Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Kubernetes Near Real-Time Monitoring and Secure Network Architectures

Miguel Ángel López Muñoz

Supervisor: Dr. Stefan Weber

A dissertation submitted in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science – Data Science

Submitted to the University of Dublin, Trinity College, August 2022

Declaration

I, Miguel Ángel López Muñoz, declare that the work described in this dissertation is, except where otherwise stated, entirely my own work, and has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed: Miguel Ángel López Muñoz Date: August 18, 2022

"Well, I'm going to have myself a little nap.
The only thing to do, really, after a nice toast."

- Siegward of Catarina

Abstract

The popularity of cloud deployments for applications has made Kubernetes the largest container orchestration system. Given the many benefits and functionalities it features, its growth is not showing any signs of stopping.

This en masse adoption, however, has put Kubernetes' security flaws on the spotlight, as most of its industrial deployments have suffered from security issues, ranging from architectural vulnerabilities and insecure default configurations native to Kubernetes, to user misconfigurations. As a consequence, multimillion dollar losses and several Gigabytes of sensitive user data were stolen as a result of attacks.

Unfortunately, there is a lack of precise, comprehensive and up to date documentation pertaining to Kubernetes security, compiling risks associated with vulnerabilities, how to exploit them, how to configure a cluster to be as secure as possible and security best practices. This deficit has led to a lack of skills on how to design secure Kubernetes architectures. Furthermore, there are very few open-source monitoring tools, capable of detecting attacks and educating users on attack vectors. The combination of both would help users understand in detail how a secure Kubernetes cluster is constructed, from development to deployment, while allowing them to explore and visualize simulated or real attacks.

This dissertation's aims are twofold. First, to describe in detail the most crucial configuration pitfalls, the best practices to follow and the common attacks that can be conducted against a Kubernetes cluster. Second, to develop a near real time, scalable and fault-tolerant monitoring tool, that serves as a baseline for industrial Kubernetes deployments. This tool features concepts such as Stream processing, fast indexing and representation using Kafka, Elasticsearch and Kibana.

Both of these goals were achieved in this dissertation, nonetheless, there are improvements (TLS implementation, Kafka topic/broker replication or Firewalling), studied in the future work section of this dissertation, that should be made to transform the monitoring tool from a prototype to a fully functional industrial design.

Acknowledgements

A Mis Padres por todo el apoyo, el cariño, y el tiempo a mi lado, sin vosotros esto no hubiera sido posible.

A Mamá, por estar a mi lado en cada momento del desarrollo de la tesis, por ayudarme a encontrar la fuerza para seguir, aunque pensase que no podía más, por calmarme en los momentos que estaba al límite, por todas las ideas, por enseñarme a ser perfeccionista hasta el último detalle y por enseñarme como debería ser un ingeniero. Por haberme enseñado a ser un roble.

A Papá, por enseñarme a mirar hacia una meta, por aguantarme en las sobremesas, por darme siempre ánimo, por enseñarme a dar lo máximo de ti mismo para la gente a la que quiero, por enseñarme a que nada se acaba hasta que realmente termina.

To Stefan, without your help and guidance this would have not been possible.

Nicorentzat, sostengu, adore, ulermen eta alai aldi guztiengatik, baita urrun egon ere y por ayudarme a mantenerme a flote en uno de los momentos más duros de mi vida.

A Dani, por venir a verme dos veces, por todos estos años, por hacerme tan cercano un mar de distancia y por la comprensión, el cariño y la paciencia en un año así.

A Alba, por hacerme compañía las madrugadas de trabajo, a Joa por ayudarme a despejarme tanto de curro como de preocupaciones casi cada tarde, a ambos por tantas noches de hacerme sentir como si no me hubiera ido, y por venir a verme junto con Dani en lo que fue una de las mejores semanas en años.

Contents

List of Figures

1 Introduction

Kubernetes [1] is an open-source container orchestration system, capable of automating, managing and scaling deployments of containerized applications in local, distributed or cloud environments. According to Gartner, more than 75% of global companies will have containerized applications in production by the end of the year 2022, compared with the 30% of 2020 [2]. With this context, it is no surprise that Kubernetes keeps growing in popularity.

Despite the benefits that come from using Kubernetes, building a secure Kubernetes cluster is not easy, and many of the companies that use this orchestration system have faced security issues. In a survey performed by RedHat[1] 93% of the users asked, encountered security issues, either related to misconfigurations, exposed vulnerabilities, or attacks. Of these issues, 95% were the result of human error, and mostly linked to bad practices and misconfigurations [3]. Misconfiguring a Kubernetes cluster or leaving vulnerabilities exposed leaves the system open to very common attacks. Some of these attacks are:

- Exploiting vulnerable components that are accessible from the public Internet.
- Stealing and scraping of credentials.
- Using the resources of compromised pods to mine crypto for the attacker.
- Moving sideways from a compromised pod to the rest of the Kubernetes components.
- Concealing the signals of attacks.

It is important to note that implementing security in Kubernetes is very different from implementing security on a traditional system, even if the concepts applied for both systems are the same. There are two major pillars at the disposal of Kubernetes users to prevent these attacks from happening, or to stop them if they happen: best practices and security tools.

[1] https://www.redhat.com/en

First of all, best practices help configuring the cluster in the most secure way, thus minimizing the rest of vulnerabilities exposed above. Some of the most relevant best practices are:

- Preventing components from being accessed from the public Internet, if possible.
- Storing credentials ciphered in a dedicated location.
- Monitoring the activity of components to detect malicious code running in them.
- Isolating components from each other.
- Giving components the minimum privileges to perform their tasks.

Unfortunately, documents on best practices and secure configurations are lackluster and incomplete in the Kubernetes website. In addition, there are known vulnerabilities in many of the Kubernetes default configuration options that users are not properly warned about. Adding up to this, Kubernetes offers multiple options for configuring pods, services and nodes. The combination of which results in near infinite cluster configurations. On top of this, single or combination of multiple specific configurations can lead to vulnerabilities or possible vulnerabilities. Furthermore, Kubernetes is constantly changing and evolving, making it hard to keep up with new vulnerabilities. Most of the information regarding vulnerability exploits and their defenses is fragmented across the Internet and not always is up to date.

Regarding open-source security tools, they help users comply with the best practices and enhance the overall security of the cluster. Some security tools called static, are designed for their use before the deployment, to check for vulnerabilities, perform tests, and audit the overall security of the cluster configuration, while others called dynamic, perform on the running cluster, monitoring network traffic in real time, checking logs and alerting when attack signatures are detected. Despite their usefulness, there are very few open-source time monitoring security tools. As an example, there is a monitoring tool prototype made in the thesis master, of student Mr. Karode [4], that shows the TCP traffic in a Kubernetes of a guestbook application.

The main consequences of security issues in industrial Kubernetes deployments have ranged from delays in deployments in 55% of cases [3] , to revenue losses in 31% of cases. Some of the most high profile examples in corporations are:

CAPITAL ONE: This banking company was attacked in 2019 as a result of a Kubernetes misconfiguration [5]. As a result, the credit information of around 106 million users was exfiltrated.

FANCY BEAR: the hacker group known as APT-28 or Fancy Bear has, according to the NSA, launched brute force attacks to Kubernetes networks in order to steal credentials allowing them compromise company networks, between 2021 and 2022 [6].

CRYPTOMINING: There are many examples of Kubernetes vulnerabilities being exploited in recent years to cryptojack containers [7]. Dockerhub had several infected images that, if installed, would use the user's resources to mine cryptocurrencies for the attacker. In a similar manner a misconfiguration in the Kubernetes dashboard led to Tesla's whole Kubernetes deployment to be used for cryptocurrency mining. Microsoft Azure also suffered their fair share of cryptojacking attacks as a result of using infected images.

1.1. Motivation

Since the adoption of Kubernetes in industrial environments, the great majority of deployments have suffered from security issues as a result of the general lack of knowledge on secure configurations and attack vectors. While the concepts behind traditional security measures are valid on Kubernetes, it is no surprise that their implementations are not fit to apply in this context due to the structure of Kubernetes networks and the heterogeneity of deployments. Moreover, Kubernetes offers very little in terms of documentation regarding secure configurations. On top of this, while there are many documents pertaining to Kubernetes attacks, network security, and best practices, there are no overviews pertaining to all the aforementioned. Finally, while Kubernetes attacks have similarities to attacks on classic networks, there are very few real time open-source monitoring tools available to detect them.

Considering these problems, there are two main research gaps this dissertation will try to address:

- The general lack of knowledge on how to develop, deploy and run a Kubernetes cluster in a secure manner.
- The scarcity of open-source near-real time monitoring tools in Kubernetes that gives responses in less than one second.

1.2. Research Questions

After getting a glimpse at the current state of the Kubernetes' network security, the state-of-the-art solutions, the vulnerabilities and the possible consequences of such risks being exploited, (concepts described in detail in the following chapters), the following research questions are formulated:

- What is the current state of Kubernetes security?
- Which design would be more fit to a distributed and highly scalable paradigm?
- Would it be possible to analyze traffic in near-real time with a response time less that one second using stream processing tools as Kafka?
- How would it be possible to standardize this system in order for it to be deployed in any container-based application?

1.3. Objectives

The main goals of this dissertation are:

- Provide a general and up to date guide on Kubernetes security overview on security concepts, attacks, vulnerabilities, best practices, and security tools, emphasizing in network security.

- Develop a security tool/architecture prototype capable of monitoring a Kubernetes cluster in real time, serving as a baseline design for a tool that could be deployed in industrial/real-world environments.
- Develop a security tool that serves as an educational tool to identify network attacks and suspicious traffic on Kubernetes.

The specific goals are:

- Discuss the state of the art of security in Kubernetes.
- Make a brief overview on the most relevant Kubernetes attacks and how to defend against them.
- Survey security implications of Kubernetes, both design and configuration wise.
- Convey a comprehensive list of best practices and the implications of not following them.

On the other hand, relating about the near-time monitoring tool the specific goals are:

- Redesign Karode's architecture [4] to implement real time data streaming data pipelines instead of using HTTP and secure it.
- Implement a store and a dashboard system compatible with the new system architecture.
- Improve dashboard capabilities in order to better understand and visualize packet data.

1.4. Contributions

In this dissertation there are two main groups of contributions. The first group pertains to over viewing Kubernetes' security implications, while the second is related to the design of an open source, real-time monitoring tool.

Contributions overviewing Kubernetes security:

- Summarize Kubernetes' known problems in terms of network security.
- Briefly describe known attack vectors and defenses.

- Aggregate known best practices for Kubernetes security.
- Analyze the security problems and advantages of the proposed architectures.
- Identify the changes in configuration and further design requirements if this system were to be implemented in a real life industrial scenario.

In terms of the design of a Kubernetes monitoring tool, the S. Karode's monitoring tool was taken as the starting point to design and build a new architecture oriented to stream processing with Kafka pipelines instead of Karode's HTTP pipelines. Meaning that the sniffer code, container configuration files, images, had to be extensively modified. With the following advantages:

- Improvement in efficiency of data transferring, resulting in near-real time monitoring through stream processing and fast data indexing using Kafka and the Elastic stack.
- Easier deployment and configuration of the monitoring system, as internal IP's of the Kubernetes components are not needed to be known.
- Better scalability, as Kafka allows for broker replication, offers higher throughput and lower latency.
- Improvements in security, as Kafka, first implements TLS and second allows for the cluster to have fewer ports open.
- The storage and dashboard components were re-designed, using Kafka, which are able to handle the publish subscribe paradigm, instead of using HTTP POST and GET, thus avoiding encountering a bottleneck if the Karode's dashboard were to be used.
- Elasticsearch data indexing engine along with Kibana allow for complex, precise, and highly customizable data representation graphs to be designed in a fast way through a visual interface, without the need to modify or write code.
- Kibana features a functionality that can automatically alert users if a previously defined condition is met, said conditions can be configured to fit known attack signatures present in network packets.

1.5. Dissertation Structure

This thesis is divided in the following chapters:

Chapter 1: Introduction. This chapter explains the importance of the security issues in Kubernetes and the motivation to address them. As a result, a set of research questions are formulated, to which this work provides answers. Finally, the objectives of the dissertation and the most relevant contributions are described.

Chapter 2: Background. This section describes Kubernetes concepts, such as its components and networking, with special detail to components related to security. Additionally, it introduces relevant topics required to understand the design of the security tool developed, such as Kibana, Elasticsearch or Kafka.

Chapter 3: State of the Art of Kubernetes Security. This chapter describes in detail the State of the art of Kubernetes security, including security tools, best practices and known vulnerabilities, their fixes and exploits.

Chapter 4: Near Real-Time Scalable and Fault-tolerant Monitoring Tool. Describes the end-to-end development (requirements, design, implementation and test) of a monitoring system for Kubernetes clusters.

Chapter 5: Conclusions and Future Work. Finally, this chapter describes the most relevant conclusions drawn during the development and writing of this dissertation. It also describes the future endeavors that should be followed to further develop the work undertaken in this dissertation

Appendix A 1. Additionally, the appendix features Python source code, YAML configuration files, corresponding to the programming of the components of the developed monitoring system.

2 Background

This section consists in the review of literature pertaining to the topics treated in this dissertation. Said topics being: Microservice and container architectures, Kubernetes, security implications and solutions in distributed container-based networks, attack vectors in Kubernetes networks, (dubious topics) stream processing paradigm in Kafka, scalability in distributed systems, and machine learning driven security systems.

2.1. Microservices and Container Architectures

Microservice Architecture (MA) is an architecture style characterized for dividing large applications in a sum of loosely coupled simple services, called microservices, that cooperate using asynchronous messaging communication [8] [9]. This architectural style allows for each microservice belonging to a system to be managed, updated, developed and deployed independently, simplifying maintainability and providing scalability [10] [11] [12] [13].

Microservice Architecture was born in 2011 as a variant of the Service Oriented Architectural style (SOA) [14]. In recent years, MA has gained popularity, becoming the template for cloud native apps, as a way to deploy, scale, and replicate them easily. Companies such as Netflix, Uber, Spotify or Soundcloud have used microservices architectures extensively.

In spite of the latter, microservice architectures have inherited the security problems of the distributed systems and SOA architectures preceding them. Microservices require secure communication between each other, as they present larger surface of attacks, increasingly complex networks, trust requirements, heterogeneity, lack of a global system enforcing security, etc. [10].

In order to package microservices and deploy them in the cloud, containers are preferred to virtual machines as they take less space and ensure that they run the same, regardless of the infrastructure [15] [16] . A container is a lightweight executable made of a software package with its application code, configuration files, and operating system libraries and dependencies. Docker [17] has become the de facto open containerization technology. Containers are available for both Linux and Windows based applications and can run across many platforms or the cloud [18] [19].

2.2. Kubernetes Architecture

To automatically deploy, configure and manage containers, a container orchestration system, such as Kubernetes is needed [20]. “Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery” [1].

The main benefit of combining containerized microservices and orchestration software is the deployment of applications consistently wherever needed over distributed architectures or hybrid cloud architectures [16].

When Kubernetes is deployed a cluster is created. In the architecture of Kubernetes (Figure 1) a cluster is composed of worker machines called worker nodes. Every cluster has at least one node. These nodes are in charge of running containerized applications and hosting pods. A pod represents a set of running containers in the cluster. The control plane is in charge of managing worker nodes and pods in a cluster. Control plane is a container orchestration layer, tasked with exposing the Kubernetes API, whose purpose is to define, implement and manage the life cycle of containers.

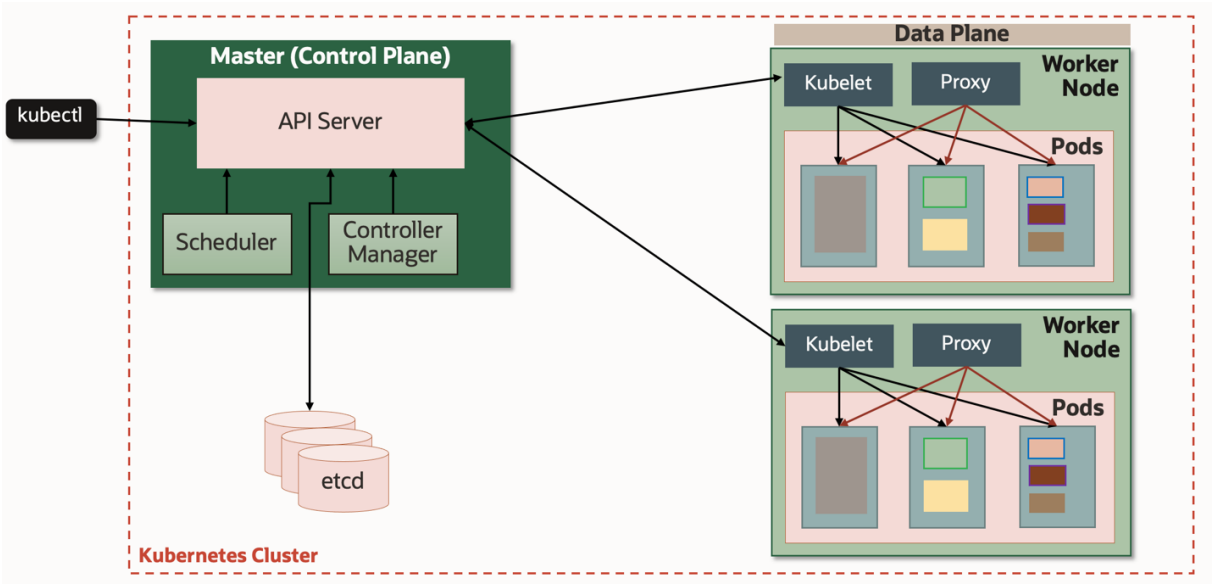

Figure 1: Architecture of a Kubernetes Cluster (Source: thenewstack.io)

2.3. Software Defined Networks

Kubernetes networks are Software Defined Networks (SDN), while they do not work in the exact same manner as typical SDNs, an introduction to them will be very useful to further understand Kubernetes networks.

Traditional networks are composed of a control plane that decides how traffic is handled and a data plane, where the forwarding of the traffic happens. Both planes are tightly coupled, which means introducing new elements or functionalities to the network, such as firewalls, Intrusion Detection Systems (IDS), load balancers, require modifying the network structure, which in turns affects the logic in the control plane.

Software Defined Networks (SDN) in turn, are programmable networks, in which control plane and data plane are separated (Figure 2). This is done by introducing a controller, which is a software tasked with making decisions over how traffic is handled. As a result, hardware devices such as switches and routers roles become to forward packets according to the controller orders. This paradigm results in SDNs being easier to manage and to modify [21].

