared to table 7.2)

After extensive research we have managed to evaluate only two of the three eBPF tools we have chosen. KubeArmor was not compatible with our benchmark that tests the performance of the system outside of the Kubernetes environment which KubeArmor works. Tetragon and Tracee on the other hand have managed similar results, with a slight advantage in terms of resource usage for Tetragon.

Despite not being able to filter events related to our benchmark using KubeArmor we have observed a relatively high resource usage after installing it on our system. This could mean that it is less efficient than Tetragon or Tracee during more intensive workloads where KubeArmor would have to also gather thousands of events every second.

Chapter 8

Conclusions

With this thesis, we have successfully evaluated the feasibility of using Tetragon in a large 5G network cluster. Our findings indicate that Tetragon is indeed well-suited for monitoring our cluster. Despite an initial steep learning curve, Tetragon eventually offers simple usability and precise customizability. Our initial efforts at monitoring system calls are just the beginning of what can be achieved with Tetragon, especially with a deeper understanding of the Linux kernel and its inner workings. So far, we have tested Tetragon with basic system calls, integrated it into our cluster, and organized the gathered data into a centralized dashboard for easier visualization. Based on our work, Tetragon appears to be a viable tool for the observability of a large cluster, with the potential to expand and be tailored to meet more of our specific needs.

8.1 Future plans

While Tetragon can be excellent when dealing with use cases such as file integrity monitoring, there are still plenty of cases where we would need further support from other tools and software. One of such cases is the detection of malicious access to our system, which falls short with the usage of just eBPF and Grafana. The complexity and scale of modern networks, especially in 5G environments, necessitate a more robust and nuanced approach to security monitoring. Using tools that can rely on machine learning to better analyze all the data we gather from events can help us more clearly identify which actions are malicious and distinguish them with certainty from regular use, especially in clusters of thousands of machines where automation is the only solution. Machine learning algorithms can detect patterns and anomalies that are not immediately obvious to human analysts, providing a higher level of insight and accuracy in threat detection. Integrating such advanced analytical tools with Tetragon can enhance our ability to preemptively respond to threats, reduce false positives, and ensure the integrity and security of our 5G networks. Furthermore, the scalability of machine learning models makes them particularly well-suited for the dynamic and expansive nature of 5G deployments, where real-time data processing and decision-making are crucial.

Integrating machine learning with Tetragon requires further analysis to assess its feasibility and business viability. However, this integration could significantly enhance our ability to intercept malicious access attempts in our cluster, making it a highly valuable future project.

Bibliography

[1] Kubernetes. *Kubernetes: Production-Grade Container Orchestration.* URL: https://kubernetes.io/docs/concepts/overview/ (cit. on p. 5).

[2] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. «Borg, Omega, and Kubernetes». In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444 (cit. on p. 5).

[3] Chandler Harris. *Microservices vs. monolithic architecture.* URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (cit. on p. 6).

[4] Avinetworks. *Kubernetes Architecture.* URL: https://avinetworks.com/glossary/kubernetes-architecture/ (cit. on p. 7).

[5] Cilium. *eBPF-based Networking, Observability, Security.* URL: https://cilium.io/ (cit. on p. 13).

[6] Flannel-io. *Network fabric for containers, designed for Kubernetes.* URL: https://github.com/flannel-io/flannel (cit. on p. 13).

[7] Weaveworks. *Weave Net - Weaving Containers into Applications.* URL: https://github.com/weaveworks/weave (cit. on p. 13).

[8] Kubernetes. *Using RBAC Authorization.* URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (cit. on p. 15).

[9] Kubernetes. *Autoscaling Workloads.* URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/ (cit. on p. 16).

[10] eBPF. *eBPF Documentation.* URL: https://ebpf.io/what-is-ebpf/ (cit. on pp. 18–21, 23, 24).

[11] Aqua. *eBPF Linux: How It Works, Use Cases and Best Practices.* URL: https://www.aquasec.com/cloud-native-academy/devsecops/ebpf-linux/ (cit. on p. 19).

[12] Grafana Labs. *Dashboard anything. Observe everything.* URL: https://grafana.com/ (cit. on p. 29).

[13] Grafana Labs. *Grafana Loki.* URL: `https://grafana.com/oss/loki/` (cit. on p. 31).

[14] Grafana Labs. *Promtail agent.* URL: `https://grafana.com/docs/loki/latest/send-data/promtail/` (cit. on p. 33).

[15] Julius Volz and Björn Rabenstein. *Prometheus: Monitoring at SoundCloud.* 2015. URL: `https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud` (cit. on p. 35).

[16] Prometheus. *From metrics to insight.* URL: `https://prometheus.io/` (cit. on pp. 35, 36).

[17] Tetragon. *eBPF-based Security Observability and Runtime Enforcement.* URL: `https://tetragon.io/` (cit. on p. 39).

[18] Tetragon. *Tetragon gRPC API.* URL: `https://tetragon.io/docs/reference/grpc-api/` (cit. on p. 44).

[19] Kornilios Kourtis and Anastasios Papagiannis. *File Monitoring with eBPF and Tetragon.* 2024. URL: `https://isovalent.com/blog/post/file-monitoring-with-ebpf-and-tetragon-part-1/` (cit. on p. 48).

[20] Mimi Zohar. «LSS-EU 2018: Overview and Recent Developments Linux Integrity Subsystem». In: (2017), p. 29. URL: `https://events19.linuxfoundation.org/wp-content/uploads/2017/12/LSS2018-EU-LinuxIntegrityOverview_Mimi-Zohar.pdf` (cit. on p. 49).