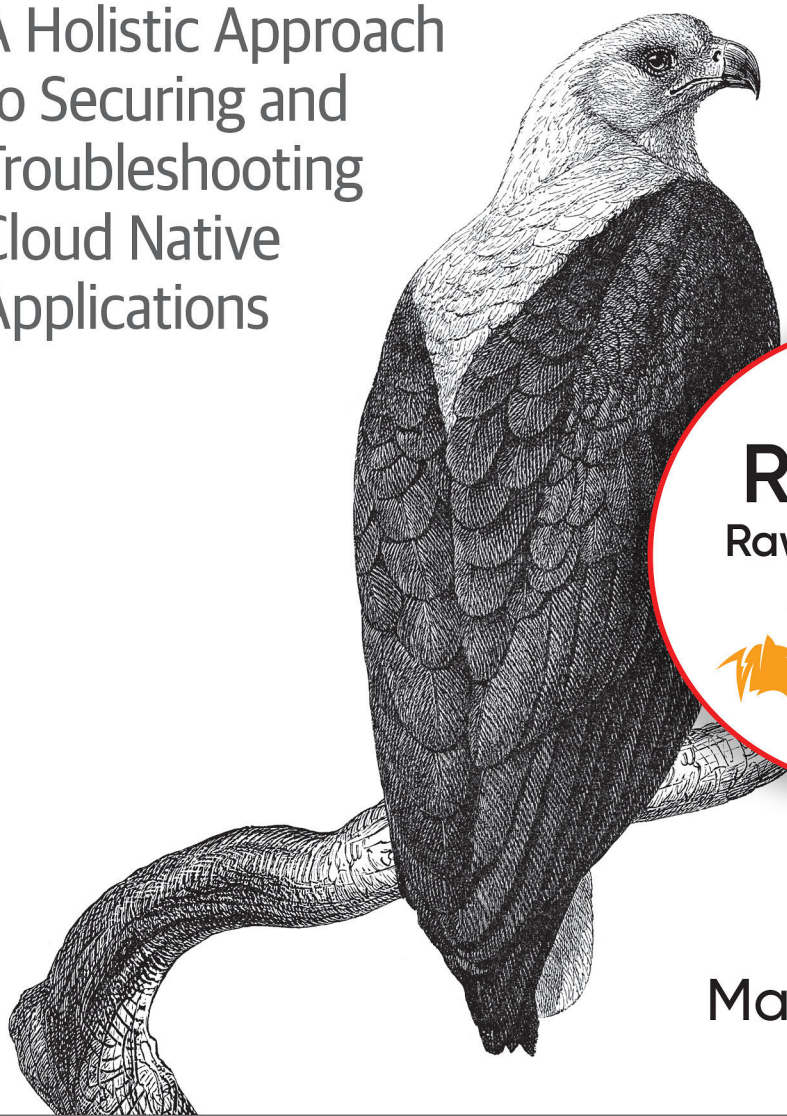
O'REILLY®

Kubernetes Security and Observability

A Holistic Approach to Securing and Troubleshooting Cloud Native Applications



Early Release
Raw & Unedited

Sponsored by TIGERA

Alex Pollitt &
Manish Sampat

TIGERA
Inventor and maintainer of Project Calico

PROJECT CALICO

Linux eBPF
Standard Linux
Windows HNS

Kubernetes-native security and observability as code.

Modern security and observability for distributed cloud-native applications.

NORTH-SOUTH CONTROLS	EAST-WEST CONTROLS	SECURITY and COMPLIANCE	OBSERVABILITY
Control access to and from services outside the cluster.	Micro-segmentation at the application, container/VM, and host levels.	Enterprise-specific. Encryption. Intrusion detection and prevention.	Rich contextual info as a service graph. Automated diagnostics.

Get started with a free trial of Calico Cloud

tigera.io/tigera-products/cloud-trial

Trusted by Innovators

AT&T

DISCOVER

GLOBUS

MERCK

L3HARRIS™
FAST. FORWARD.

MINDBODY

servicenow

REALPAGE
OUTPERFORM

Kubernetes Security and Observability

A Holistic Approach to Securing and Troubleshooting Cloud Native Applications

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Alex Pollitt and Manish Sampat

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes Security and Observability
by Alex Pollitt and Manish Sampat

Copyright © 2022 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Virginia Wilson and John Devins
Production Editor: Kate Galloway
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

February 2022: First Edition

Revision History for the Early Release
2021-04-06: First Release
2021-04-12: Second Release
2021-05-03: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098107109* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Kubernetes Security and Observability*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Tigera. See our statement of editorial independence.

978-1-098-10895-3

Table of Contents

Preface

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Kubernetes is not secure by default. Existing approaches to enterprise and cloud security are challenged by the dynamic nature of Kubernetes and the often associated goal of increased organizational agility. Successfully securing, observing and troubleshooting mission critical microservices in this new environment requires a holistic understanding of the breadth of considerations, including organizational challenges, how new cloud native approaches can help meet the challenges, and the new best practices and how to operationalize them.