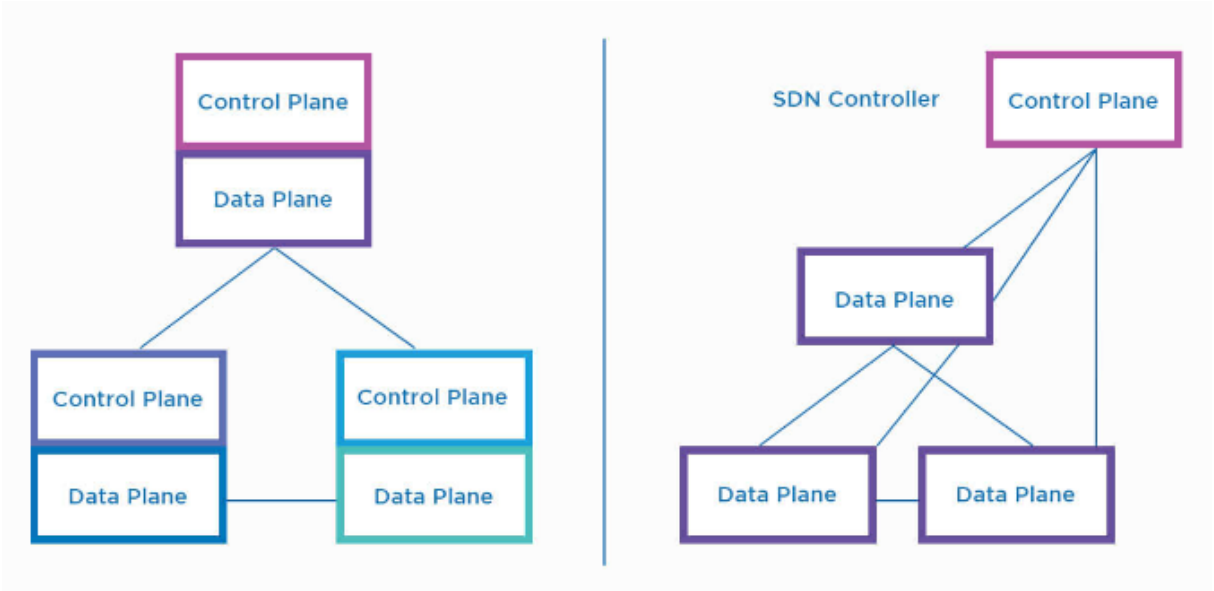

Figure 2: Traditional Networking VS SDN (Source: FileCloud Blog)

2.4. Kubernetes Components and Networking

As Kubernetes security is the main topic in this dissertation, it is necessary to introduce the concepts in which Kubernetes is based on, the basic Kubernetes' architectures, components, and functionalities and finally the general security concepts, best practices, and security tools in Kubernetes.

2.4.1 Components

In order to understand the following sections, a description of Kubernetes' components, networking, and configuration will be conducted [1].

Kubernetes cluster. When Kubernetes is deployed a cluster is created. A cluster is composed of worker machines called nodes. These nodes are in charge of running containerized applications and hosting pods. Every cluster has at least one node.

Pods. A pod represents a set of running containers in the cluster. A pod consists of one or more containers and is the most basic computing unit in Kubernetes [1]. There are two main ways in which a pod can operate: Pods containing one container, and Pods containing more than one container. In this second case, the containers in the pod are very tightly coupled, share resources and work together to form an application.

Pause Containers. Hosts the containers in a pod and saves network configuration, such as IP addresses, network endpoints, Network namespaces (netns) and so on for them. This way if a worker node crashes or restarts their network configuration will not be lost (note that this appears in windows for Kubernetes when it is used in Linux too)

Kubernetes service. A Kubernetes service is an abstraction that exposes an application running in a set of pods as a network service. A service gives pods their own IP's and a single DNS name to a group of pods. A service is also capable of load balancing between pods. The reason behind this is that pods are a non-permanent resource, this means they are constantly created and destroyed. This means the pods running at every given moment can be different from the pods running a moment before. Pods are constantly communicating and asking for resources in between each other, the service is in charge of indicating to the pods to which IP's to connect in order to obtain said resources. Services can be configured

to only redirect internal traffic, or traffic generated by pods inside the cluster, to endpoints in nodes within the cluster.

The control plane is in charge of managing worker nodes and pods in a cluster. Control plane is a container orchestration layer, tasked with exposing the API, in order to define, deploy and manage the lifecycle of containers.

Both control plane, and nodes are made up of a plethora of components (Figure 3).

Control plane is composed of the following components:

- **kube-apiserver**: Validates and configures data for the Kubernetes API objects. It also services REST operations and provides the frontend with the cluster's shared state. It acts as the frontend of the control plane, as it exposes the Kubernetes API.
- **etcd**: Consistent and highly available key value store containing all cluster data. Including the secrets.
- **kube-scheduler**: Assigns nodes to newly created pods
- **kube-controller-manager**: Runs controller processes. Those are control loops that check the current state of the cluster and make changes to drive the cluster to the desired state.
- **Admission Controller**: Code that intercept requests to the Kubernetes API server, before they reach the server, but after authentication and authorization. An admission controller limits requests to create, modify objects, and connect to the proxy. They define what should and should not be allowed in terms of creation and modification of a pod given the security context.
- **Cloud-controller-manager**: Embeds cloud specific control logic, allowing a cluster to be linked to the cloud. It also separates components that interact with the cloud from the ones that only work and communicate locally.

The Nodes are made up of the following components:

- **Kubelet**: Ensures that pods are running and healthy according to Pod Specifications which are YAML or JSON objects that describe pods. It registers nodes with the apiserver.

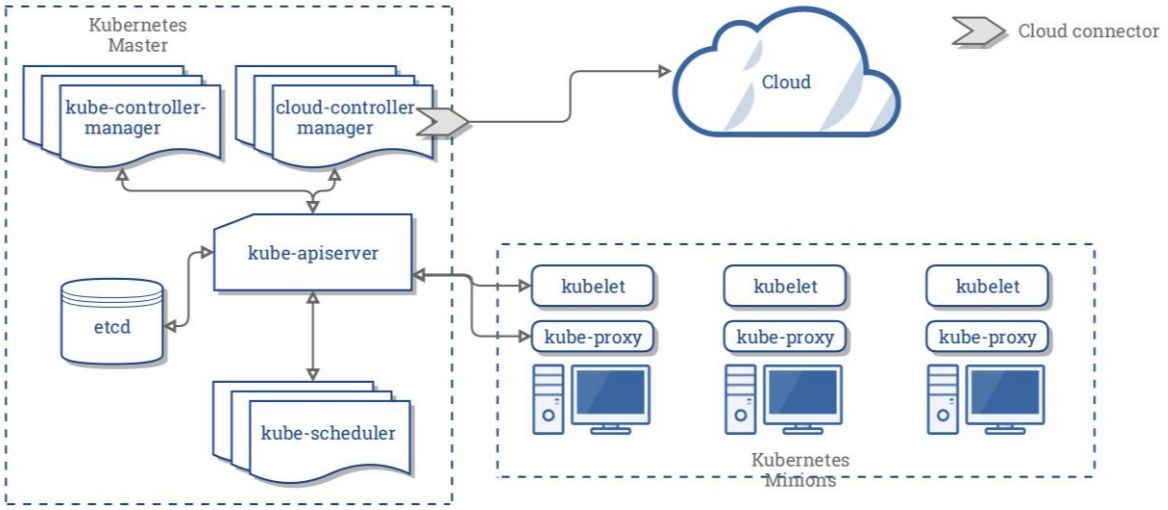

Figure 3: Kubernetes components (Source: Kubernetes)

- **Kube-proxy**: proxy running on each node of the cluster. It supports DNS with addons. It can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends

- **Container runtime**: Software responsible for running containers. It is a daemon that runs on each node, provides a Container runtime interface to the Kubelet and loads the plugins needed for the correct functioning of the node, such as the CNI.

There are some other components and plugins that, while they are not essential to run a Kubernetes cluster, are very commonly used:

- **CNI**: or container network interface is in charge of enforcing network policies. It is required if a user wants to modify the Kubernetes network model.

- **Core DNS**: addon that provides the Kubernetes cluster with a DNS service compliant with the Kubernetes specifications.

- **Ingress**: The ingress exposes services within a cluster via HTTP/S routes to the exterior. An ingress can be configured to give services externally reachable URLs, load balance, terminate SSL/TLS and offer virtual name hosting. In order to work properly an ingress controller is needed (Figure 4).

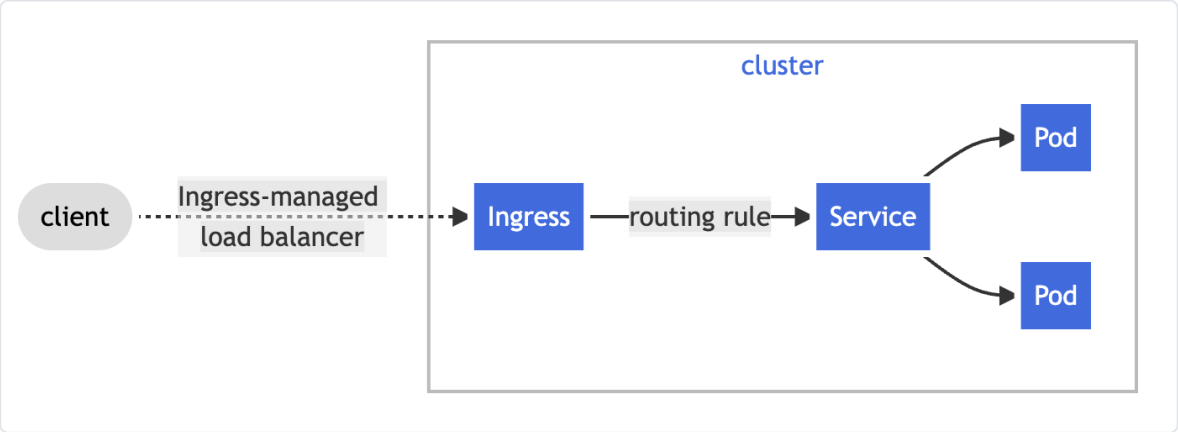

Figure 4: Ingress, traffic routing from client to services (Source: Kubernetes)

- **Ingress controller**: Ingress controller is in charge of load balancing and it might set up new frontends or reconfigure the edge router (Figure 4). Ingress controllers are not started when the cluster starts. Ingress controllers/ingress are third party tools, although they are necessary for the correct functioning of a cluster.

2.4.2 Networking

This subsection treats Kubernetes networking, both in cluster and networking towards the external internet (Figure 5).

As mentioned earlier, every pod in a cluster gets its own cluster wide IP address. Additionally, links between pods, port mapping between hosts and containers, load balancing, etc. are handled by the services.

Kubernetes, by default, sets up a network model in which pods can communicate with every other pod and node without the use of NAT. Agents such as daemons or the Kubelet can also communicate with every pod in the node without restriction.

Containers within a pod share IP, MAC addresses and ports. This means, first, containers are able to communicate between each other via local host and second, they must coordinate port usage.

DNS: Kubernetes DNS assigns DNS names to every service in the cluster, including the DNS server, and configures the Kubelets to instruct pods/containers to resolve DNS names using the DNS server's IP.

When a pod makes a DNS query without indicating a namespace, the response will only encapsulate DNS names in the same namespace. In addition, only services and pods are given DNS records, which can be either A or AAAA (both records point to IP addresses with the only difference that AAAA records point to IPv6 addresses). Finally, DNS configuration can be done on a pod-to-pod basis.

Now that the basic structure of a Kubernetes network has been explained, a more in depth explanation on pod communication via services will be conducted.

As mentioned, the default configuration for a Kubernetes network allows pods to see each other without Nat, and to reach each other localhost. If a proxy is set up, or if done locally, anybody can access every component of the cluster, for example, a pod using SSH or using curl to make queries to pod IP's.

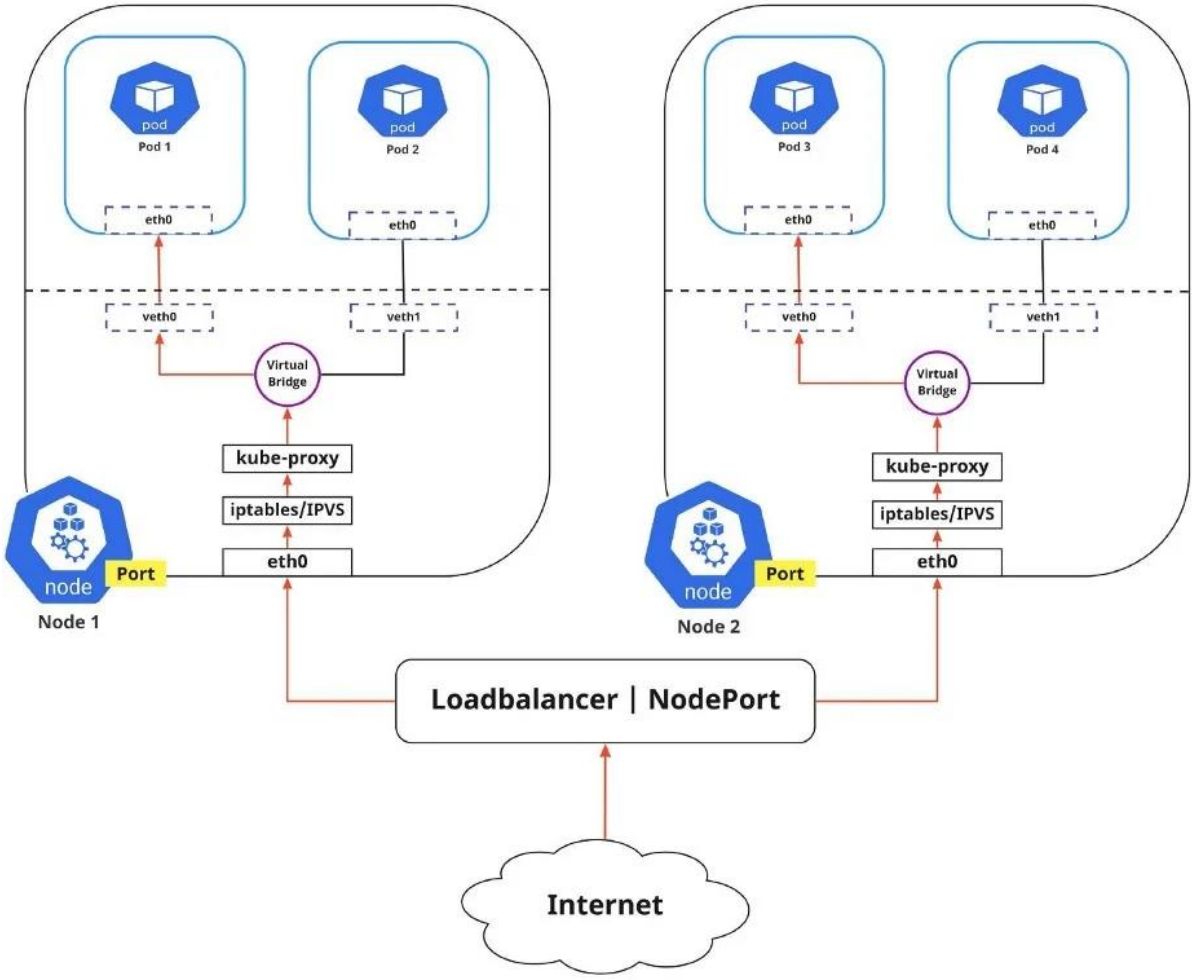

Figure 5: Kubernetes Networking Schema (Source: opensource.com)

Kubernetes gives the recommendation to use self-signed certificates to secure the communication if the service is exposed to the internet. In order to use certificates, the server must be indicated to do so in the YAML file and secrets must be set up in the etcd to allow pods to access the certificates.

Network Policies: In order to control traffic flow at the transport and network OSI layers Network policies can be implemented. Network policies are application-centered constructs that define how a pod is allowed to communicate with different network entities. The resources limited by a policy are defined by its type.

Network policies will not be followed by pods if there is not a CNI set up and configured to enforce network policies. Network policies are additive, this means that if two policies with the same type are defined for a pod, both will apply concurrently.

There are three main identifiers to determine with whom a pod can communicate with: a) Allowed pods (a pod must be allowed to communicate with itself), b) allowed namespaces and c) allowed IP blocks (traffic to and from the node where a Pod is running is always allowed).

By default, all outbound connections are allowed in a pod. One can restrict outbound connections in a pod by defining a network policy for a pod with the egress type. If there is such a policy, the only outbound connections allowed for that pod will be the ones detailed in said egress policy. The same applies for inbound connections, by default, they are allowed completely. Defining a network policy with ingress type will limit the allowed inbound connections to the ones detailed in the policy and the ones from inside the node.

In order for a pod to be able to make a connection to a destination, both ingress and egress policies must allow said connection, if one of the two does not, the connection will not happen.

Finally, network policies are incapable of:

- Forcing internal cluster traffic to go through a common gateway.
- Handling TLS in any way, shape or form.
- Targeting a specific node

- Targeting a specific service
- Creating default policies that would apply to all namespaces
- Logging security events
- Being formulated as, certain connections are denied the rest are allowed.
- Denying loopback/localhost traffic

2.5. Stream Processing with Kafka

Kafka is a distributed high-throughput, fault-tolerant event streaming platform developed in 2011 by LinkedIn and made an Apache open-source project [22] [23]. Event streaming is a kind of processing that captures data in near real-time from any event source [24] (i.e., transactions, databases, devices or software applications) and stores that data in the form of multiple event streams that can be processed in near real-time (< 100 ms) or with batch processing (more than one hour). Kafka is the worldwide de facto standard in stream processing used for streaming analytics (website user activity tracking), notifications (metrics and logging or commit logs) critical monitoring (cybersecurity attack detections in real time), or even to high-performance pipelines.

Kafka offers a publish/subscribe service to produce and consume events from different streams. In Kafka, each stream has a name, called topic, each event has a record <key, value> and there are two types of actors: producers and consumers (Figure 6). A producer (publisher) publishes a record in a topic, and a consumer (subscriber) that, after subscribing to that topic, consumes in FIFO order the records produced in the topic. The producer sets the key and value. The subscription offers pull delivery mode, that is, the consumer has to make a request to retrieve records from a topic. In a topic, there can be several producers and consumers simultaneously producing and consuming events.

Real time processing with partitions. In order to maximize the number of events processed by unit of time in a certain topic, Kafka offers the concept of topic partitions and consumer groups. The topic can be split in partitions, where all records of this topic with the same key must be published in the same partition. Consumers are organized into groups. Each group can have one or more consumers. Figure 7 shows an example of two consumer groups that

consume from topic A with four partitions. A consumer consumes from one or more topic partitions (set up by the consumer). For example, consider a topic where a producer publishes 1000 records/s, and a consumer takes 100 ms to process one record. With a group of one consumer, as a consumer processes 10 records/s, it will need 100 s to process 1000 records. Meanwhile, after 100 s of processing, there will be 1000 r/s*100 s records pending to be processed and this model will collapse. Therefore, it will be necessary to create a consumer group with more than 100 consumers.