While there is no shortage of resources that provide education on a particular aspect of Kubernetes, navigating through these ad hoc resources and formulating a comprehensive security and observability strategy can be a daunting task, and in many cases leads to security gaps that significantly undermine the desired security posture.

That's why we wrote this book—to guide you towards a holistic security and observability strategy across the breadth of these considerations, and to give you best practices and tools to help you accelerate your Kubernetes adoption.

Over the years working at Tigera and building Calico, a networking and security tool for Kubernetes, we have gotten to see the Kubernetes user journey up close. We have seen many users focus on getting their workloads deployed in Kubernetes without thinking through their security or observability strategy, and then struggle as they try

to understand how to secure and observe such a complex distributed system. Our goal with this book is to help minimize this pain as much as possible by sharing with you what we've learned. We mention a number of tool examples throughout, and Calico is among them. We believe that Calico is an excellent and popular option, but there are many good tools to choose from. Ultimately, only you can decide which is best for your needs.

Let's start by looking at what a typical adoption journey looks like.

The Stages of Kubernetes Adoption

Any successful Kubernetes adoption journey follows three distinct stages.

1. The Learning Stage. As a new user, you begin by learning how Kubernetes works, setting up a sandbox environment, and starting to think about how you can use Kubernetes in your environment. In this stage you want to leverage the online Kubernetes resources available and use open source technologies.
2. The Pilot / Pre-Production Stage. Once you familiarize yourself with Kubernetes and understand how it works, you start thinking about a high level strategy to adopt Kubernetes. In this stage you typically start a pilot project to set up your cluster and onboard a couple of applications. As you progress in this stage, you will have an idea about which platforms you're going to use, and whether they will be on-premise or in the cloud. If you choose cloud, you will decide whether to host the cluster yourself or leverage a managed Kubernetes service from a cloud provider. You also need to think about strategies to secure your applications. By this time, you would have realized that Kubernetes is different due to its declarative nature. This means that the platform abstracts a lot of details about the network, infrastructure, host, etc., and therefore makes it very easy for you to use the platform for your applications. Because of this, the current methods you use to secure your applications, infrastructure and networks simply do not work, so you now need to think about security that is native to Kubernetes.
3. The Production Stage. By this point you have completed your pilot project and successfully onboarded a few applications. Your focus is on running mission-critical applications in production, and on considering whether to migrate most of your applications to Kubernetes. In this stage you need to have a detailed plan for security, compliance, troubleshooting and observability in order to safely and efficiently move your applications to production and realize all the benefits of the Kubernetes platform.

The popularity and success of Kubernetes as a platform for container based applications has many people eager to adopt it. In the past couple of years, there has been an effort by managed Kubernetes service providers to innovate and make adoption easier. New users may be tempted to go past the learning and pilot stages in order to get to the "Production Stage" quickly. We caution against skipping due diligence. You must consider security and observability as critical first steps before you onboard mission-critical applications to Kubernetes; your Kubernetes adoption is incomplete and potentially insecure without them.

Who This Book Is For

This book is for a broad range of Kubernetes practitioners who are in the Pilot/Pre-Production stage of adoption. You may be a platform engineer, or part of the security or DevOps team. Some of you are the first in your organization to adopt Kubernetes and want to do security and observability right from the start. Others are helping to establish best practices within an organization that has already adopted Kubernetes but has not yet solved the security and observability challenges Kubernetes presents. We assume you have basic knowledge of Kubernetes—what it is, and how to use it as an orchestration tool for hosting applications. We also assume you understand how applications are deployed, and about their distributed nature in a Kubernetes cluster.

Within this broad audience, there are many different roles. Here is a non exhaustive list of teams that help design and implement Kubernetes-based architectures, that will find value in this book.

The Platform team

The platform engineering team is responsible for the design and implementation of the Kubernetes platform. Many enterprises choose to implement a "container as a service platform" (CaaS) strategy. This is a platform that is used enterprise wide to implement container based workloads. The platform engineering team is responsible for the platform components and provides this as a service to application teams. This book helps you understand the importance of securing the platform and best practices to help secure the platform layer. This way you can provide application teams a way to onboard applications on a secure Kubernetes platform. It helps you learn how to manage the security risk of new applications to the platform.

The Networking team

The networking team is responsible for integrating Kubernetes clusters in an enterprise network. We see these teams play different roles in an on-premise deployment of Kubernetes and in a cloud environment where Kubernetes clusters are self hosted

or leverage a managed Kubernetes service. You will understand the importance of network security and how to build networks with a strong security posture. Best practices for exposing applications outside the Kubernetes platform as well as network access for applications to external networks are examples of topics covered in this book. You will also learn how to collaborate with other teams to implement network security to protect elements external to Kubernetes from workloads inside Kubernetes.

The Security team

The security team in enterprises is the most impacted by the movement toward cloud native applications. Cloud native applications are applications designed for cloud environments and are different from traditional applications. As an example these applications are distributed across the infrastructure in your network. This book will help you understand details about how to secure a Kubernetes platform that is used to host applications. It will provide you a complete view of how to secure mission critical workloads. You will learn how to collaborate with various teams to effectively implement security in the new and different world of Kubernetes.

The Compliance team

The compliance team in an enterprise is responsible for ensuring operations and processes in an organization meet the requirements of compliance standards adopted by an organization. You understand how to implement various compliance requirements and how to monitor ongoing compliance in a Kubernetes based platform. Note that we will not cover detailed compliance requirements and various standards but we will provide you with strategies, examples about tools to help you meet compliance requirements.

The Operations team

The operations team is the team of developers/tools/operations engineers responsible for building and maintaining applications. They are also known as "DevOps" or Site Reliability Engineers (SREs). They ensure that applications are onboarded and meet the required Service Level Agreements (SLAs). In this book you will learn about your role in securing the Kubernetes cluster and collaboration with the security team. We will cover the concept of "Shift Left" Security, which says security needs to happen very early in the application development lifecycle. Observability in a Kubernetes platform means the ability to infer details about the operation of your cluster by viewing data from the platform. This is the modern way of monitoring a distributed application and you will learn how to implement observability and its importance to security.

What You Will Learn

In this book you will learn how to think about security as you implement your Kubernetes strategy. For example, building applications, building infrastructure to host applications, deploying applications and running applications. We will present security best practices for each of these with examples and tools to help you secure your Kubernetes platform. We will cover how to implement auditing, compliance, and other enterprise security controls like encryption.

You will also learn best practices with tools and examples that show you how to implement observability, and that demonstrate its relevance to security and troubleshooting. This enhanced visibility into your Kubernetes platform will drive actionable insights relevant to your unique situation.

By the end you will be able to implement these best practices, and make informed decisions about your infrastructure, networking and network security choices.

CHAPTER 1
Security Strategy

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In this chapter, we will cover a high level overview of how you can build a security strategy for your Kubernetes implementation. We will cover each of these concepts in more detail in subsequent chapters in the book. You need to think about a security strategy when you are in the pilot/pre production phase of your Kubernetes journey. If you are part of the security team, this chapter is very important for you. If you are part of the network, platform or the application teams, this chapter shows how you can be a part of the security strategy and discuss the importance of collaboration between the security, platform and application teams. We will cover the following concepts that will guide you with your security strategy.

- How securing Kubernetes is different from traditional security methods
- The lifecycle of deploying applications (workloads) in a Kubernetes cluster and best practices for each stage.

- Examples of well known security frameworks and how you can use them in your security strategy

Security for Kubernetes - a new and different world

In this section we'll highlight how Kubernetes is different, and why traditional security methods do not work in a Kubernetes implementation.

As workloads move to the cloud, Kubernetes is the most common orchestrator for managing them. The reason Kubernetes is popular is its declarative nature, it abstracts infrastructure details and allows users to specify the workloads they want to run and the desired outcomes. The application team does not need to worry about how workloads are deployed and where workloads are run and other details like networking, they just need to set up configurations in Kubernetes to deploy their applications..

Kubernetes achieves this abstraction by managing the workload creation, shutdown and restart. In a typical Kubernetes implementation, a workload can be scheduled on any available resource (physical host or a virtual machine) in a network based on the workloads' requirements, known as a Kubernetes cluster. Kubernetes monitors the status of workloads (workloads are deployed as pods in Kubernetes) and takes corrective action as needed (e.g. restart unresponsive nodes) . Kubernetes manages all networking necessary for pods (and hosts) to communicate with each other. You have the option to decide the networking technology being used by selecting from a set of supported network plugins. While there some configuration options for the network-plugin, you wil not be able to directly control networking behavior (IP address assignment or in typical configurations where the node is scheduled etc).

Kubernetes is a different world for security teams. Traditional methods would be to build a "network of machines" and then onboard workloads (applications). As a part of onboarding, the process was to assign IPs, update networking as needed, define and implement network access control rules. After these steps, the application was ready for users. This process ensured that security teams had a lot of control, and they could onboard and secure applications with ease. The applications were easy to secure as applications were static in terms of assigned IPs, where they were deployed etc.

In the Kuberntes world, workloads are built as container images and are deployed in a Kubernetes cluster using a configuration file (yaml). This is typically integrated in the development process, and most development teams use continuous integration (CI) and continuous delivery (CD) to ensure speedy and reliable delivery of software. What this means is that the security team has very limited visibility into the impact of each application change on the security of the cluster. Adding a security review step to this process is counterproductive as the only logical place to add that would be

when the code is being committed. The development process after that point is automated, and disrupting that would conflict with the CI/CD model. So how can you secure workloads in this environment?

In order to understand how to secure workloads in Kubernetes, it is important to understand the various stages that are a part of deploying a workload.

Deploying a workload in Kubernetes - Security at each stage.

In the previous section, we described the challenge of securing applications that are deployed using the CI/CD pipeline. This section describes the lifecycle of workload deployment in a Kubernetes cluster and explains how secure each stage. The three stages of workload deployment are the build, deploy and runtime stages. Unlike traditional client-server applications where an application existed on a server (or a cluster of servers), applications in a Kubernetes deployment are distributed, and the Kubernetes cluster network is used by applications as a part of normal operation of the application.