Fault tolerant consuming. In order not to have information of a topic without processing, when a consumer of a group that was consuming from a partition topic crashes, Kafka service assigns the partition that belonged to the consumer who crashed to another consumer of the group. Figure 7 shows the consumer c2 of group 2 is consuming from partitions p2 and p3.

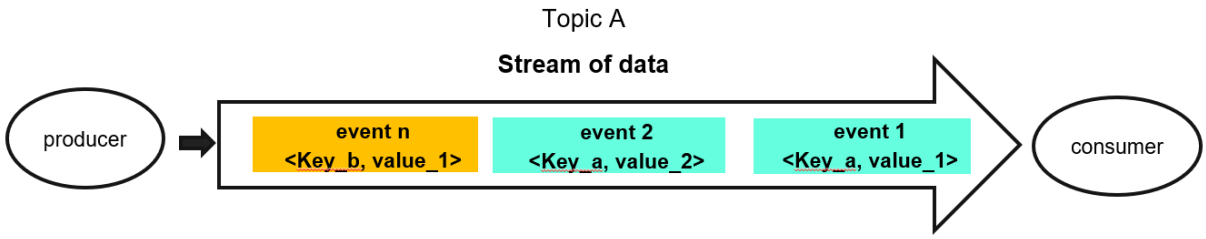

Figure 6: Stream of data events with two different keys published in a topic A.

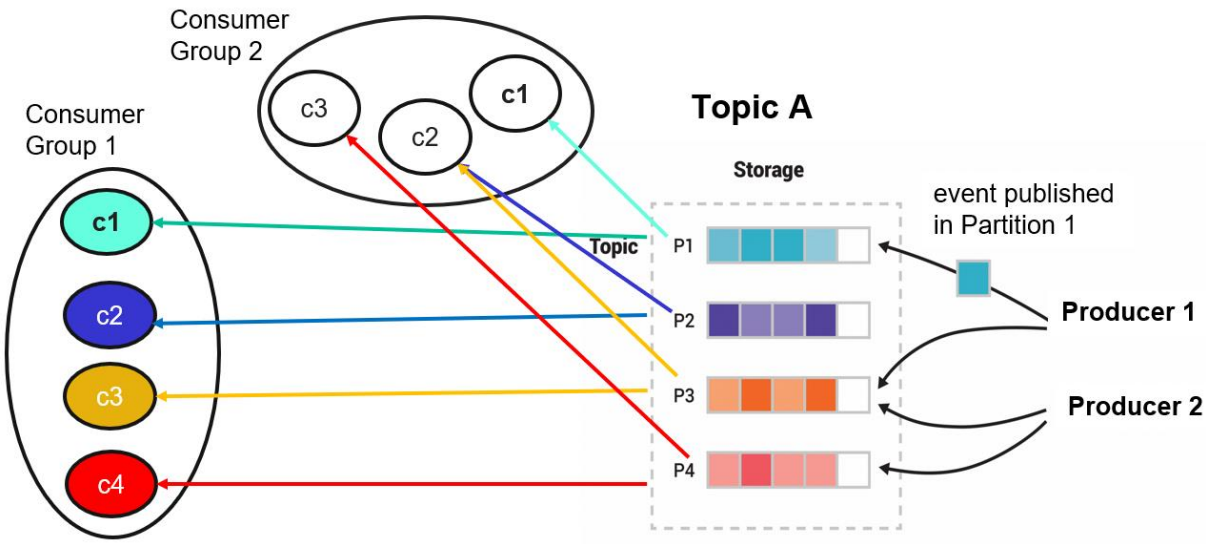

Figure 7: Concurrent consumption of the same records among different consumer groups.

Fault tolerant service. Kafka offers a fault tolerant service through the configuration option of broker replication. A broker is the process that supports the Kafka pub/sub service. With one broker configuration, if a Kafka broker crashes, the service stops. To configure a fault tolerant service that tolerates at most two failures, it is necessary to start at least three Kafka brokers. One broker acts as a leader and the others as followers. When the leader crashes, the Kafka service is temporarily stopped, then, there is a leader election among the running brokers using the Zookeeper service, to resume the Kafka service as soon as possible with a new leader.

Fault tolerant Topics. When the service is fault tolerant, it is also possible to replicate the topic (or their partitions) among the Kafka brokers. For example, a topic can be created with a replication factor of three. If Kafka has three brokers, one of the brokers will be the leader of the topic and the rest will be the followers. Each broker will maintain an updated copy of the topic. Every topic record to be published is received by the leader. Then, the leader forwards the record to the followers, and waits to have a majority of brokers (including itself) saying that they have stored it in their topic before answering to the publisher. This kind of replication is known as atomic consistency.

2.6. The Elastic Stack

The Elastic Stack (ELK stack) is a product of Elastic enterprise that comprises Elasticsearch, Kibana, Beats, and Logstash [25]. The ELK stack is able to collect data in any format and from any source, for searching, analyzing, and visualizing (Figure 8). Confluent has changed the Apache License of ELK stack to an Elastic License. This event resulted in Amazon Web Services introducing the OpenSearch project in 2021, a community-driven, open source fork of Elasticsearch and Kibana under Apache License [26]. The following description of the functionality of Elasticsearch can be also applied to OpenSearch.

Logstash and Beats collect and aggregate the data that will be stored in Elasticsearch. Elasticsearch is the distributed search and analytics engine, while Kibana offers dashboards to visualize, and analyze data.

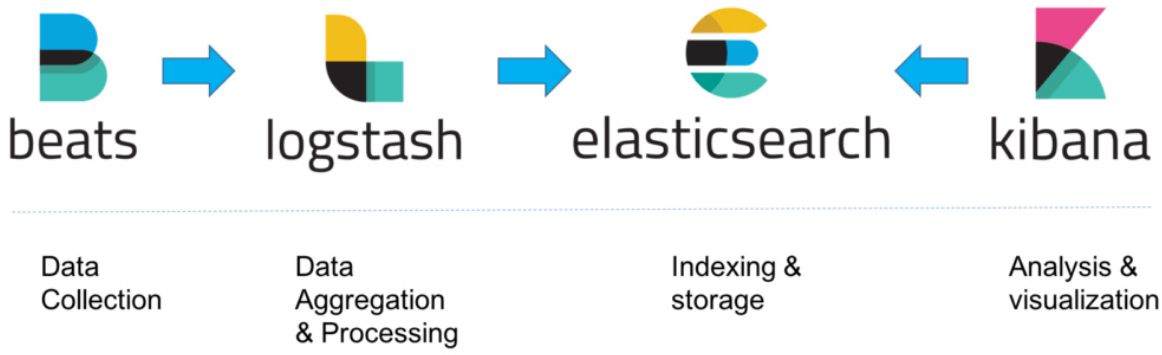

Figure 8: ELK Stack (Source Elastic [25]).

Elasticsearch is a RESTFUL scalable service for storing textual and non-textual data as JSON documents, and for near-real time data searching and data analytics. The Apache Lucene search engine inside Elasticsearch not only permits simple retrieval and aggregation of data, but also the discovery of trends or patterns in data. It is also a scalable service that maintains the response time of the data queries even when the data volume increases.

A document is indexed after storing it. An index can be seen as a collection of documents, where each document is a collection of key-value pairs containing the user data. Elasticsearch is very fast at rebuilding per-field data structures related to search results. Another interesting feature is that by default, Elasticsearch does not need schemas to define the document to be indexed. Elasticsearch map types found in the document to the Elasticsearch data types. However, it is possible to define explicit dynamic mapping when indexing and storing a document that contains types not detected by Elasticsearch.

Applications can access the Elasticsearch service through an Elasticsearch client available in Java, JavaScript, Go, .NET, PHP, Perl, Python or Ruby.

The main functionalities offered by Elasticsearch are:

Data Searching. The Elasticsearch REST API offers the Query DSL language for structured queries (similar to SQL queries) in a document, full text queries that return all documents that match the query sorted by the degree of matching, and complex queries that combine the two. Searching ranges from individual terms or phrases to similarities or prefixes.

Data Analyzing. Elasticsearch offers data aggregation for discovering key metrics, patterns, and trends. Aggregations are also very fast, enabling the analysis and visualization of data

in near-real time. When Kibana dashboards are used, Elasticsearch data aggregation is also updated in the dashboard in near-real time too.

Machine Learning. Elasticsearch offers machine learning features to automatically analyze time series data for detecting normal or unusual behavior of data or identify anomalous patterns in values, counts or frequencies.

3 State of the Art of Kubernetes Security

This section will introduce the concepts and tools required to properly understand this dissertation. Furthermore, an overview of security in Kubernetes will be made, allowing the reader to understand the current state of Kubernetes security.

3.1. Security in Kubernetes

This section describes the main aspects about the cybersecurity in Kubernetes [27] [28]. First of all, it compares the traditional security architectures in data networks with Kubernetes cluster networks, and it describes the main obstacles to achieve secure Kubernetes deployments in industrial environments. Afterwards the most common methods, and open-source tools used to overcome these obstacles are described.

Kubernetes has lately been adopted by many companies to deploy their projects in. “Gartner, the well-known US-based technology research and consulting organization, predicts that by the end of 2022, more than 75% of global companies will be running containerized applications in production” [2].

In addition, according to the report on the current trends of industrial Kubernetes made by RedHat security is the main concern [3], as at least 93% of the respondents experienced security incidents in the last year. These incidents are mostly the result of human error (95%). The most relevant is misconfiguration which was encountered by 53% of the consulted, 38% detected major vulnerabilities, 30% were faced with runtime security incidents, and lastly, 22% failed security audits. These incidents are not exclusive as the surveyed could select more than one incident.

The RedHat survey [3] shows optimistic results for the future security in Kubernetes, as at least 77% of the participants have already started using DevSecOps, of which, 27% have already integrated security and development fully, while 50% are starting to merge development and security.

DevOps can be defined as a set of software engineering principles, methods and practices applied to any software development life cycle and systems based on collaboration models between work teams (mainly development and operations) and stakeholders [29]. While DevSecOps is a DevOps practice about ”breaking the silos of security, giving that knowledge to the different teams, and ensuring that security is implemented at the right level and at right time” [30] [31].

The main conclusions that are drawn from the survey are the following:

1. There is a general lack of knowledge on best practices for Kubernetes
2. Most industrial users lack the knowledge to properly set up and configure Kubernetes in a secure manner.
3. Integration of security in all phases of the development cycle using DevSecOps is essential to prevent most of said incidents from happening, as pointed out by RedHat.

Problems described in 1) and 2) can be overcome by developing skills about security best-practices, security concepts and techniques in Kubernetes, and applying open-source tools in Kubernetes deployments.

3.1.1 Kubernetes Network Security vs Traditional Network Security Architectures

The main goal of secure network architectures is to strengthen network security by separating external facing services and resources, such as DNS servers, mail, and proxy, from internal resources and services [32] [33]. This way, internal services can only be accessed from the inside of the Local Access Network (LAN) or via a proxy. This goal is achieved by secure network architectures setting up Security Information and Event Management (SIEM) systems and firewalls [34] at the gates of the public Internet, that is, in a sub-network located in between the companies LAN and the Internet. This subnet is called Demilitarized Zone (DMZ). There are many examples of secure architectures in traditional hardware networks, such as Dragon1[2],

[2] https://www.dragon1.com/

Checkpoint[3], Citrix networks[4], or Cisco[5] in which firewalls protect limited access points to the network, with proxy servers set up to access internal services. These require a single or very few access points in which to locate the firewalls and SIEM systems default (Figure 9). Unfortunately, trying to apply this architectural design without any adaptation in Kubernetes networks would mostly be ineffective, as pods, containers and nodes are interconnected with each other by default.

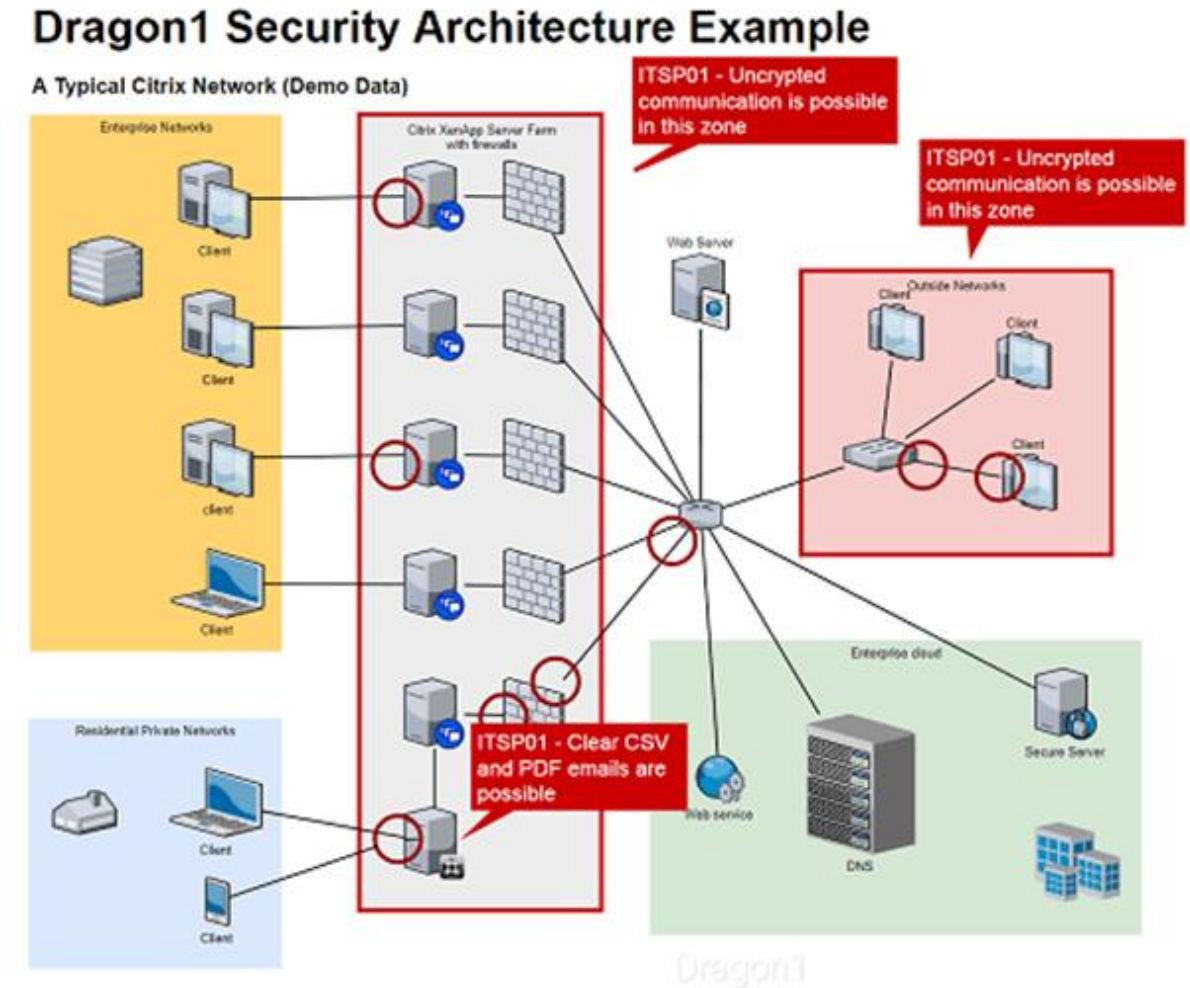

Figure 9: Example of Traditional Security Architecture (Source: Dragon1)

[3] https://www.checkpoint.com/

[4] https://www.citrix.com/

[5] https://www.cisco.com/

The aforementioned traditional security architectures (Figure 10 and Figure 11) provide segmentation among other security benefits [35]. Network segmentation is an architectural approach that consists in dividing a network in segments or subnets, which act as independent networks. The main benefit of segmentation is that smaller networks are easier to manage and control and design network policies according to their needs. Luckily, the concept of network segmentation is also applicable to K8's networks [36]. In order to provide network segmentation in Kubernetes, network policies are implemented. Network policies allow the administrator to determine how and with whom pods, nodes and clusters are permitted to communicate, thus reducing the attack surface the network has. Network policies require a Container Network Interface (CNI) to be enforced. CNI's are Kubernetes plugins configured with the network specifications that a worker node should use.

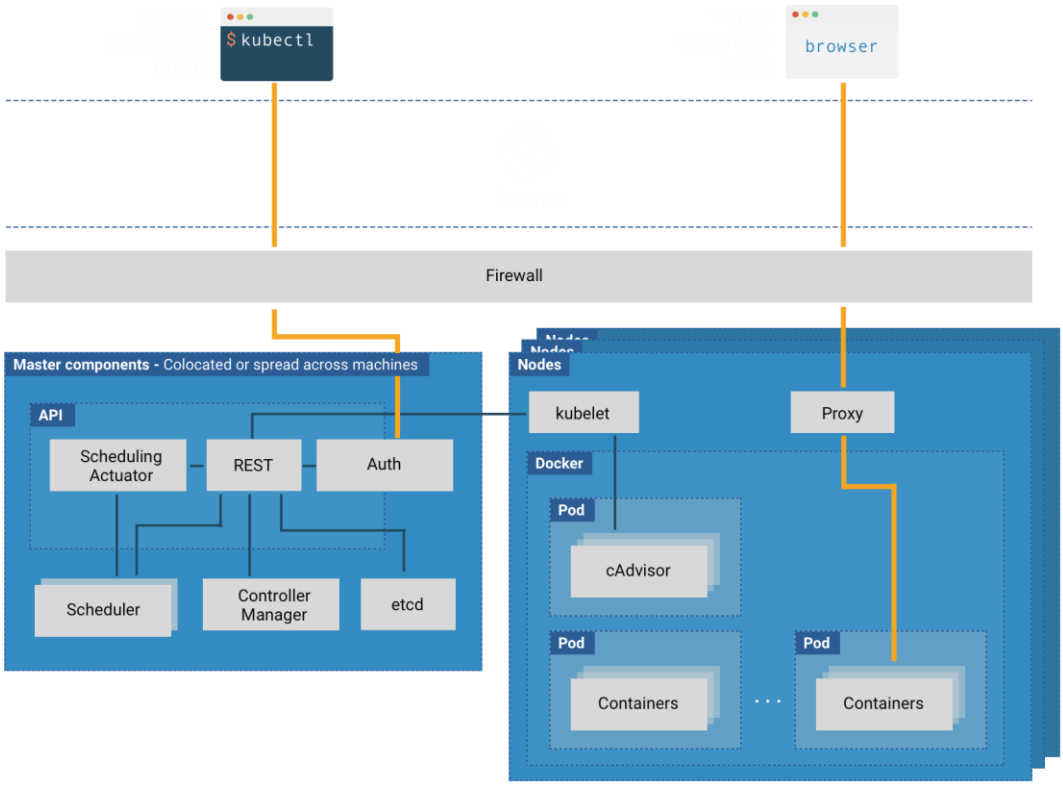

Figure 10: Secure infrastructure for Kubernetes clusters (Source: Giant Swarm)

Despite their usefulness, CNIs are not without pitfalls. First of all, CNI's are not standardized, as there are many available plugins that perform in different manners. Second, if a CNI was compromised or rendered inoperative, network policies would either not be enforced, or upright set up by the attacker. Third, CNIs do not disallow nor track nodes communicating over localhost, entailing the risk that if a node is compromised, an attacker would be able to freely move sideways to other nodes over localhost. Moreover, pods running in the same worker node are allowed to communicate freely over a virtual Ethernet bridge, incurring the risk that if a pod were compromised, an attacker would also be able to access every pod connected to that Ethernet bridge [36].