- You need to consider security best practices as workloads and infrastructure are built, this is important due to the fact that applications in Kubernetes are deployed using the CI/CD pipeline.
- You need to consider security best practices when a Kubernetes cluster is deployed and applications are onboarded.
- Finally applications use the infrastructure and the Kubernetes cluster network for normal operation and you need to consider security best practices for application runtime.

Figure 1-1 illustrates the various stages and aspects to consider when securing workloads in a Kubernetes environment.

Figure 1-1. Workload deployment stages and security at each stage

As you can see in Figure 1-1, the three stages that are a part of deploying a workload in Kubernetes are the build, deploy and runtime stage. The boxes below each stage describe various aspects of security that you need to consider for each stage.

- The build stage is the stage where you create (build) software for your workload (application) and build the infrastructure components (host or virtual machines) to host applications. This stage is part of the development cycle and in most cases the development team is responsible for this stage.In this stage you consider security for the CI/CD pipeline, security for image repositories, scanning images for vulnerabilities, hardening the host operating system. You need to ensure that you implement best practices to secure the image registry and avoid compromise of images in the image registry. This is generally implemented by securing access

to the image registry,a lot of users use private registries and do not allow images from public registries. Finally you need to consider best practices for secrets management, secrets are like password that allow access to resources in your cluster. We will cover these topics in detail in Chapter 3. We recommend that when you consider security for this stage, you should collaborate with the security team so that security at this stage is aligned with your overall security strategy.

- The next stage is the deploy stage where you set up the platform that runs your Kubernetes deployment and deploy workloads. In this stage you need to think about the security best practices for configuring your Kubernetes cluster, for providing external access to applications running inside your Kubernetes cluster. In this stage, you need to consider security controls like policies to limit access to workloads (Pod Security Policies), network policy to control application access to the platform components, role based access control (RBAC) for access to resources (for example, service creation, namespace creation, adding/changing labels to pods). In most enterprises the platform team is responsible for this stage. As a member of the platform team you need to collaborate with both the development and the security teams to implement your security strategy.
- The final stage is the runtime stage where you have deployed your application and it is operational. In this stage you need to think about network security, which involves controls using network policy, threat defense which is using techniques to detect and prevent malicious activity in the cluster and enterprise security controls like compliance, auditing and encryption. The security team is responsible for this stage of the deployment. As a member of the security you need to collaborate with the platform team and the development team as you design and implement runtime security. Please note that collaboration between teams (development, platform and security) is very important for building an effective security strategy. We recommend that you ensure all these teams are aligned,

Please note that unlike traditional security strategies where security is enforced at a vantage point (like the perimeter) in case of a Kubernetes cluster, you need to implement security at each stage. Another thing to note is that all teams involved (Application, Platform and the security) play a very important role in implementing security, so the key to implementing a successful strategy is collaboration between teams. Remember, security is a shared responsibility. Let's explore each stage and the techniques you can use to build your strategy.

Build Time Security: Shift Left

This section will guide you through various aspects of build time security with examples.

Image Scanning

During this stage, you need to ensure that applications do not have any known unpatched critical or major vulnerabilities as disclosed as Common Vulnerability Enumerations (CVEs) in the National Vulnerabilities Database (NVD), and that the application code and dependencies are scanned for exploits and vulnerable code segments. The images that are built and delivered as containers are then scanned for unpatched critical or major vulnerabilities disclosed as Common Vulnerability Enumerations (CVEs). This is usually done by checking the base image and all packages that are part fo the base image against a database that tracks vulnerable packages. In order to implement scanning there are several tools both open source and commercial that are available to you. For e.g Whitesource, Snyk, trivy, anchor and even cloud providers like Google offer scanning of container images. We recommend that you select a scanning solution that understands how containers are built and scans not only the operating system on the host, but also scans base images for containers. Given the dynamic nature of Kubernetes deployments it is very important for you to secure the CI/CD pipeline, code and image scanning needs to be a part of the pipeline, and images being delivered from the image registry must be checked for compromise. You need to ensure access to the registry is controlled to avoid compromise. The popular term used to describe this stage is shifting security left towards the development team, also known as "shift left" security.

Host operating system hardening

During this stage you must ensure that the application being deployed is restricted to having the required privileges on the host where it is deployed. In order to achieve this, you should use a hardened host operating system that supports controls to enable restricting applications to only necessary privileges like system calls and file system access. This allows you to effectively mitigate attacks related to "privilege escalation", where a vulnerability in the software being deployed in a container is used to gain access to the host operating system.

Minimizing the attack surface: Base container images

We recommend you review the composition of the container image and minimize software packages that make up the base image to include only packages that are absolutely necessary for your application to run. In Dockerfile-based container images, you can start with a parent image and then add your application to the image to create a container image. One example is that you start building a base image in Docker by using the "FROM scratch" directive. This will create a minimal image, you can then add your application and required packages. This will give you complete control of the composition of your container images and it will also help with CVE management as you do not need to worry about patching CVEs in packages in a container image that aren't required by your application. In case building a scratch image

is not a viable option for you, you can consider starting with distroless images (a slimmed down Linux distribution image) or an alpine minimal image as base images for your container.

The techniques described above will help you design and implement your build time security strategy. As a part of the development team, you will be responsible to design and implement build time security in collaboration with the platform and the security teams to ensure that the build time security is aligned with overall security strategy. We caution against believing the myth that "shift left" security is your whole security strategy. It is incorrect and a naive approach to securing workloads. There are several other important aspects, deploy and runtime security that need to be considered as a part of your security strategy.

Deploy Time Security

The next stage in securing workloads is to secure the deployment. To accomplish this you need to harden your Kubernetes cluster where the workloads are deployed. You will need a detailed review of the Kubernetes cluster configuration to ensure that it is aligned with security best practices. You need to start by building a trust model for various components of your cluster. A trust model is a framework where you review the threat profile and define mechanisms to respond to a threat profile. You should leverage tools like Role Based Access Control (RBAC), Label taxonomies, Label governance and admission controls to design and implement the trust model. These are mechanisms to control access to resources and controls and validation applied at resource creation time. These topics are covered in detail in Chapters 3, 4, and 4. The other critical components in your cluster are the Kubernetes datastore and Kubernetes API server and you need to pay close attention to details like access control and data security when you design the trust model for these components. We recommend you use strong credentials, Public Key Infrastructure (PKI) for access and Transport Layer Security (TLS) for data in transit encryption. This topic is covered in detail in Chapter 2.

You should think of the Kubernetes cluster where mission critical workloads are deployed as an entity and design a trust model for the entity. This would require you to review security controls at the perimeter. This will be challenging due the Kubernetes deployment architectures; we will cover this in the next section. For now, let's assume the current products that are deployed for at the perimeter like web access control gateways and Next Generation Firewalls, are not aware of Kubernetes architecture. We recommend you tackle this by building integrations with these devices. This will make these devices aware of the Kuberenetes cluster context so they can be effective in applying security controls at the perimeter. This way you can create a very effective security strategy where the perimeter security devices work in conjunction with security implemented inside your Kubernetes cluster. As an example you need to make these devices aware of the identity of your workloads (e..g IP address,

TCP/UDP port etc) as these devices can effectively protect the hosts that make up your Kubernetes cluster but in most cases cannot distinguish between workloads running on a single host, If you're running in a cloud provider environment, you can use the concept of security groups which are virtual firewalls that allow access control to a groups of nodes (e.g. EC2 instances in AWS) that host workloads in the cloud provider environment. Security groups are more aligned with the Kubernetes architecture than traditional firewalls and security gateways; however, even security groups are not aware of the context for workloads running inside the cluster.

To summarize, when you consider deploy time security you need to implement a trust model for your Kubernetes cluster and build an effective integration with perimeter security devices that protect your cluster.

Runtime Security

Now that you have a strategy in place to secure the build and deploy stages, you need to think about runtime security. The term runtime security is used for various aspects of securing a Kubernetes cluster, for example on a host running software and any configuration that protects the host and workloads from unauthorized activity (e.g. system calls, file access) is also called runtime security. Chapter 4 will cover host and workload runtime security in detail In this section we will focus on the security best practices needed to ensure secure operation of the Kubernetes cluster network. Kubernetes is an orchestrator that deploys workloads and applications across a network of hosts. You must consider network security as a very important aspect of runtime security.

Kubernetes promises increased agility and more efficient use of compute resources compared to static partitioning and provisioning of servers or VMs. It does this by dynamically scheduling workloads across the cluster taking into account the resource usage on each node, and connecting workloads together on a flat network. By default, when a new workload is deployed the corresponding pod could be scheduled on any node in the cluster, with any IP address within the pod CIDR range. If the pod is later rescheduled elsewhere then it will normally get a different IP address. This means that pod IP addresses need to be treated as ephemeral. There is no long term or special meaning associated with pod IP addresses or of its location within the network.

Now consider traditional approaches to network security. Historically in enterprise networks, network security was implemented using security appliances (or virtual version of appliances) such as firewalls and routers. The rules being enforced by these appliances were often based on a combination of the physical topology of the network and allocating specific IP address ranges to different classes of workloads.

As Kubernetes is based on a flat network, without any special meaning of pod IP addresses, very few of these traditional appliances are able to provide any meaningful workload-aware network security and instead have to treat the whole cluster as a sin-

gle entity. In addition, in the case of east-west traffic between two pods hosted on the same node, the traffic does not even go via the underlying network, so these appliances won't see this traffic at all and are essentially limited to north-south security, which is securing traffic entering the cluster from external sources and traffic originating inside the cluster destined to sources outside the cluster.