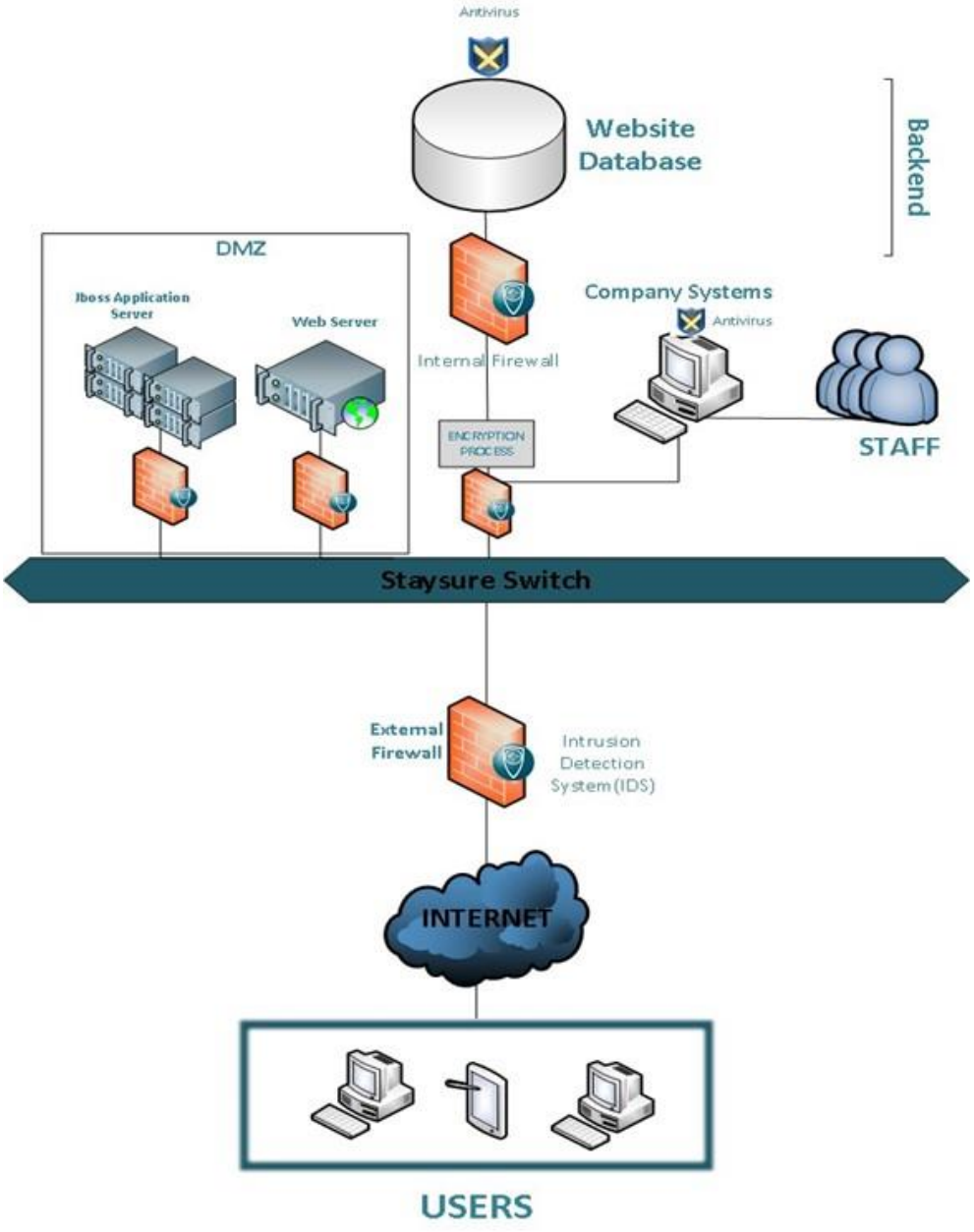

Figure 11: Basic high-level secure network architecture (Source: Right Turn Security)

3.1.2 Best Practices

In order to develop the most secure Kubernetes deployment possible, it is also important to not only use security tools, but to follow best practices in the development, deployment and runtime phases of the Kubernetes cluster [37] [38] [39]. Best practices have been defined for the general concept of IT security as "*human practice, security related, repeated or customary method, which has shown effectiveness by experience and is among the most effective practices that could be used in a particular scenario"* [40].

Kubernetes's best practices can be classified observing the area they pertain to as Cloud, Cluster, Container and Code best practices [41].

CLOUD BEST PRACTICES: Refer to the server infrastructure, and cover the following concepts:

- TLS access to Kube API server: Every connection used to access the API server should be done with TLS using secure ports. The API server must be provided with a TLS private key and a TLS certificate. Localhost ports to access the Kubernetes API should be disabled outside of the test configurations.

- Enforcing TLS between components: To prevent attacks on traffic, if communication is needed between the APIs in a cluster, the cluster should be deployed including the pertinent TLS certificates. Moreover, if any of the APIs are to be located on a public port, distinctions between trusted and untrusted connections should be made.

- Secret Encryption: A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Secrets must be stored encrypted in the etcd (key-value storage used by Kubernetes to store all its information data, state, metadata, etc.), reducing the attack surface of the cluster, as pods and container images do not contain said secrets. Pods and containers can access sensitive information they need via the etcd. The default setting, which keeps the secrets in plaintext, must be avoided. The key should be kept in the API server [42].

- Firewalling: Firewalling isolates pods and clusters, determining which ones can communicate with whom, and how. A Firewall must be set up between etcd and the API. Ports pertaining to API access, etcd access, Kubelet and master/worker nodes, should be

open, but not accessible by unauthorized entities. Moreover, distinct identities can be given to pods, allowing access through the etcd to only the secrets necessary for them to run properly.

- Isolating nodes from the exterior: Nodes should be isolated from public networks. In order to achieve this, control and data traffic must be separated from each other. Nodes should be configured with an ingress controller, so control plane traffic only operates on a specified port.

- Eliminating single points of failure: In case of the aforementioned CNI or the master nodes being compromised the Kubernetes cluster can be either stripped of any network security policies implemented beforehand or rendered unconfigurable. To avoid the latter, replication should be set up.

CLUSTER BEST PRACTICES:

- Credentials should be stored outside images and pods: Secrets should only be stored in a well-known place, in the Kubernetes case, the etcd. An image containing secrets could be attacked, resulting in the theft of the secrets, this risk is accentuated for publicly accessible containers/apps.

- Follow least privilege principle/Set up role-based access control: Authorization based on identity and logging should be required. This means each user should only be able to access the essential resources with the least privilege possible to perform their task. This is done by enabling Role Based Access Control (RBAC) which determines the actions each user is allowed to perform for each resource.

CONTAINER BEST PRACTICES:

- Image verification using checksum or signature: As many open-source code is used for projects, images should always be verified, even if the image is pulled from a legitimate site or repository. Compromised images can lead to supply chain attacks.

- Image scanning: Images should be scanned to detect any possible anomaly, malicious code or bad practice to prevent either attacks or vulnerabilities being exposed.

- Follow least privilege principle for applications: Applications should never run as root, if this were to happen, malicious code/users could escalate privileges, gaining access to the full container features.

CODE BEST PRACTICES:

- Code scanning: Before running code on a Pod it should be scanned, else, bad practices on coding, or code that was injected by an attacker in an infected image could either result in vulnerabilities being exposed, or attacks upright.

- Process whitelisting: Application processes running in a pod must be known and identifiable. A runtime whitelist of processes should be made for each pod. The appearance of an unknown process is usually a signal of an attack happening, and therefore it should be shut down automatically.

3.1.3 Open Source Security Tools

Open-source security tools are useful to achieve secure configurations, detect vulnerabilities, trace suspicious activity, follow best practices and monitor the general activities of the Kubernetes cluster. If a security tool is designed to run before the deployment of the cluster is called static, while tools designed for runtime are called dynamic.

Some of the most popular static tools are:

- Kubelinter[6]: Checks for both vulnerabilities, and bad practices that could result in vulnerabilities. It acts in a similar manner as the Unix Lint command.

- Checkov[7]: Runs thousands of checks on a Kubernetes cluster, while it was originally static it can run some checks dynamically. It acts similarly to Kubelinter,

- Kube-Bench[8]: Performs benchmark tests, provided by the CIS (Centre for internet security).

---

6 **https://docs.kubelinter.io/#/**

7 **https://www.checkov.io/**

8 **https://blog.aquasec.com/announcing-kube-bench-an-open-source-tool-for-running-kubernetes-cis-benchmark-tests**

Main dynamic open-source tools are the following:

- Falco[9]: Performs dynamic threat detection, checking for unexpected sys calls, logging suspicious activities, and placing detection triggers on multiple layers of the technology stack. Such as examining cloud audit logs (like Cloudtrail), checking the state of the cluster and its containers, scanning application code and host OS and kernel calls [43].

- Kube-Hunter[10]: Runs test dynamically over the selected IP's, domain names or networks. Said tests point out network bad practices and are very useful running along other intrusion detection tools or port scanners.

- Terrascan[11]: Allows continuous integration and development through GitHub/GitLab. Further, interacts with Kubernetes via admission webhooks. Admission Webhooks are HTTP callbacks that receive admission requests and perform actions accordingly, in this case, running scripts or tools if suspicious requests reach Kubernetes.

- Prometheus[12]: is an open-source toolkit for system monitoring and alerting built by SoundCloud[13] in 2012. Prometheus [44] generates collections of data observations in OpenMetrics format. A collection is identified by metric name and is composed of a time series of key/value pairs. Inside a cluster, Prometheus server follows the pull model to scrape local data metrics at regular interval from applications of a cluster. Each application offers a HTTP endpoint with the metrics. A Prometheus server is single server with centralized storage in a cluster. To forward metric streams collected by Prometheus from outside the cluster, the Prometheus Remote Write protocol is used. These forwarding is done with the push model. Prometheus is not a security oriented tool, but it can be programmed to analyze security issues and generate alerts using its component Alertmanager.

9 **https://falco.org/**

10 **https://kube-hunter.aquasec.com/**

11 **https://runterrascan.io/**

12 **https://prometheus.io/**

13 **https://soundcloud.com/**

- S. Karode's monitoring system [4]: this system is inspired in the Prometheus architecture. It sniffs traffic from pods running in a cluster using side containers. Useful to detect certain attacks and to monitor the cluster during runtime. This system was the main part of Sameer Karode's Masters Dissertation in Trinity College Dublin.

In juxtaposition to open-source solutions, major companies, such as RedHat, Amazon, or Google, involved in developing hybrid cloud managers, are developing their own security tools tasked with security controls and security policy enforcement.

3.2. Kubernetes Misconfigurations, Vulnerabilities, Exploits and Possible Solutions

In this section, an overview of common networking misconfigurations that result in vulnerabilities, inherent Kubernetes networking vulnerabilities, their exploits and possible solutions will be made.

This section will go in one part from most internal networking, container to container, to most external, component to the Internet, component to the cloud. In the second the general threat matrix of Kubernetes will be reviewed.

It will be important to keep in mind that given the expanse of known vulnerabilities in the Kubernetes networking architecture it will not be possible to detail every single one of them.

3.2.1 Container Related Networking Vulnerabilities and Misconfigurations

The first relevant vulnerability comes from the pause container. Pause containers are in charge of keeping the network specifications of every container in a pod, specifically the netns. As a result, escaping from the Pause container netns means also escaping from the pod netns, ending up in the host node netns. An attacker capable of this, would be able to see network interfaces, routing rules and other pod's netns, giving the attacker full knowledge of the Kubernetes cluster network architecture. Moreover, if the attacker had privileged access, the whole node would be compromised [36]. In my opinion to avoid an attacker exploiting this design flaw the administrator should limit as much as possible how the pause container can communicate, for

example not allowing SSH to the pause container and setting up an IDS to track every packet sent to and from the pause container. But most importantly, the container, and as a consequence the worker node should be as isolated as possible, while this does not fix the issue, it helps prevent it. It could also be interesting to use honey pots if an admin suspects this kind of attack will be present.

As mentioned in section 3.1.1 (Kubernetes Network Security vs Traditional Network Security Architectures) containers use the same network stack and communicate over localhost. Moreover, network policies cannot limit localhost connections in between containers, nor set up TLS. This means that moving sideways between containers is trivial as there are no limits, nor authentication, nor ciphering between them. As a result, a container being compromised will probably result in every container in the node being compromised, which will end with the whole pod being compromised. The only way to defend against this, the same as the last issue, is to isolate the host pods and host node as much as possible, in order to avoid any attacker compromising a pod, as both pods and nodes can be limited in how they communicate by the network policies.

Network policies can only affect how containers communicate with the exterior of the node. As a result, if an attacker compromises a container, and gains administrator rights the whole node will probably be compromised. To avoid this, first, every component in the cluster should use TLS to communicate with the exterior, second, RBAC should be set up, so even if an attacker reaches a container, it would not be able to perform every action they want, third, set up a monitoring system to check for suspicious activities on the container fourth, secrets should not be stored inside containers, and finally containers should be given the least privilege possible.

3.2.2 Pod Related Networking Vulnerabilities

Pods communicate in between each other with the help of the CNI. This plugin is in charge of routing packets and enforcing security policies, as it keeps track of the IP and location of each pod (subnet and node). If two pods on the same node want to communicate, they will do so over a virtual ethernet bridge [36]. If a container or Pod were compromised, CNI plugins using that bridge could become vulnerable to Layer 2 network attacks:

- **ARP Spoofing**: if pods in a cluster are allowed low level access to the network with the NET_RAW option, a malicious pod could use the ARP protocol to perform a Man in the

middle with every other pod in the node, by discovering the MAC and IP address of said pods and impersonating them.

- **DNS spoofing** [45]: DNS requests on a Kubernetes cluster always arrive at the Core DNS pods. If the Core DNS pod can resolve the request it will return an IP in the local cluster, if not, it will, by default, redirect the query to other nameservers (Figure 12). An attacker with access to a pod with RBAC enabled could be able to ARP spoof the ethernet bridge, to supplant the cluster DNS server. As DNS requests arrive at the ethernet bridge, the attacker would have full control on DNS resolutions in the cluster. This is achieved by crafting packets, which in turn is only allowed if the NET_RAW capability is enabled. (Which usually is, quoting Kubernetes and docker "It is unfortunate that the Kubernetes & Docker container default is to allow CAP_NET_RAW, but to maintain backwards compatibility we don't think we can change this default in the short term" [45].

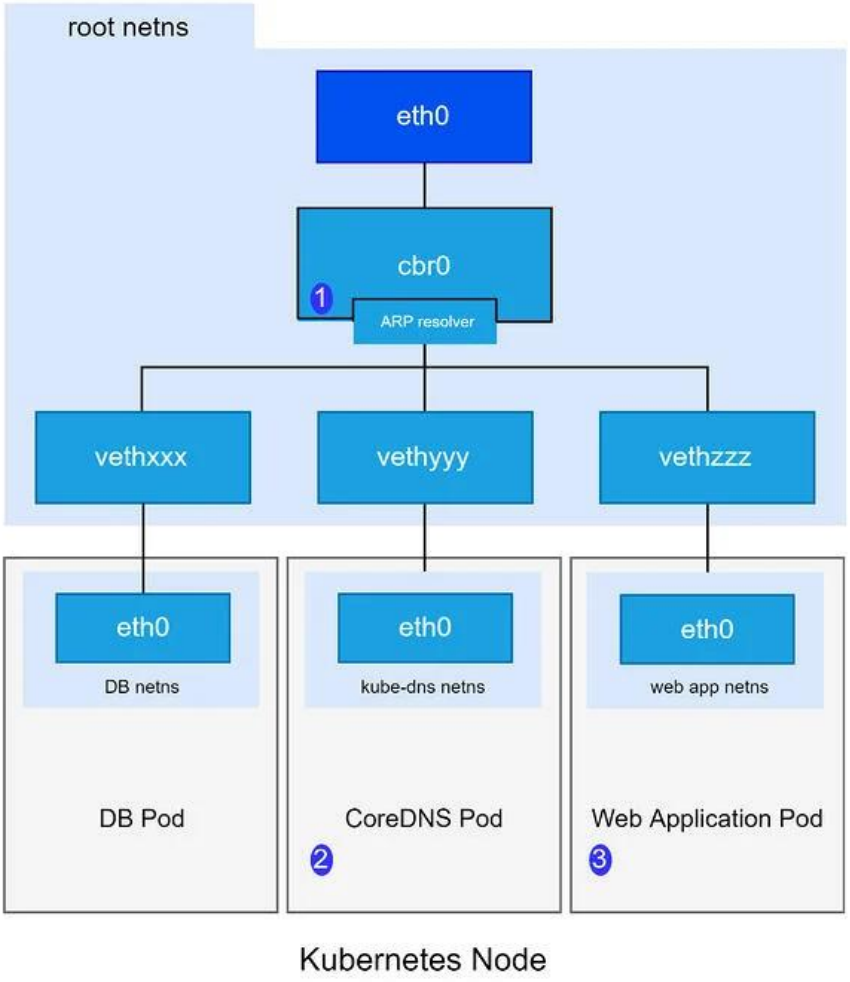

Figure 12: Kubernetes DNS Example architecture (Source: Aqua Blog).