Given all of this, it should be clear that Kubernetes requires a new approach to network security. This new approach needs to cover a broad range of considerations, including:

- New ways to enforce network security (which workloads are allowed to talk to which other workloads) which do not rely on special meaning of IP addresses or network topology, and works even if the traffic does not traverse the underlying network. Kubernetes network policy is designed to meet these needs.
- New tools to help manage network policies that support the new development processes and the desire for microservice to bring increased organizational agility, such as policy recommendations, policy impact previews, and policy staging.
- New ways to monitor and visualize network traffic, covering both cluster scoped holistic views (e.g how do you easily view the overall network and network security status of the cluster) and targeted topographic views to drill down across a sequence of microservices to to help troubleshoot or diagnose application issues.
- New ways of implementing intrusion detection & threat defense, including policy violation alerting, network anomaly detection, and integrating threat feeds.
- New remediation workflows, so potentially compromised workloads can be quickly and safely isolated during forensic investigation.
- New mechanisms for auditing configuration and policy changes for compliance.
- New mechanisms for auditing configuration and policy changes, and Kubernetes aware network flow logs to meet compliance requirements (since traditional network flow logs are IP based, so have little long term meaning in the context of Kubernetes).

We will review an example of a typical Kubernetes deployment in an enterprise to understand these challenges. Figure 1-2 is a representation of a common deployment model for kubernetes and microservices in a multi-cloud environment. A multi-cloud environment is an environment where an enterprise deploys Kubernetes in more than one cloud provider (e.g. Amazon Web services, Google Cloud etc). A hybrid cloud environment is an environment where an enterprise has a Kubernetes deployment in at least one cloud provider environment and a Kubernetes deployment on premise in their datacenter. Most Enterprises have a dual cloud strategy and will have clusters running in Amazon Web Services (AWS), Microsoft Azure or Google Cloud, more enterprises also have some legacy applications running in their data cen-

ter. Workloads in the data center will likely be behind a security gateway that filters traffic coming in through the perimeter. Microservices running in these Kubernetes deployments are also likely to have one or more dependencies on:

- Other cloud services like AWS RDS or Azure DB
- 3rd party api endpoints like twilio
- SaaS services like SalesForce, Zuora
- Databases or legacy apps running inside the data center.

Workloads in the data center will likely be behind a security gateway that filters traffic coming in through the perimeter.

Observability in Kubernetes is the ability to derive actionable insights about the state of Kubernetes from metrics collected (more on this below). While observability has other applications like monitoring and troubleshooting, it is important in the context of network security. Observability concepts applied to flow logs correlated with other Kubernetes metadata (pods labels, policies, namespaces etc) are used to monitor (and then secure) communications between pods in a Kubernetes cluster, detect malicious activity by comparing IPs addresses with known malicious IP addresses and use Machine learning based techniques to detect malicious activity. These topics are covered in the section below. As you can see in Figure 1-2, the Kubernetes deployment poses challenges due to silos of data in each cluster and potential loss of visibility to associate a workload in one cluster talking to a workload in another cluster or to an external service.

Enterprise Kubernetes deployment

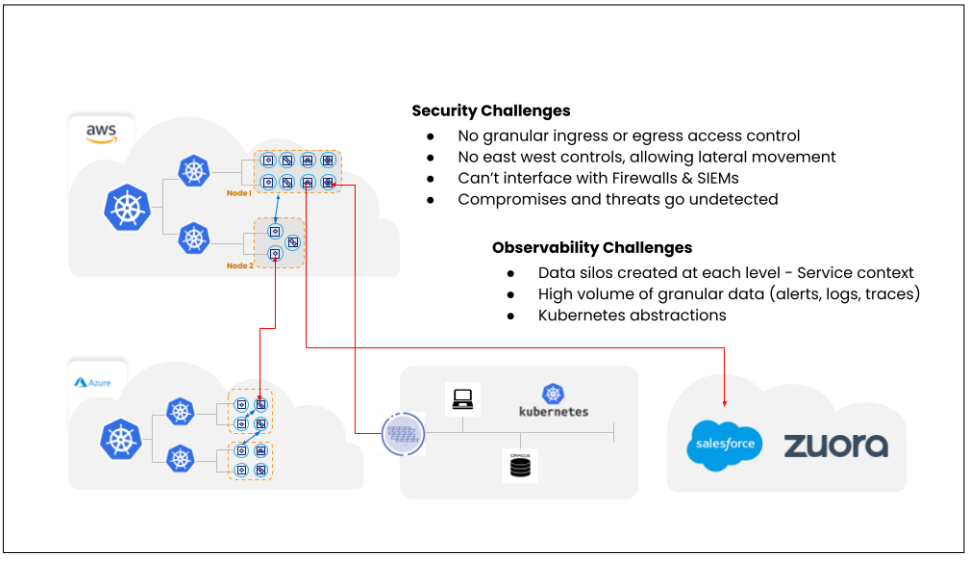


Figure 1-2. Example of a Kubernetes deployment in an enterprise

As you can see in Figure 1-2, the footprint of a microservices application typically extends beyond the Virtual Private Cloud (VPC) boundaries, and securing these applications requires a different approach from the traditional perimeter security based approach. The approach is a combination of Network Security controls, Observability, Threat Defense and Enterprise Security controls. We will cover each of these next.