As mentioned the best possible defenses against the exploiting of these two vulnerabilities, and by extension every L2 vulnerability are to segment the network as much as possible, to reduce the elements a pod/container can communicate freely, to drop the NET_RAW options from pods to prevent an attacker from being able to craft and modify packets, and to set up L3 routers instead of L2 switches, this way reducing the broadcast domain of pods/containers. Unfortunately, changing from ethernet bridges to IP switches means the whole architecture needs to be modified. Moreover, even if these measures are taken there are still ways to perform similar attacks, this serves as a perfect introduction to the next subsection.

3.2.3 CNI Related Attacks

A CNI implements an overlay network, which is a virtual or logical network running on top of a physical network. This is usually done with Virtual Extensible LAN(VXLAN) in many CNI's such as Flannel[14], Cilium[15], WeaveNet[16] and Canal[17]. The reason to use VXLANs is to distinguish node net spaces from pod subnets without NAT, in order to comply with Kubernetes' internal networking requirements. This means, that even if a pod has RBAC and the NET_RAW option dropped, but needs to communicate via VXLAN, the cluster is vulnerable to attacks if that pod is compromised. An attacker with access to the pod, could sniff packets to identify destination MAC address and port of the Virtual Tunnel endpoint (VTEP) of the VXLAN. With this information an attacker can encapsulate malicious VXLAN traffic aimed at pods in other nodes (as it knows the VTEP) in IP packets directed at the node where the pod is hosted. This is possible because node specific network policies cannot be defined. This means, for example, if an attacker compromises a pod, it could easily perform a DoS attack on the pod where the cluster database is located. On top of that, if the CNI uses IP-in-IP encapsulation, the same attack can be performed just by IP scanning, encapsulating IP traffic to the target pod, and sending it to the node [46].

CNIs have many options to update routing data, one of these options are routing daemons that use protocols such as BGP or Border Gateway Protocol. If an attacker gains control over the routing updates, stealthy man the middle attacks can be achieved. The vulnerability CVE-2021-

[14] https://github.com/flannel-io

[15] https://cilium.io/

[16] https://www.weave.works/oss/net/

[17] https://ubuntu.com/kubernetes/docs/cni-canal

26928 is a clear example of this (which is still unpatched but has been addressed in terms of how to best defend against it). To explain the vulnerability and how to exploit it, Calico[18] will be used as an example, as it is one of the most popular CNIs for Kubernetes. BGP will not be explained in this dissertation as it is outside of its scope, but it is relevant to know that before any BGP message is sent a full three way TCP handshake is required and that only BGP peers can update routes. Calico uses Bird as its BGP daemon. A BGP Hijacking attack consists in the injection of a new malicious route that is more specific than the current route, this way BGP will always interpret the most specific route as the fastest and most efficient.

If an attacker can execute code remotely in a pod, or the compromised pod has HostNetwork Configured, it can establish a TCP session with the valid BGP peer, this way the Calico daemon will close the connection to the valid BGP peer, then the attacker can, in the meantime, send BGP packets updating the route to a malicious address which will act as a man in the middle [47].

Giving HostNetwork to the pods that need it, configuring the CNI daemons that use BGP to use TCP-MD5, and updating Calico to a version that can ask peers for passwords to authenticate BGP peers are the best mitigations for this attack.

The latter were specific cases of attacks on CNIs related to either the general architecture used, or to how the specific CNI works. Additionally, there are some security implications related to the usage of CNIs. These are, as CNIs are in charge of enforcing network policies, thus to some extent in charge of firewalling, if an attacker were to compromise the CNI, it would automatically gain privileged access to the worker nodes. Moreover, if the CNI was rendered useless, the firewalling it achieves, and the network policies would stop having effect. On top of this [ref understanding] given the vulnerabilities of the CNIs, the crippling limitations of network policies, and that component IPs are dynamic, this is, they change constantly, achieving proper firewalling is extremely difficult [36]. This is because network policies do not affect relevant parts of in-cluster traffic and using SDN adapted firewalls will not solve that problem either, as the firewall will not be able to see traffic inside the cluster. This is also a reason why the tool was developed.

[18] **https://www.tigera.io/project-calico/**

These attacks are the result of the design of the Kubernetes architecture and how pods, containers and nodes communicate in between each other. Unfortunately, the Kubernetes threat matrix is more extensive and affects a variety of specific components, configurations, plugins, addons, and so on.

In the next sections a brief overview of all of these attacks and their corresponding defenses will be conducted.

3.2.4 Attack on Nodes

Even though it is mentioned throughout the dissertation, if an attacker gains access to a node, it is not hard to escalate privileges and move sideways until the full cluster is compromised. For this reason, it is extremely important to set up RBAC, give minimum privileges to nodes, and close every single port that is not necessary for the node to work properly.

3.2.5 Attacks on Kubernetes API Server

While there are components that can expose endpoints, the API server should be the only one exposed to the exterior of the cluster. Access to the API server can be done via kubectl or other libraries that allow applications to make REST calls to the API. As the API offers both secure and insecure ports, an attacker can freely access components, control them or create new ones bypassing all security measures such as authentication and authorization. To avoid this the unsecured port should be closed and every connection towards the API server should use TLS.

Even after this, anonymous health checks to the cluster can be performed, and could subsequently be disabled.

The key to defending the API server is to set up Authentication, Authorization and Admission. The first can be easily done with TLS. In case an unknown user makes an HTTP request to the cluster it will be denied. Authorization is made using RBAC and it is supported by Kubernetes. If an authenticated user makes a request, the RBAC policies will determine if the request is allowed to go through or not. Admission is handled by the admission controllers, which can modify, deny or let through requests to the API independently of the user's role, for example denying exec or attach commands from any container.

3.2.6 Attacks on Control Plane Traffic

Even if it goes against Kubernetes' network architecture, every control plane component should be configured to use TLS. This way, even if an attacker has access to a pod, it will not be able to eavesdrop control plane traffic, nor substitute genuine components with malicious components. To maintain security, TLS certificates should be cycled periodically, and components should be updated constantly (thus if a component is compromised the deployment will be new, instead of a redeployment of the infected component).

3.2.7 Kubelet API Attacks

As Kubelet acts as an intermediary between the API server and Container runtime it should always use TLS bootstrapping, so TLS certificates are generated for each new node. If this is not done, the RBAC is not enabled for the Kubelet, or the insecure port in the Kubelet API is open, an attacker could run commands on the Kubelet, gain access to pod secrets, and in essence take over the cluster.

3.2.8 Container Runtime/Image Attacks

Kubernetes can use different container runtimes, depending on where the application is deployed. The most common one is Docker. In order to avoid the runtime being compromised, it is recommended to configure Kubernetes only to pull images from secure and registered repositories, to only pull the latest image, and to scan every image in search of malicious code and/or vulnerabilities. Using docker as an example, images that are not found locally are pulled from Dockerhub. Any user can upload images to Dockerhub, including vulnerable images. An attacker may download an image from Dockerhub and reupload it with a backdoor added to it. This way if an image is not scanned, the user incurs the risk of an image being malicious, infected, or backdoored.

3.2.9 Escaping from Container to Host

An attacker can jump from the container to the host system. This is usually done by exploiting kernel vulnerabilities, using privileges from a container running as root, or taking advantage of other misconfigurations. Generally, to prevent this from happening containers should always follow least privilege principle best practices. Moreover, there are specific configurations, such

as the privileged flag that allow this attack to happen, this flag should always be disabled if possible.

A container with the privileged flag activated is capable of using every kernel capability and to access the host devices. This way, an attacker with access to a privileged container may load modules to the host's kernel that allow the attacker to reverse shell the host or run malicious code.

3.2.10 Privilege Escalation

Privilege escalation consists in gaining root privileges from an unprivileged account. Privilege escalation can be done in many ways, but the most common is exploiting misconfigurations in RBAC, excessively privileged add-ons and infrastructure components.

For example, very commonly used add-ons featured powerful daemons, before updates (this is capable of performing privileged actions), whether they were open source such as Antrea and Cilium or not, such as Analyzed Kubernetes Platforms (from Microsoft Azure), Elastic Kubernetes Service (from Amazon Web Services) and Google Kubernetes Engine [48].

An example on how to perform privilege escalation is the following:

When a pod is created it is assigned the node best fit to host it. This decision is made by Kubernetes by default, but by modifying the node-selector option in the pods YAML, an attacker can force the pod to be run in the master node. "nodeSelector is the simplest recommended form of node selection constraint. You can add the nodeSelector field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify" [49]:

```
spec:
    tolerations:
    - key: ""
    operator: "Exists"
    effect: "NoSchedule"
```

The pod in question will always be deployed in the master node independently of the node selector filtering out master node as an option for hosting pods. From there the attacker can add commands to the YAML file to access and steal the private key and certificates of the master

node (which in this case will be the Certificate authority or CA), present in the master node files in the following manner:

```
command:
    - sh
    - -c
    - echo -e "KEY:\n"; cat /etc/kubernetes/pki/ca.key; echo -e "CRT\n"; cat
        /etc/kubernetes/pki/ca.crt; sleep 3600; volumeMounts:
    - mountPath: /etc/kubernetes/pki
```

Those commands mount the pod in the location where the PKI files are located, and then print the private key and certificate. With these files an attacker can create a new certificate and use it to access kubectl as a cluster admin [50] [51].

The best way to defend against these types of attacks is to forbid pods from being scheduled in the master node, forbid pods from being deployed in the hostPath volumes, restrict access to the etc/ files, and forbid the use of arbitrary tolerations. This can be achieved by using admission controllers.

3.2.11 etcd Attacks

The etcd is used to store all cluster data in Kubernetes, including the IP addresses of the cluster, and secrets. By default, all of this is stored plaintext. If an attacker has access to the master node, it will also have access to all the relevant cluster information. For this reason, first, everything stored in the etcd should be ciphered, and second every component that communicates with the etcd should do it through TLS, so that way connections to the etcd may not be done without a legitimate certificate. Finally, the etcd should be firewalled, so reconnaissance may not be done [51].

3.2.12 Kubernetes Dashboard Attacks

The Kubernetes Dashboard is a web-based UI, that allows users to deploy applications in a cluster, troubleshoot the apps, monitor the cluster, manage cluster resources, overview applications and create deployments, services, pods etc. By default, the Kubernetes dashboard is public, and unprotected, this means anybody can access it, and therefore, have admin rights to the whole cluster.

In order to access the Dashboard in a secure configuration a user needs either a token or a kubeconfig file, which contains a certificate signed by the cluster's CA, this CA is the same for the whole cluster, although a third-party CA may be used. To create a kubeconfig file a user generates a key and uses it to generate a Certificate Signing Request (CSR). This CSR is passed by the cluster admin to the CA in order to be signed, obtaining a signed certificate (CRT). Then the kubeconfig file is created with the CRT, the CA's name, the cluster's name and the user's name. If the user does not have a role, a role is assigned to it [51]. Moreover, dashboards in order to be protected should always be firewalled, and only accessible through reverse proxies.

4 Near Real-Time Scalable and Fault-tolerant Monitoring Tool

One of the main goals of this thesis is to design and implement a tool prototype for near-real time monitoring of TCP traffic in Kubernetes. This tool shows raw and aggregated data in a dashboard and stores it for batch processing. This chapter shows the phases of the waterfall software development life cycle used for the development of the monitoring tool. These phases are requirements, design, developing and testing and operations.

4.1. Requirements of Monitoring Tool

The monitoring tool must fulfil the following requirements:

- **Oriented to Kubernetes**. The tool must monitor applications run in Kubernetes.
- **TCP packets sniffing.** The tool must sniff TCP packets from one or more applications.
- **Visualization and storing of data.** The data must be visualized in a dashboard and stored for batch processing.
- **Near real-time processing of packets.** Sniffed packets are available in less than 1 second. to be stored, processed or visualized.
- **Fault tolerant monitoring service.** The tool continues working even in presence of failures inside of any of its components.
- **Scalable service.** When a there is an increasing flow of data, the tool does not decrease its quality in terms of response time when displaying or storing data [52].
- **Use of open-source software:** to offer an open tool that can be modified and improved**.**

- **Multi IT infrastructure environments to run.** The tool prototype will be able to run on different HW platforms. From hosts with restrictions of time and resources, as a laptop, to deployments in the cloud.

4.2. Design of the Tool

The design phase produces the main architecture of the tool, taking into account the analysis requirements. Before explaining this architecture, it is needed to give a brief definition of the sidecar pattern used to be able to sniff packets from the application.

Sidecar pattern. It is a microservice design pattern used to offer external, but related functionalities to a service or application, such as monitoring or logging. The sidecar name comes from the small one-wheeled vehicle attached to a motorcycle. The application can live without the sidecar service but, as the sidecar is useless without a motorcycle, the sidecar service makes no sense without the application that is attached to. That means that the sidecar service shares the same life cycle and resources with the application.

The design phase describes the election of the following technologies for fulling the requirements:

- **Oriented to Kubernetes and any HW**. Election of Minikube[19], as it is a lightweight open-source implementation of Kubernetes cluster with one computer that can be deployed in a local machine.

- **TCP packets sniffing**. The sniffer module uses the open-source packet Scapy[20] packet sniffer tool. The sniffer module is a sidecar container that runs in the same pod as the monitored app, as both containers share the same network interface.

- **Near real-time processing of packets.** To get fast monitoring, the concepts of event stream processing and fast indexing of data were applied. The open-source Strimzi[21] Kafka service offers the publish/subscribe Kafka Service in Kubernetes [53].

---

19 https://minikube.sigs.k8s.io/docs/

20 https://scapy.net/

21 https://strimzi.io/

- **Fault tolerant monitoring tool**. Obtained configuring Kafka[22] and Elasticsearch[23] were configured as fault tolerant services. For example, setting up a Kafka service with at least three brokers and its internal Zookeeper service replicated, permits at least two simultaneous failures in Kafka brokers without stopping the service.

- **Scalable tool**. When the flow of sniffing packets grows, response times degrade. The concept of partitions in Kafka topics can be considered parallel processing of packet substreams. The concepts of fault tolerant and scalability in Kafka are explained in detail in the background chapter.

- **Near-real time storing, recovery and visualization of information**. Fast indexing of data will be obtained with Elasticsearch. The data can be searched hastily and sent to the Kibana dashboard. Kibana has been chosen because it belongs to the Elasticsearch stack (ELK stack).

The Kafka, Elasticsearh and Scapy tools have been described in detail in the chapter 2 (Background) section 2.6 (The Elastic Stack).

Figure 13 shows the first approximation of the design of the architecture with its main modules: app pod, Strimzi Kafka service, scraping service with Elasticsearch and Kibana dashboard.

From top to bottom, the architecture elements are described in more detail:

- **App pod** has two containers:
    - **The app container** runs the application to be monitoring.
    - **The sidecar sniffer container** comprises two modules:
        - **TCP sniffer module** that sniffs TCP packets using the Scapy library.
        - **Kafka producer module** publishes each packet in a topic of the Kafka service. The name of the topic might be the name of the cluster where this app is running, and the record containing the packet is a tuple with the form <key = app Id, value = JSON TCP packet>. The key contains an app

[22] **https://kafka.apache.org/**

[23] **https://www.elastic.co/**

identifier, and the record value contains the TCP packet in JSON format. The publishing requires the use of serializers for the key and JSON packets to convert them to a sequence of bytes.

- **The Strimzi Kafka service has two modules:**
  - **One or more Kafka brokers** that maintain the topics with the records published in them.
  - **The Zookeeper service** maintains the synchronization of brokers using a znode tree.

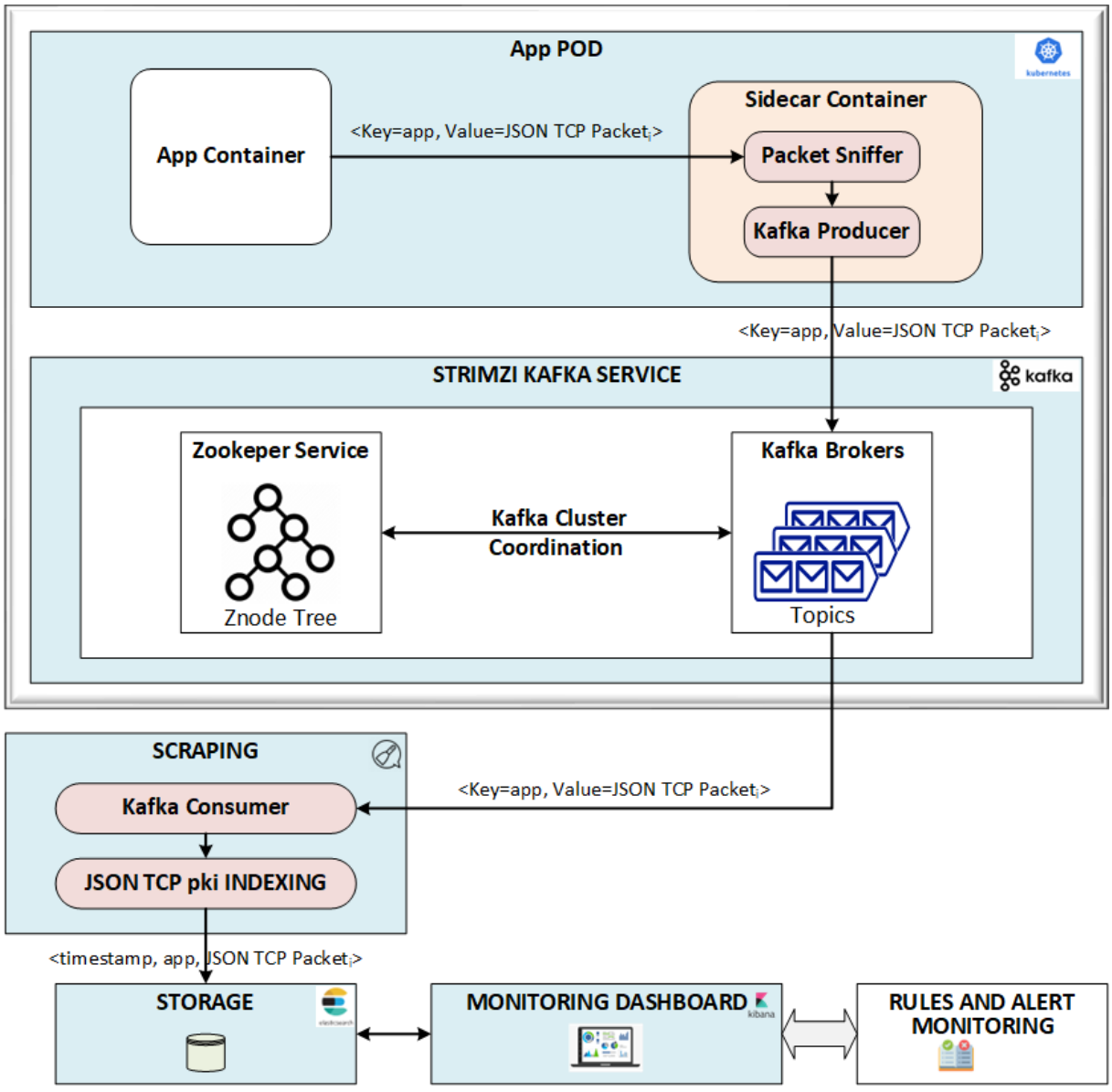

Figure 13: First approximation of the architecture of the monitoring tool.

- **The scraping service has two modules:**
    - **A Kafka group consumer** consumes the records published in the Topic in FIFO order. A deserializer must be used to rebuild the record as a tuple <key, value> with the original types.
    - **Elasticsearch indexer** that indexes and stores each record with the Elasticsearch engine.
- **The Kibana Visualization Dashboard** module for creating dashboards of indexed data or aggregated data.

4.3. Implementation of TCP Monitoring over a Guestbook App

To focus the efforts on the tool and reduce the complexity and errors not related directly with the development of it, both a Kubernetes cluster with only one node (Minikube) and a simple application were used (Guestbook application).

Guestbook app with Redis[24] description. The Guestbook application is a stateless web application for posting messages onto a message board. This application is composed of two services: the PHP Frontend service and the Redis service in the backend. Redis is an open-source data store service held in memory. The frontend service sends read and write message requests to the Redis service. The Redis service is configured with two services: one leader and one follower. Write requests are in charge of the Redis leader while read requests are in charge of the Redis follower.

The design shown in 4.2 has to be adapted to include the Guestbook app. Figure 14 shows the new design where the Guestbook frontend, the Redis leader and follower containers appear along with their corresponding sniffer sidecars.

[24] https://redis.io/

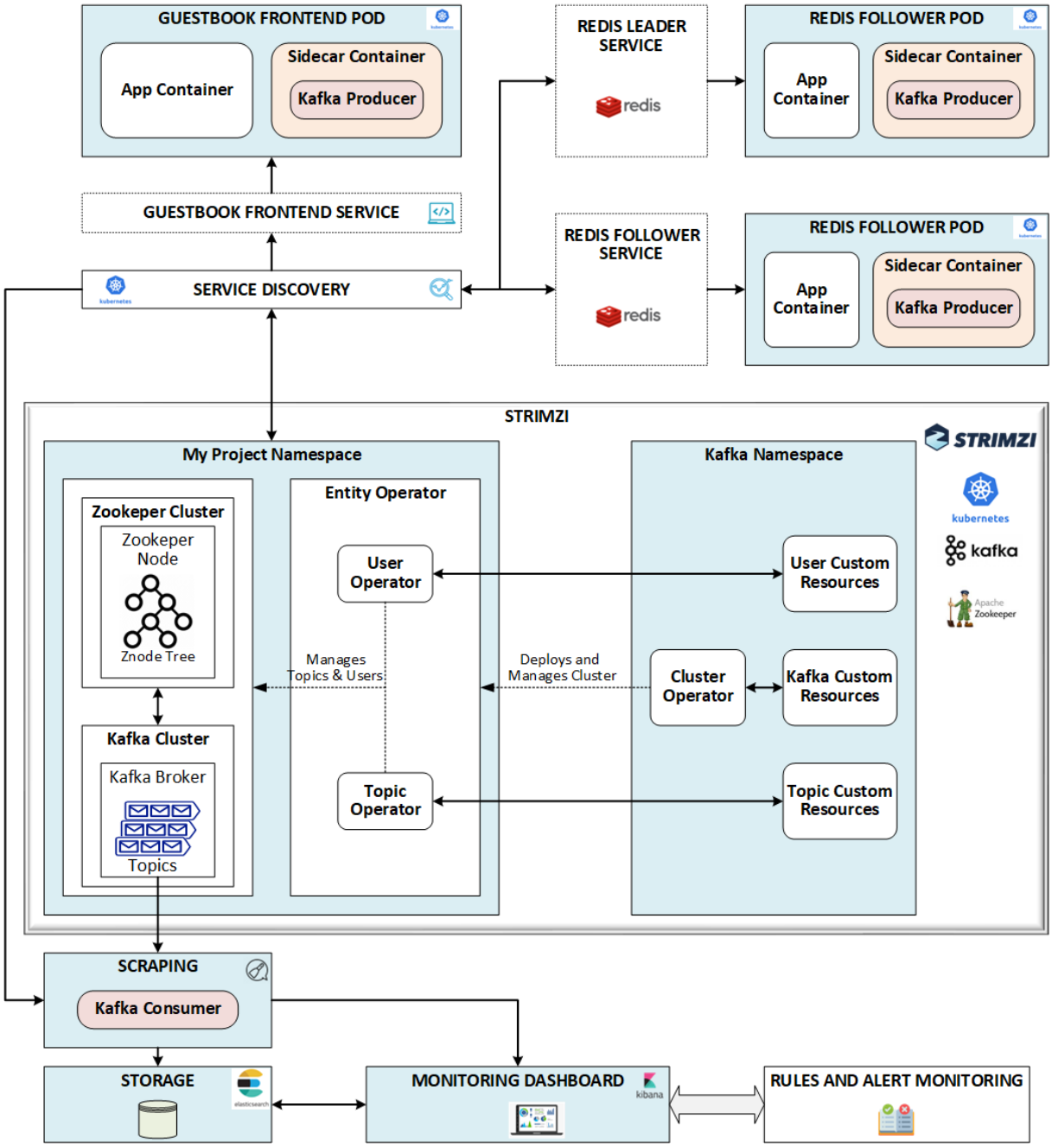

Figure 14: Example of TCP Monitoring Guestbook App Architecture.

Note that now it is needed to distinguish three sniffing flows (frontend, leader and follower) to process them separately. This thesis proposes the use of three different Kafka keys. All sidecars will publish in the same topic “guestApp”, and their keys will be “frontend”, “leader” and “follower”. The frontend side card will publish in the topic “guestApp” with key=”frontend”, the sidecar leader with the key=“leader” and the follower with the key=”follower”. This choice was made to ease the obtention of a scalable architecture, as it allows parallel processing. This can be achieved by designing the scraper to assign a consumer process per stream.

4.4. Comparison of this Monitoring Tool with the Karode's MSc Thesis

As little open-source monitoring solutions exist, a subgoal of this thesis is comparing this thesis' Monitoring Tool with the monitoring tool developed in the MSc thesis of TCD student Sameer Karode [4]. This section compares both monitoring systems in terms of throughput (sniffed TCP packets/s), fault tolerant and scalable monitoring servicing.

Similar to this thesis, Karode's thesis proposed the use of the sidecar pattern for monitoring based on the Prometheus architecture tool [44]. The application monitored in Karode's dissertation was PHP Redis guestbook, mounted on Minikube.

But one drawback of Karode's thesis was the lack of scalability and fault tolerant service.

PHP Redis guestbook has a frontend that displays the messages written in the guestbook and allows users to write new messages, communicating with the Redis leader service in case of a write request and with the Redis follower service in case of a read request [54]. This application exposes the frontend service to the external network via the Kubernetes proxy.

Looking at Figure 15, Karode architecture adds sidecar containers to each of the leader, follower and frontend services. Each sidecar container runs a Meteorshark[25] instance. Meteorshark is a multithreading program, in which one thread is in charge of sniffing packets, using Scapy [55], formatting them in JSON and adding them to a queue, while the other thread pushes the packets to Meteorshark's REST app. Each pod has its corresponding REST app.

In Karode's architecture in Figure 15, the scraper that runs outside Kubernetes is programmed using JavaScript (js) implementing js promises to receive the packets posted by the sidecar containers, bundling them every three seconds or if 3000 packets arrive to later represent them. The Monitoring Dashboard process accesses the scraping server to download sniffed packets from the three pods and show them.

While this architecture for packet sniffing and monitoring for Kubernetes is interesting and provides an idea for decentralized monitoring for Kubernetes networks, it also has some design

[25] https://github.com/thepacketgeek/meteorshark

flaws that are either inefficient or compromise the security of the architecture. First, Karode used the HTTP protocol to transfer the sniffed packets throughout all the architecture.

As the guestbook application has three pods, Karode's architecture has to simultaneously maintain three HTTP connections for each sidecar, another three between the scraper and the Meteorshark REST apps and one linking the scraper to the monitoring dashboard. This solution is not scalable, as the number of web servers and HTTP connections grow linearly with the number of the pods. In addition, HTTP pipelines are not the best fit to handle high throughput, low latency, real time data streams compared to TCP. Consequently, as in an intrusion detection context, the speed in which an attack is detected is crucial to prevent further damage, the proposed strategy was found to be sub optimal.

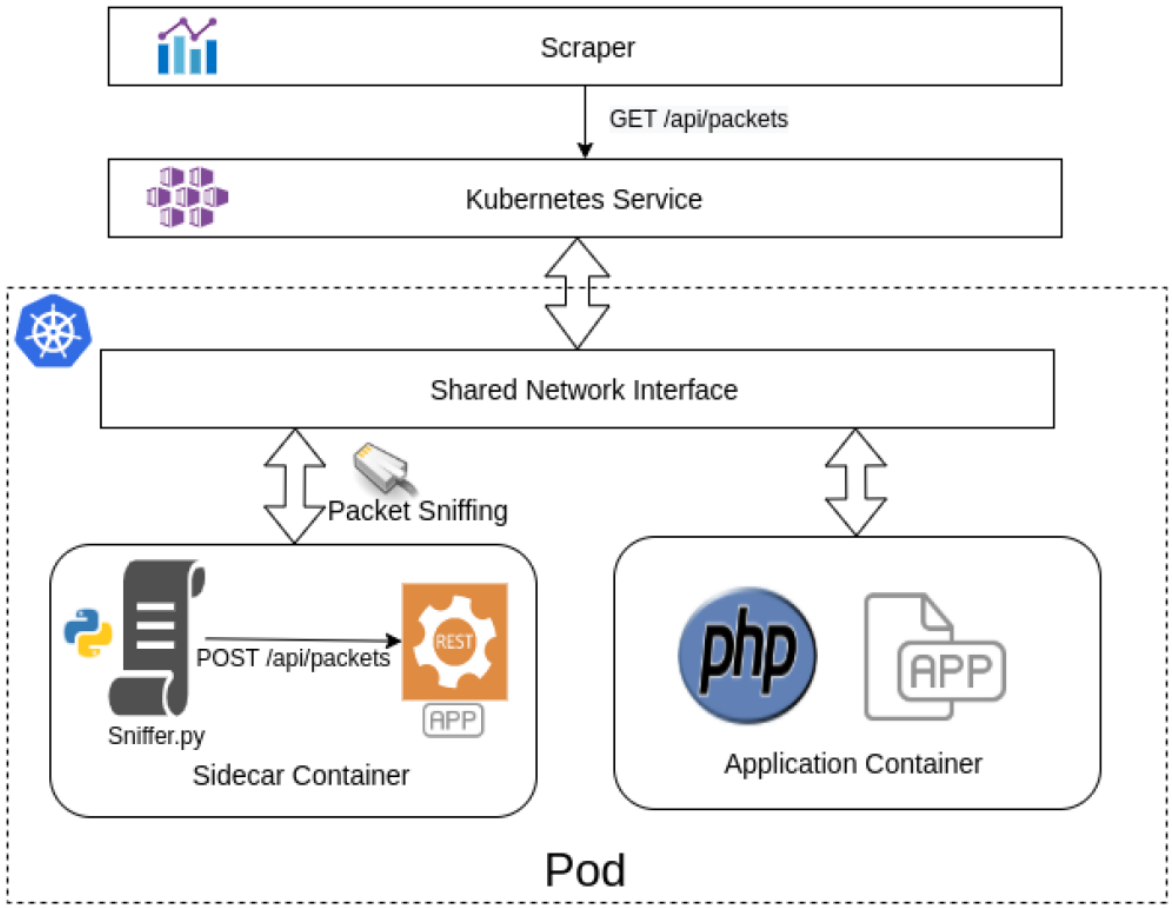

Figure 15: Architecture diagram of packet sniffing sidecar container (Source [4]).

Additionally, Karode's sidecar code needs to be modified every time the IP addresses of the sidecar containers or dashboard change, as these IP directions are hardwired in the code. Therefore, each deployment of this system, especially knowing the volatility of IP directions in Kubernetes systems/clusters, would require additional code modifications, set up, and image building. Therefore, if multiple instances of a dashboard need to be deployed, or packets need to be sent to different machines, first each IP would have to be added to the sniffer code and second, each packet would have to be resent several times once more for each new receptor.

Karode monitoring dashboard, while effective in its original purpose, lacks several functionalities in terms of data visualization, alerting and responding to incidents. Namely, it is not configured to alert certain users if attack signatures are detected (even if it can detect certain ones) and does not allow users to easily build complex views of the packets being received without modifying the code extensively if it is possible.

Finally, while the system is focused on the monitoring aspect of an intrusion detection system, it seems there are several architecture design improvements, pertaining to pod/cluster/service isolation, exposure to the public Internet, ciphering not being implemented, lack of RBAC, least privilege principle not being followed, in between others. If this system were to be implemented in a real life industrial environment, it would be deemed as high risk vulnerabilities. Probably, it was not feasible to address these issues properly given the time frame in which a thesis is made, but at least they should have been identified, and solutions should have been proposed for future work.

All the aforementioned reasons led to the redesign of the architecture ih this thesis. The comparison is featured in Figure 16.

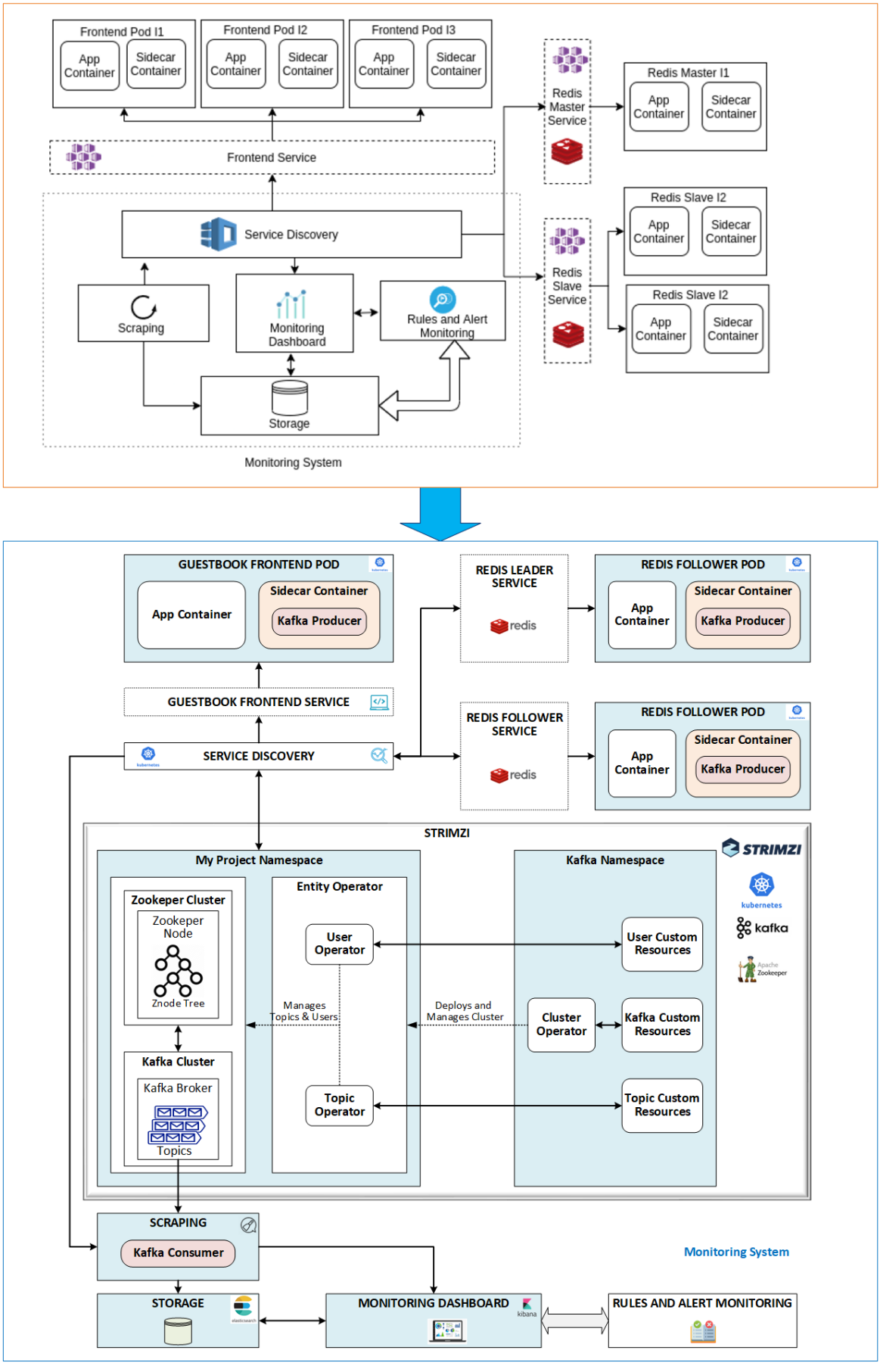

Figure 16: Monitoring tool architectures comparison.

4.5. Monitoring Results

The result of the coding section of this thesis was a Kubernetes monitoring tool, capable of being integrated to an application, in this case Redis guestbook. This tool graphically displays the packets sniffed from every pod in the application, allowing for many visualizations, options, and to configure alerts when certain conditions are met.

Given the conclusions drawn in chapter 3, a Kubernetes cluster should be monitored globally, with a special focus on their security weakest points. This monitoring should allow the administrators to respond as fast as possible in case of an attack. Monitoring using Kafka in combination with tools from the Elk stack, such as Kibana and Elasticsearch, has a plethora of advantages to achieve this goal. Kibana allows the building of fully customizable views easily, featuring data aggregation, performing mathematical functions on the incoming data, and the segregation of data by any of the packet fields. Moreover, dashboards combining different views can be constructed.

As each Kubernetes deployment may be architecturally weak to certain types of attacks if a pod is compromised, a user cognizant of this information can craft single visualizations focused on a certain pod, including precise ports, using packet data indexed by timestamp. Furthermore, multiple visualizations can be combined in a dashboard able to update as fast as each second.

Packet traffic pattern analysis. The goal of this analysis is to detect anomalies in current traffic behavior, such as traffic spikes, or to compare with previously stored cluster traffic patterns. Figure 17 features a time series visualization with the number of packets sniffed from both leader and follower pods, and two more time series segregating follower packets from leader packets.