Network Security Controls

Native security controls available from cloud providers (for example - AWS Security Groups, Azure Network Security Groups) or security gateways (for example, NG firewalls) on the perimeter of the VPC or data center do not understand the identity of a Microservice inside a Kubernetes cluster. For example, you can not filter traffic to or from a kubernetes pod or service with your Security Group rules or firewall policies. Additionally, by the time traffic from a pod hits a cloud provider's network or a 3rd party firewall, the traffic (depending on the cloud provider's architecture) has a source network address translation applied to it (SNAT). In other words the source IP address of traffic from all workloads on the node is set to the node IP, so any kind of allow / deny policies, at best, will have the node level (node's IP address) granularity.

Kubernetes workloads are highly dynamic and ephemeral. Let's say a developer commits a new check in for a particular workload. The automated CI/CD workflow will kick in, build a new version of the pod (container) and start deploying this new version of the workload in Kubernetes clusters. Kubernetes orchestrator will do a rolling

upgrade and deploy new instances of the workload. All of this happens in an automated fashion, and there is no room for manual or out of band workflows to reconfigure the security controls for the newly deployed workload.

You need a new security architecture to secure workloads running in multi / hybrid cloud infrastructure. Just like your workload deployment in a Kubernetes cluster, the security of the workload has to be defined as code, in a declarative model. Security controls have to be portable across Kubernetes distributions, clouds, infrastructure and/or networks. These security controls have to travel with the workloads so if a new version of the workload is deployed in an Amazon. Elastic Kubernetes Service EKS VPC (instead of on-premise clusters) you can be assured that the security controls associated with the service will be seamlessly enforced without you having to rework any network topology or out of band configuration of security groups or VPC/perimeter firewalls.

Network security controls are implemented by using a network policy solution that is native to Kubernetes and provides granular access controls. There are several well known implementations of network policy (e.g. Calico, Weavenet, Kube-router, Antrea) that you can use for implementing network policy. In addition to applying policy at Layer 3/ Layer 4, (TCP/IP), we recommend you look at solutions that support application layer policy, (e.g. HTTP/HTTPS) policy. We recommend picking a solution that is based on the popular proxy, Envoy as it is very widely deployed for application layer policy. Kubernetes supports deploying applications as microservices (small components serving a part of the application functionality) that are deployed over a network of nodes. The communication between microservices relies on application protocols (e.g. HTTP). Therefore there is a need for granular application controls which can be implemented by application layer policy. For example, in a three tier application the frontend microservice may only be allowed to use HTTP GET based requests to the backend database microservice (read access) and not allowed to use HTTP POST to the backend database microservice which will allow write access. All these requests can end up using the same TCP connection so it is essential to add a policy engine that supports application level controls as described above.

Observability

Observability is very useful for monitoring a distributed system like Kubernetes. Kubernetes abstracts lot of details and in order to monitor a system like Kubernetes, you cannot collect and independently baseline and monitor individual metrics (for e.g. a single network flow, a pod create/destroy event, a CPU spike on one node) - what is needed is a way to monitor these metrics in the context of the Kubernetes. For example, a pod associated with a service or a deployment was restarted and it caused latency for transactions related to a service for a short amount of time or a pod was unavailable as it had a network policy not allowing access to an essential resource and this caused the associated service or deployment to be unavailable (a service outage).

This becomes even more complex when you consider an application comprises several services (microservices) which are in turn backed by several pods.

Observability is useful in troubleshooting and monitoring or workloads in Kubernetes as well as for security. As an example Observability in the context of a service in Kubernetes will allow you to do the following

- Visualize your Kubernetes cluster as a service graph, which shows how pods are associated with services and the communication flows between services.
- Overlay application (L7) and network traffic (L3/L4) on the service graph as separate layers as that will allow you to easily determine traffic patterns and traffic load for applications and for the underlying network.
- View metadata for the node where a pod is deployed on (e.g. CPU, memory, host OS details).
- View metrics related to operation of a pod, traffic load, application latency (e.g. HTTP duration), network latency (network round trip time), pod operation (e.g. RBAC policies, service accounts, container retstarts)
- View DNS activity (DNS response codes, latency, load) for a given service (pods backing the service)
- Trace a user transaction that needs communication across multiple services. This is also known as distributed tracing.
- View network communition of a given service to external entities
- View Kubernetes activity logs (e.g. audit logs) for pods and resources associated with a given service.

We will cover the details of Observability in Chapter 12, for this discussion we will cover a brief description of how you can use Observability as a part of your security strategy.

Network Traffic Visibility. As mentioned before a solution that provides network flows aggregated at a service level with context like namespaces, labels, service accounts, network policies is required to adequately monitor activity and access controls applied to the cluster. For e.g, there is a significant difference between reporting that IP1 communicated with IP2 over a port 8080 and reporting that pods labeled "frontend" communicated with pods labeled "backend" on certain ports or traffic patterns between deployments of pods in a Kuberenetes cluster. This will allow you tot review communication from external entities and apply IP address based threat feeds to detect activity from known malicious IP addresses or even traffic from unexpected geographical locations. We will cover details for these concepts in Chapter 11.