Port traffic analysis. The goal of this analysis is to detect suspicious traffic coming from unused ports, or to detect traffic coming from control plane ports that should not be used by the pod. An example of this analysis in Kibana is shown in Figure 18 using a two level pie graph. It displays first, the split of packets coming from leader and follower, and second the destination port of these packets. Kibana also allows the user to construct heatmaps, which were used to split the sniffed packets, from leader and follower, and give each one a destination port heatmap, as featured in Figure 19.

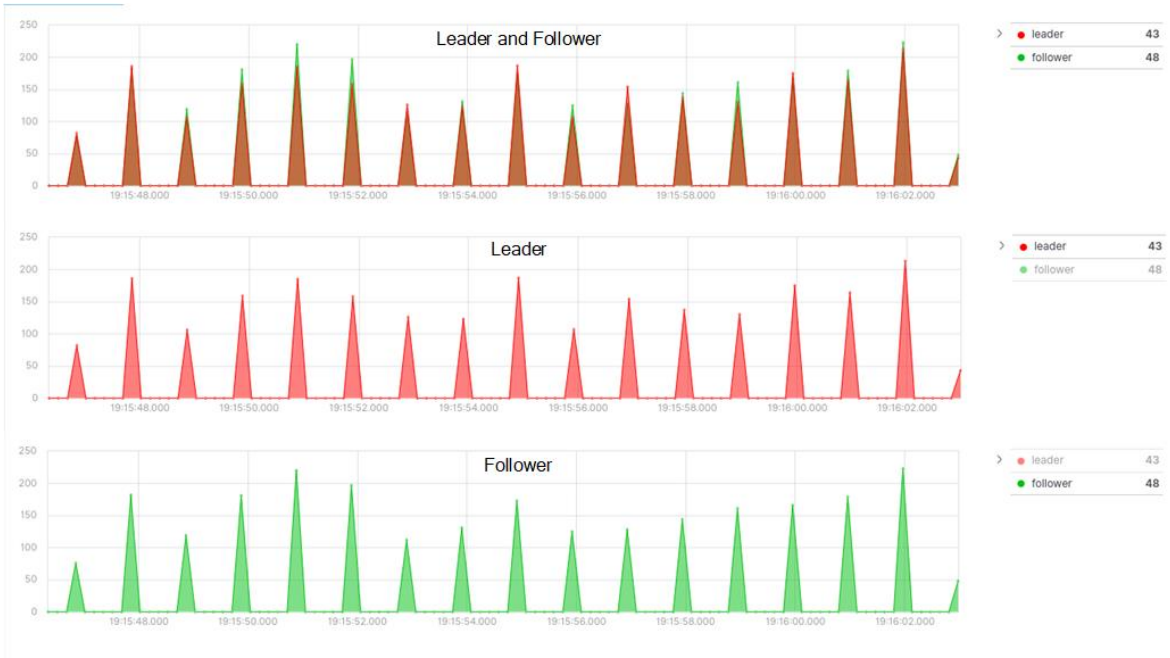

Figure 17: Visualization of packet traffic pattern analysis

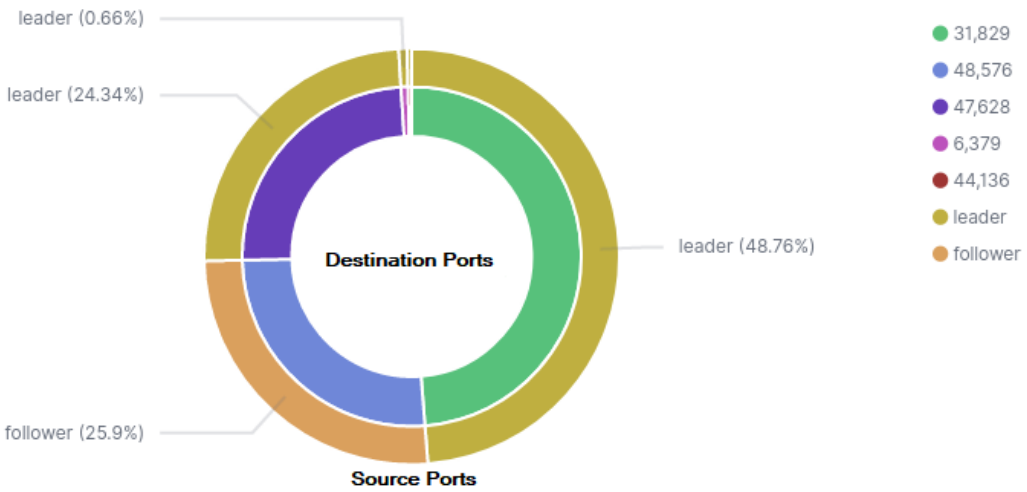

Figure 18: Visualization of port traffic analysis.

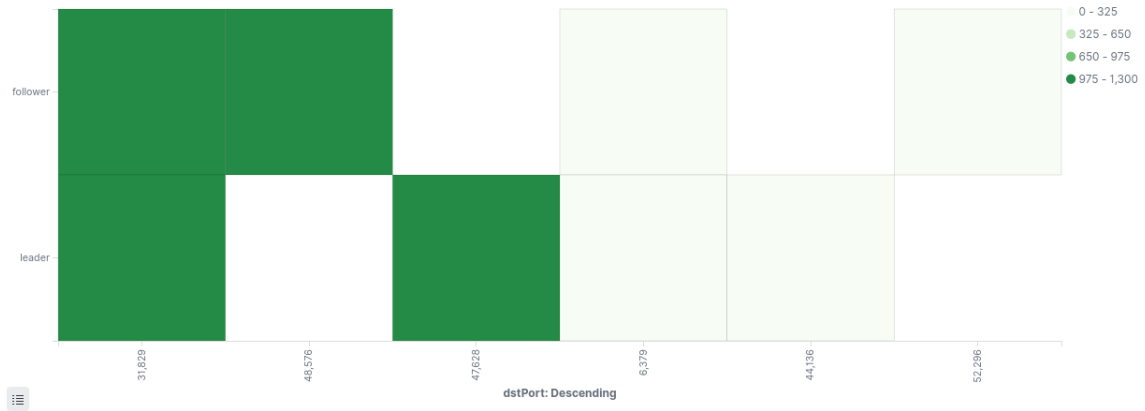

Figure 19: Visualization of heatmap port traffic analysis.

Global monitoring. Finally, Figure 20 shows a Kibana dashboard combining the aforementioned views, including a table of destination ports.

GDPR compliance. In order to comply with the GDPR legislation, IP addresses belonging to users should be anonymized and ciphered before storing them. Despite that, any IP addresses belonging to pods, nodes or services can be captured with no further considerations [56].

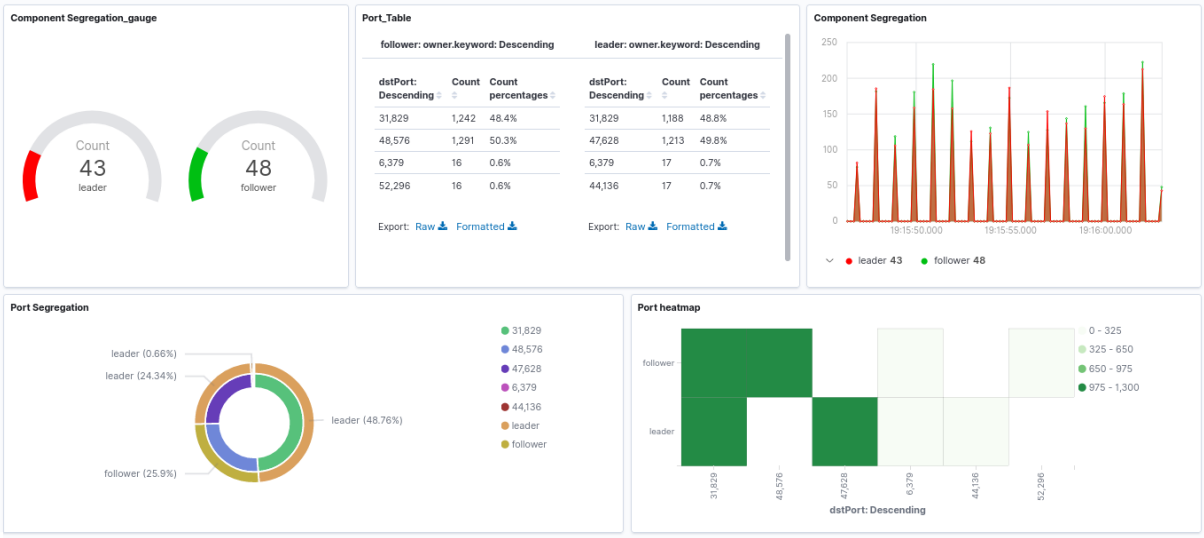

Figure 20: Example of Dashboard.

5 Conclusions and Future Work

The main goals for this dissertation were defined in Chapter 1 (section 1.3 Objectives), and although they were not achieved fully, a working design of a near real-time monitoring tool was achieved. Furthermore, an overview on Kubernetes security including architectural flaws, vulnerabilities, exploits, attacks and defenses was performed.

While achieving a completely secure Kubernetes architecture is a problem still unsolved, this dissertation has both developed a design that could be used in the future and has unified and extended on very relevant security topics and nuances necessary to achieve a completely secure architecture.

This section will summarize the conclusions taken out of the development of the tool and of the research work regarding Kubernetes security and will set the objectives to transform this tool into a full fledged security tool ready to deploy in industrial environments

5.1. Conclusions

From the development of this tool, and the research needed to design it, being aware of the security implications not only regarding the tool but the whole Kubernetes system a great amount of knowledge was gained, which allowed me to reach certain conclusions. First of all the research questions will be answered:

- **What is the current state of Kubernetes security?** → From the research conducted, Kubernetes security is in a state of fragile balance. This is a direct cause of Kubernetes not correctly documenting the security implications of default configurations, not having an exhaustive and comprehensive guide on how to secure a deployment. As a result, much of the documentation on known and solvable vulnerabilities, on best practices and quasi-secure configurations is fragmented across the internet, leading to an astounding amount of security incidents.

- **Which design would be more fit to a distributed and highly scalable paradigm?** → As described in chapter 4 (section 4.2 Design of the Tool) the many benefits of stream processing, was theoretically deemed as one of the best options, this was demonstrated in practice in section 4.5 (Monitoring Results)

- **Would it be possible to analyze traffic in real time in a more efficient way?** → In the same manner as the last research question, the Kafka architecture described in section 4, allows for a much more efficient way of sniffing, sending and processing sniffed packets.

- **How would it be possible to standardize this system in order for it to be deployed in any container-based application?** → The particular methods and requirements in order to transform this tool into a standardized, ready for deployment in any environment tool, are described more in detail in section 5.2 (Future work). But in general terms, there would be a need to implement every single best practice described in chapter 3 (section 3.1.2 Best Practices3.1.2) and add every configuration needed to defend the system against the attacks and vulnerabilities described in Chapter 3 (section 3.2 Kubernetes Misconfigurations, Vulnerabilities, Exploits and Possible Solutions)

- **What are the main problems associated with Kubernetes Security?** → As described in the first research question there is a set of problems related to Kubernetes security:

    - Lack of unified documentation on how to precisely configure Kubernetes in a secure manner depending on the type of architecture used.

    - Lack of acknowledgement of the possible consequences of using default options in the Kubernetes documentation.

    - Steep limitations in regards of the security measures Kubernetes allows to implement without external addons and plugins

    - Kubernetes and its addons and plugins are in a continuous state of evolution, making it very hard to keep track of all the new security implications, patches and features without the proper documentation.

As a general conclusion, while Kubernetes is a very useful tool to deploy and develop distributed applications, even supporting cloud developments has some serious security pitfalls. These security problems arise from, the general degree of misinformation in Kubernetes users

on how to properly configure their deployments to be secure, this is a result of Kubernetes not addressing security properly in their documentation and giving general but imprecise suggestions on how to configure a secure cluster. On top of this, if a user wants to obtain precise knowledge on what are the common vulnerabilities, they will have to use several documents, each written by different researchers or security companies, and still, said user could overlook precise configuration details that could lead to serious vulnerabilities.

Finally, from the extensive research work performed in this dissertation, the second main reason Kubernetes has faced numerous security problems is that, by design Kubernetes is not prepared to handle security, from the limitations of security policies to Kubernetes not patching well known vulnerabilities justifying back compatibility, to Kubernetes not supporting TLS inside nodes, and in essence many of the Kubernetes vulnerabilities are a result of how the architecture was designed. As a result, the best way of securing Kubernetes would be to revise and redesign certain parts of the Kubernetes architecture, quoting Frank Dickson, program vice president, Security and Trust at IDC [57]: "*Security should be embedded throughout the container life cycle. …There is a need to fundamentally change their approach to security, embracing embedded security in the application development process, an approach referred to as 'Shift Left.' Shift left requires one to think less about security products and more about continuous security processes.*"

5.2. Future work

This subsection will be focused mainly on the measures that should be taken in order to make this tool the most secure and useful if it were to be deployed in a real life scenario. While all of these design requirements were kept in mind during the whole development of the tool, unfortunately the lack of time made them impossible to implement. The most important ones are:

- Implement the best practices listed in chapter 3
- Use a third-party CA to emit different certificates to each component in the tool
- Configure replications in the Kafka brokers to eliminate single points of failure
- Configure the tool to track packets only adjacent to the cluster it pertains

- Integrate known attack vectors in the Kibana options so notifications are raised when detected
- Implement TLS both in Kafka and in Kubernetes, in every single possible component
- Modify the default config options that are known to generate vulnerabilities

Bibliography

[1] “Kubernetes,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[2] S. Moore, “Gartner Forecasts Strong Revenue Growth for Global Container Management Software and Services Through 2024,” Gartner, Inc., Sydney, 2020.

[3] A. Kohgadai, “The State of Kubernetes Security in 2022,” May 2022. [Online]. Available: https://www.redhat.com/es/blog/state-kubernetes-security-2022-1.

[4] S. P. Karode, *Monitoring Kubernetes Clusters With Dedicated Sidecar Network Sniffing Containers,* Dublin: Trinity College, 2020.

[5] T. Taylor, “5 Kubernetes security incidents and what we can learn from them,” TechGenix , 2020.

[6] L. Vaas, “Widespread Brute-Force Attacks Tied to Russia’s APT28,” Threatpost, 2021.

[7] C. Osborne, “Tesla cloud systems exploited by hackers to mine cryptocurrency,” ZDNet, 2018.

[8] B. Shafabakhsha, R. Lagerströmb and S. Hacksb, “Evaluating the Impact of Inter Process Communication in Microservice Architectures,” in *8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020)*, 2020.

[9] S. Kul and A. Sayar, “A Survey of Publish/Subscribe Middleware Systems for Microservice Communication,” in *5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2021.

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*, Springer, 2017, p. 195–216.

[11] M. Fowler, “Microservices,” [Online]. Available: https://martinfowler.com/articles/microservices.html.

[12] S. Newman, Building Microservices: Designing Fine-Grained Systems, O’Reilly Media, Inc., 2015.

[13] L. A. Vayghan, M. A. Saied, M. Toeroe and F. Khendek, “Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes,” in *IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019.

[14] T. Erl, Service-Oriented Architecture: Concepts, Technology & Design, Pearson Education India, 2005.

[15] D. FIRESMITH, “Virtualization via Containers,” Carnegie Mellon University, 2017. [Online]. Available: https://insights.sei.cmu.edu/blog/virtualization-via-containers/.

[16] Y. Yu, *OS-level Virtualization and Its Applications,* Stony Brook University, 2007.

[17] Docker Inc., “Docker,” Docker Inc., [Online]. Available: https://www.docker.com/.

[18] S. Miller, T. Siems and V. Debroy, “Kubernetes for Cloud Container Orchestration Versus Containers as a Service (CaaS): Practical Insights,” in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021.

[19] A. M., A. Dinkar, S. C. Mouli, S. B. and A. A. Deshpande, “Comparison of Containerization and Virtualization in Cloud Architectures,” in *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, Bangalore, 2021.

[20] A. Menychtas, A. Gatzioura and T. Varvarigou, “A Business Resolution Engine for Cloud Marketplaces,” in *IEEE Third International Conference on Cloud Computing Technology and Science*, Athens, 2011.

[21] M. Mousa, A. M. Bahaa-Eldin and M. Sobh, “Software Defined Networking concepts and challenges,” in *11th International Conference on Computer Engineering & Systems (ICCES)*, Cairo, 2016.

[22] A. S. Fundation, “Apache Kafka,” 2022. [Online]. Available: https://kafka.apache.org/.

[23] N. Garg, Apache Kafka, Packt Publishing, 2013.

[24] P. L. Noac'h, A. Costan and L. Bougé, “A performance evaluation of Apache Kafka in support of big data streaming applications,” in *IEEE International Conference on Big Data (Big Data)*, 2017.

[25] E. B.V., “Elastic,” 2022. [Online]. Available: https://www.elastic.co/.

[26] OpenSearch, “OpenSearch,” Amazon Web Services, 2022. [Online]. Available: https://opensearch.org/.

[27] D. B. Bose, A. Rahman and S. I. Shamim, "'Under-reported' Security Defects in Kubernetes Manifests," in *IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, 2021.

[28] Red Hat, Inc., "State of Kubernetes security report," 2022.

[29] D. W. Group, "IEEE 2675-2021 - IEEE Approved Draft Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package and Deployment," IEEE, 2021 .

[30] K. Carter, "Francois Raynaud on DevSecOps," *IEEE Software,* vol. 34, no. 5, pp. 93-96, 2017.

[31] V. Mohan and L. B. Othmane, "SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps," in *11th International Conference on Availability, Reliability and Security (ARES)*, Salzburg, 2016.

[32] K. Bilal, S. U. Khan, J. Kolodziej, L. Zhang, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang and D. Chen, "A Comparative Study of Data Center Network Architectures," in *European Conference on Modelling and Simulation (ECMS)*, 2012.

[33] H. QI, M. SHIRAZ1, J.-y. LIU, A. GANI, Z. A. RAHMAN and T. A. ALTAMEEM, "Data center network architecture in cloud computing: review, taxonomy, and open research issues," *Journal of Zhejiang University SCIENCE C,* vol. 15, no. 9, p. 776–793, 2014.

[34] D. R. MILLER, S. HARRIS, A. A. HARPER, S. VANDYKE and C. BLASK, Security Information and Event Management (SIEM) Implementation, McGraw-Hill Higher Education., 2011.

[35] Paloalto Networks, "What Is Network Segmentation?," PaloAlto, [Online]. Available: https://www.paloaltonetworks.com/cyberpedia/what-is-network-segmentation.

[36] F. Minna, B. Chandrasekaran, A. Blaise, F. Rebecchi and F. Massacci, "Understanding the Security Implications of Kubernetes Networking," *IEEE Security & Privacy,* vol. 19, pp. 46-56, 2019.

[37] Aqua, "Kubernetes Security Best Practices," Aqua, [Online]. Available: https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/.

[38] Sysdig, "Kubernetes Security 101: Fundamentals and Best Practices," [Online]. Available: https://sysdig.com/learn-cloud-native/kubernetes-security/kubernetes-security-101/.

[39] M. S. I. Shamim, F. A. Bhuiyan and A. Rahman, “XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices,” in *IEEE Secure Development (SecDev)*.

[40] G. King, “Best security practices: An overview,” in *23rd National Information Systems Security Conference*, Baltimore, 2000.

[41] J. Kaftzan, “Kubernetes Security Best Practices: Definitive Guide,” ARMO, 2022. [Online]. Available: https://www.armosec.io/blog/kubernetes-security-best-practices.

[42] Kubernetes, “Kubernetes Configuration: Secrets,” Kubernetes, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/.

[43] F. Fernando, *An Introduction to Kubernetes Security using Falco,* Falco, 2021.

[44] P. Authors, “Prometheus,” Linux Fundation, 2022. [Online]. Available: https://prometheus.io/.

[45] D. Sagi, “DNS Spoofing on Kubernetes Clusters,” 2019. [Online]. Available: https://blog.aquasec.com/dns-spoofing-kubernetes-clusters.

[46] N. Chako, “Attacking Kubernetes Clusters Through Your Network Plumbing: Part 1,” CYBERARC, 2020. [Online]. Available: https://www.cyberark.com/resources/threat-research-blog/attacking-kubernetes-clusters-through-your-network-plumbing-part-1.

[47] N. Chako, “Attacking Kubernetes Clusters Through Your Network Plumbing: Part 2,” CYBERARC, 2021. [Online]. Available: https://www.cyberark.com/resources/threat-research-blog/attacking-kubernetes-clusters-through-your-network-plumbing-part-2.

[48] Prisma, “| Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms,” Palo Alto Networks.

[49] Kubernetes, “Assigning Pods to Nodes,” Kubernetes, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[50] F. Carbonetti, “From dev to admin: an easy Kubernetes privilege escalation you should be aware of — the attack,” [Online]. Available: https://faun.pub/from-dev-to-admin-an-easy-kubernetes-privilege-escalation-you-should-be-aware-of-the-attack-950e6cf76cac.

[51] M. Panagiotis, *Attack methods and defenses on Kubernetes,* Piraeus: UNIVERSITY OF PIRAEUS, 2020.

[52] A. B. Bondi, “Characteristics of Scalability and Their Impact on Performance,” in *2nd international workshop on Software and performance*, Ontario, 200.

[53] STRIMZI, “Strimzi Overview guide,” STRIMZI, 2022. [Online]. Available: https://strimzi.io/docs/operators/in-development/overview.html.

[54] Kuberntes, “Example: Deploying PHP Guestbook application with Redis,” The Linux Foundation, 2022. [Online]. Available: https://kubernetes.io/docs/tutorials/stateless-application/guestbook/.

[55] P. Biondi, “About Scapy,” Scapy community, 2022. [Online]. Available: https://scapy.readthedocs.io/en/latest/introduction.html#about-scapy.

[56] G. KOSAKA, *Achieve and Enforce GDPR Compliance for Containers & Kubernetes,* NeuVector Inc., 2020.

[57] F. Dickson, “How and Why Kubernetes Complicates Security,” IDC Corporate, 2022.

A 1. Appendix

The project is composed of the following files organized as a file tree:

Files to start and shutdown the deployment of Guestbook Redis app in Minikube:

```
├── deploy.sh
├── deployments
│   ├── frontend-deployment.yaml
│   ├── redis-master-deployment.yaml
│   └── redis-slave-deployment.yaml
├── services
│   ├── frontend-service.yaml
│   ├── packets-api-frontend-service.yaml
│   ├── redis-master-service.yaml
│   ├── packets-api-redis-master-service.yaml
│   └── redis-slave-service.yaml
│   ├── packets-api-redis-slave-service.yaml
├── shutdown.sh
```

Files to generate the sidecar docker image with the Meteorshark sniffer and the Kafka producer:

```
├── Dockerfile
├── run.sh
├── sniffer
│   ├── meteorshark220801.py
│   └── snifferkafka220731.py
```

File that contains the Kafka consumer that indexes records in elastic:

```
├── monitoring-elastic
  ├── kafkaconsumerToElastic.py
```

A.1.1 deploy.sh

```
kubectl apply -f deployments/redis-master-deployment.yaml &&
kubectl get pods -A &&
kubectl apply -f deployments/redis-slave-deployment.yaml &&
kubectl get pods -A &&
kubectl apply -f deployments/frontend-deployment.yaml &&
kubectl apply -f services/redis-slave-service.yaml &&
kubectl get services -A  &&
kubectl apply -f services/redis-master-service.yaml &&
kubectl get service -A &&
kubectl apply -f services/frontend-service.yaml &&
kubectl get service -A &&
kubectl apply -f services/packets-api-frontend-service.yaml &&
kubectl get services -A
```

A.1.2 frontend-deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: frontend
  namespace: my-kafka-project
  labels:
    app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 1
  template:
    metadata:
      labels:
```

```
    app: guestbook
    tier: frontend
spec:
  containers:
    - name: php-redis
      image: gcr.io/google-samples/gb-frontend:v4
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
          # service launched automatically. However, if the cluster you are using
          # does not have a built-in DNS service, you can instead
          # access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns' line above, and
          # uncomment the line below:
          # value: env

      ports:
        - containerPort: 80
    - name: shell
      image: "packets-api:latest"
      imagePullPolicy: Never
      ports:
        - containerPort: 3000
      env:
        - name: KEY
          value: "frontend"
```

A.1.3 redis-master-deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-master
  namespace: my-kafka-project
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: k8s.gcr.io/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
         - containerPort: 6379
      - name: shell
        image: "packets-api:latest"
```

```
        imagePullPolicy: Never
        ports:
         - containerPort: 3000
        env:
         - name: KEY
           value: "leader"
```

A.1.4 redis-slave-deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-slave
  namespace: my-kafka-project
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v3
        resources:
```

```
      requests:
        cpu: 100m
        memory: 100Mi
    env:
    - name: GET_HOSTS_FROM
      value: dns
      # Using `GET_HOSTS_FROM=dns` requires your cluster to
      # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
      # service launched automatically. However, if the cluster you are using
      # does not have a built-in DNS service, you can instead
      # access an environment variable to find the master
      # service's host. To do so, comment out the 'value: dns' line above, and
      # uncomment the line below:
      # value: env
    ports:
      - containerPort: 6379
  - name: shell
    image: "packets-api:latest"
    imagePullPolicy: Never
    ports:
      - containerPort: 3000
    env:
      - name: KEY
        value: "follower"
```

A.1.5 frontend-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: my-kafka-project
  labels:
    app: guestbook
```

```
    tier: frontend
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30001
  selector:
    app: guestbook
    tier: frontend
```

A.1.6 packets-api-frontend-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: packets-api-frontend
  namespace: my-kafka-project
  labels:
    app: guestbook
    tier: frontend
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
    - port: 3000
      targetPort: 3000
```

```
    nodePort: 30002
  selector:
    app: guestbook
    tier: frontend
```

A.1.7 redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  namespace: my-kafka-project
  labels:
    app: redis
    role: master
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

A.1.8 packets-api-redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: packets-api-redis-master
  namespace: my-kafka-project
  labels:
```

```
    app: redis
    tier: backend
    role: master
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
    - port: 3000
      targetPort: 3000
      nodePort: 30003
  selector:
    app: redis
    role: master
    tier: backend
```

A.1.9 redis-slave-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  namespace: my-kafka-project
  labels:
    app: redis
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: redis
```

```
    role: slave
    tier: backend
```

A.1.10 packets-api-redis-slave-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: packets-api-redis-slave
  namespace: my-kafka-project
  labels:
    app: redis
    tier: backend
    role: slave
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
    - port: 3000
      targetPort: 3000
      nodePort: 30004
  selector:
    app: redis
    role: slave
    tier: backend
```

A.1.11 shutdown.sh

```
kubectl delete deployment frontend -n my-kafka-project &
kubectl delete deployment redis-master -n my-kafka-project &
```

```
kubectl delete deployment redis-slave -n my-kafka-project &
kubectl delete service    redis-master -n my-kafka-project &
kubectl delete service    redis-slave -n my-kafka-project &
kubectl delete service    frontend -n my-kafka-project &
kubectl delete service    packets-api-frontend -n my-kafka-project
```

A.1.12 run.sh

```
python sniffer/snifferkafka220731.py --filter="tcp" &
echo "sniffing started with kafka key:  $(printenv KEY)" &
tail -f /dev/null
```

A.1.13 meteorshark220801.py

```
#!/usr/bin/env python3

""" Library for integrating with a kafka producer
    https://github.com/thepacketgeek/meteorshark
"""
from datetime import datetime
from typing import Any, Dict, NamedTuple, Optional, Tuple

import requests
from scapy.all import (
    ARP,
    ByteField,
    Ether,
    Dot1Q,
    STP,
    Dot3,
    IP,
    IPv6,
    ICMP,
    TCP,
```

```
    UDP,
    Packet,
)

class ParsedPacket(NamedTuple):
    """ Temporary representation of a parsed packet
    """
    timestamp: int
    size: int
    src_ip: Optional[str] = None
    dst_ip: Optional[str] = None
    ttl: Optional[int] = None
    src_mac: Optional[str] = None
    dst_mac: Optional[str] = None
    app_protocol: Optional[
        str
    ] = None  # The highest level protocol included in the packet
    transport_protocol: Optional[str] = None
    src_port: Optional[int] = None
    dst_port: Optional[int] = None
    payload: Optional[str] = None

    def simple_to_api(self) -> Dict[str, Any]:
        """ Prepare packet for JSON formatting without payload """
        isotime =
datetime.fromtimestamp(self.timestamp).isoformat(timespec="milliseconds")

        return {
            "timestamp": isotime,
            "srcIP": self.src_ip,
            "dstIP": self.dst_ip,
            "L7protocol": self.app_protocol,
            "size": self.size,
            "ttl": self.ttl,
```

```
            "srcMAC": self.src_mac,
            "dstMAC": self.dst_mac,
            "L4protocol": self.transport_protocol,
            "srcPort": self.src_port,
            "dstPort": self.dst_port,
        }

    def to_api(self) -> Dict[str, Any]:
        """ Prepare packet for JSON formatting """
        return {
            "timestamp": self.timestamp,
            "srcIP": self.src_ip,
            "dstIP": self.dst_ip,
            "L7protocol": self.app_protocol,
            "size": self.size,
            "ttl": self.ttl,
            "srcMAC": self.src_mac,
            "dstMAC": self.dst_mac,
            "L4protocol": self.transport_protocol,
            "srcPort": self.src_port,
            "dstPort": self.dst_port,
            "payload": self.payload,
        }

def clean_payload(pkt: Packet) -> str:
    """ Clean up packet payload from Scapy output

    """
    return pkt.layers()[-1].summary()

def get_ips(pkt: Packet) -> Tuple[Optional[str], Optional[str]]:
    if pkt.haslayer(ARP):
        return (pkt[ARP].psrc, pkt[ARP].pdst)
```

```
    if pkt.haslayer(IP):
        return (pkt[IP].src, pkt[IP].dst)
    if pkt.haslayer(IPv6):
        return (pkt[IPv6].src, pkt[IPv6].dst)

    return (None, None)

def get_macs(pkt: Packet) -> Tuple[Optional[str], Optional[str]]:
    if pkt.haslayer(Ether):
        return (pkt[Ether].src, pkt[Ether].dst)
    return (None, None)

def get_ports(pkt: Packet) -> Tuple[Optional[str], Optional[str]]:
    if pkt.haslayer(TCP):
        return (pkt[TCP].sport, pkt[TCP].dport)
    if pkt.haslayer(UDP):
        return (pkt[UDP].sport, pkt[UDP].dport)
    return (None, None)

def get_transport_protocol(pkt: Packet) -> Optional[str]:
    pass

def get_app_protocol(pkt: Packet) -> Optional[str]:
    if pkt.haslayer(ARP):
        return "ARP"
    if pkt.haslayer(ICMP):
        return "ICMP"
    else:
        for layer in reversed(pkt.layers()):
            name = layer.__name__
            if name != "Raw":
                return name
    return pkt.lastlayer().__name__
```

```
def get_payload(pkt: Packet) -> Optional[str]:
    """ Get the payload of the packet as a string """
    return f"{pkt.payload!r}"

def get_size(pkt: Packet) -> int:
    return len(pkt)
def get_ttl(pkt: Packet) -> Optional[int]:
    if IP in pkt:
        return pkt.ttl

    if IPv6 in pkt:
        return pkt.hlim

    for layer in reversed(pkt.layers()):
        ttl = getattr(pkt[layer.__name__], "ttl", None)
        if ttl:
            return ttl

def parse_packet(pkt: Packet) -> ParsedPacket:
    src_ip, dst_ip = get_ips(pkt)
    src_mac, dst_mac = get_macs(pkt)
    src_port, dst_port = get_ports(pkt)

    return ParsedPacket(
        timestamp=int(datetime.now().timestamp()),
        src_ip=src_ip,
        dst_ip=dst_ip,
        app_protocol=get_app_protocol(pkt),
        size=get_size(pkt),
        ttl=get_ttl(pkt),
        src_mac=src_mac,
        dst_mac=dst_mac,
        transport_protocol=get_transport_protocol(pkt),
```

```
        src_port=src_port,
        dst_port=dst_port,
        payload=get_payload(pkt),
    )
```

A.1.14 snifferkafka220731.py

```
#!/usr/bin/env python3

"""
Program that sniffs tcp packets and publishes them in Kafka. Packets are published
as json objects

"""
import json
from kafka import KafkaProducer
import argparse
import logging
import os
from scapy.all import sniff, Packet
from meteorshark220801 import parse_packet, ParsedPacket

log = logging.getLogger(__name__)
log.addHandler(logging.StreamHandler())
log.setLevel(logging.INFO)

class KafkaTcpProducer:

    def __init__(self, bootstrap_servers, topic, key):
        self.bootstrap_servers = bootstrap_servers
        self.topic = topic
        self.key = key
        print(self.bootstrap_servers, self.topic, self.key)
```

```
        self.producer = KafkaProducer(bootstrap_servers=self.bootstrap_servers,
key_serializer=str.encode,
                                      value_serializer=lambda v: json.dumps(v).encode('utf-8'))

    def produce_json_pkt(self, parsed_pkt: ParsedPacket):
        """ publish JSON packet in kafka """
        json_pkt = parsed_pkt.simple_to_api()
        json_pkt["owner"] = key
        self.producer.send(topic=self.topic, key=self.key, value=json_pkt)

""" main program """

def produce_pkt(raw_pkt: Packet):
    """ Get the raw packet and publish it as a JSON packet in kafka """
    parsed_pkt = parse_packet(raw_pkt)
    producer.produce_json_pkt(parsed_pkt)

def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--count",
        type=int,
        help="The number of packets to sniff (integer). 0 (default) is indefinite count.",
        default=0,
    )
    parser.add_argument("--filter", help="The BPF style filter to sniff with.")
    parser.add_argument("--debug", help="View debug level logs",
action="store_true")
    return parser.parse_args()

if __name__ == "__main__":
    args = get_args()
    if args.debug:
        log.setLevel(logging.DEBUG)
```

```
    key = os.environ.get('KEY')
    if not isinstance(key, str):
        key = " key not Defined"
    producer = KafkaTcpProducer("192.168.49.2:31751", "tcp-topic", key)
    log.info("Started kafka tcp producer")

    log.info("Sniffing {} packets... Ctrl + C to stop sniffing".format(args.count))
    # Start sniffing some tcp packets
    sniff(filter="tcp", prn=produce_pkt, count=args.count, store=0)
```

A.1.15 kafkaconsumerToElastic.py.py

```
import datetime
import json

from elasticsearch import Elasticsearch
from confluent_kafka import Consumer

INDEX_NAME = "tcp-log220804"

class Elastic:

    def __init__(self):
        self.host = "localhost"
        self.port = 9200
        self.es = None
        self.connect()
        self.INDEX_NAME = INDEX_NAME
        self.topic = "data_log"

    def connect(self):
        self.es = Elasticsearch([{'host': self.host, 'port': self.port}])
        if self.es.ping():
            print("ES connected successfully")
```

```
        else:
            print("Not connected")

    def create_index(self):
        if self.es.indices.exists(self.INDEX_NAME):
            print("deleting '%s' index..." % self.INDEX_NAME)
            res = self.es.indices.delete(index=self.INDEX_NAME)
            print(" response: '%s'" % res)
            request_body = {
                "settings": {
                    "number_of_shards": 1,
                    "number_of_replicas": 0
                }
            }
            print("creating '%s' index..." % self.INDEX_NAME)
            res = self.es.indices.create(index=self.INDEX_NAME, body=request_body,
ignore=400)
            print(" response: '%s'" % res)

    def push_to_index(self, message):
        try:
            response = self.es.index(
                index=INDEX_NAME,
                doc_type='log',
                body=message
            )
            # print("Write response is :: {}\n\n".format(response))
            print(message)

        except Exception as e:
            print("Exception is :: {}".format(str(e)))
```

```
class KafkaConsumer:

    def __init__(self, topic: str, group: str, broker: str):
        self.jsonmsg = None
        self.group = group
        self.topic = topic
        self.broker = broker
        self.conf = {'bootstrap.servers': self.broker, 'group.id': self.group,
'auto.offset.reset': 'earliest'}
        self.consumer = Consumer(self.conf)
        self.consumer.subscribe([self.topic])

    def read_messages(self):
        """returns a json msg"""
        try:
            rawmsg = self.consumer.poll()
            if rawmsg is None:
                return 0
            else:
                # convert to string raw msg
                strmsg = rawmsg.value().decode('utf-8')
                self.jsonmsg = json.loads(strmsg)
                return self.jsonmsg
        except Exception as e:
            print("Exception during reading message :: {}".format(e))
            return 0

    def close(self):
        self.consumer.close()

if __name__ == '__main__':
    es_obj = Elastic()
    es_obj.create_index()
    consumer = KafkaConsumer('tcp-topic', 'group4', '192.168.49.2:31751')
```

```
packets = 0
limit = 5000
try:
    while packets < limit:
        jsonmsg = consumer.read_messages()
        if jsonmsg is None:
            continue
        else:
            packets = packets + 1
            es_obj.push_to_index(jsonmsg)
finally:
    consumer.close()
```