DNS activity logs. Domain Name System (DNS) is a system used to translate domain names into ip addresses. In your Kubernetes cluster, it is very critical to review DNS activity logs to detect unexpected activity, for example queries to known malicious domains, DNS response codes like NXDOMAIN and unexpected increase in bytes and packet in DNS queries. We will cover details for these concepts in Chapter 11.

Application Traffic Visibility. We recommend you review application traffic flows for suspicious activity like unexpected response codes, rare or known malicious HTTP headers (user-agent, query parameters). HTTP is the most common protocol used in Kubernetes deployments so it is important you work with your security research team to monitor HTTP traffic for malicious traffic. In case you use other application protocols (e.g. Kafka, MYSQL), you need to do the same for those as well.

Kubernetes Activity logs. In addition to network activity logs, you must also monitor Kubernetes activity logs to detect malicious activity. For example, review access denied logs for resource access, service account creation/modification, namespace creation/modification logs need to be reviewed for unexpected activity.

Machine Learning/Anomaly Detection. Machine learning is a technique where a system is able to derive patterns from data over a period of time.The output is a machine learning model, which can then be used to make predictions and detect deviations in real data based on the prediction. We recommend you consider applying Machine learning based anomaly detection to various metrics to detect anomalous activity. A simple and effective way is to apply a machine learning technique known as baselining to individual metrics, this way you do not need to worry about applying rules and thresholds for each metric, the system does that for you and reports deviations as anomalies. Applying Machine learning techniques to network traffic is a relatively new area and is gaining traction with security teams. We will cover this topic in detail in Chapters 11 and 12.

There are many solutions that you can choose from for your observability strategy for Kubernetes (Datadog, Calico Enterprise, Cloud provider based solutions from Google, AWS, Azure).

Enterprise Security Controls

Now that you have the strategy for network access controls and observability defined, you should consider additional security controls that are important and prevalent in enterprises. Encryption of data in transit is a critical requirement for security and compliance. There are several options to consider for encryption using traditional approaches like transport layer security (TLS) based encryption in your workloads, mutual TLS which is part of a service mesh platform or a VPN based approach like Wireguard (which offers a crypto key based VPN) .

We recommend that you leverage the data collection that is part of your observability strategy to build reports needed to help with compliance requirements for standards like PCI, HIPAA, GDPR, SOC2. You should also consider the ability to ensure continuous compliance and you can leverage the declarative nature of Kubernetes to help with the design and implementation of continuous compliance. For example, you can respond to a pod failing a compliance check by using the pod's compliance status to trigger necessary action to correct the situation (trigger an image update).

Threat Defense

Threat defense in a Kubernetes cluster is the ability to look at malicious activity in the cluster and then defend the cluster from it. Malicious activity allows an adversary to gain unauthorized access and manipulate or steal data from a Kubernetes cluster. The malicious activity can occur in many forms, for example exploiting an insecure configuration, exploiting vulnerability in the application traffic or the application code.

When you build your threat defense strategy, you must consider both intrusion detection and intrusion prevention. The key to intrusion detection is observability, you need to review data collected to scan for known threats. In a Kubernetes deployment data collection is very challenging due to the large amount of data that you need to inspect. We have often heard this question, "Do I need a Kubernetes cluster to collect data to defend a Kubernetes cluster? ". The answer is "no". We recommend you align your obersevrability strategy with intrusion detection and leverage smart aggregation to collect and inspect data. For e.g. you can consider a tool that aggregates data as groups of "similar" pods talking to each other on a given destination port and protocol instead of using the traditional method of aggregating by the "five tuple" (Source IP, Source Port, DestinationIP, Destination Port, Protocol). This approach will help reduce data collected significantly without sacrificing effectiveness. Remember several pods running the same container image are deployed in the same way will generate identical network traffic for a transaction. You may ask, "what if only one instance is infected, how can I detect that?", that is a good question, there are a few ways to address that, you should pick a tool that supports machine learning based on various metrics collected like connections, bytes, packets to detect anomalous workloads. Another approach is to have a tool that can detect and match known malicious IPs and domains from well known threat feeds as a part of collection or log un aggregated network flows for traffic denied by policy. These are simple techniques that will help you build a strategy, please note that the threat defense techniques evolve and you need a security research team to work with you to help understand your application and build a threat model that will be used to implement your threat defense strategy..

Security Frameworks

Finally, we want to make you aware of security frameworks that provide the industry a common methodology and terminology for security best practices. Security frameworks are a great way to understand attack techniques and best practices to defend and mitigate attacks. You should use them to build and validate your security strategy. Please note these frameworks may not be specific to Kubernetes, but they provide insights into techniques used by adversaries in attacks and the security researchers will need to review and see if they are relevant to Kubernetes. We will review two well known frameworks—MITRE and Threat Matrix for Kubernetes.

MITRE

MITRE is a knowledge base of adversary tactics and techniques based on real-world observations of cyber attacks. The MITRE ATT&CK® Matrix for Enterprise is very useful because it provides the tactics and techniques categorized for each stage of the cyber security security kill chain. The cyber security kill chain is a description of stages in a cyber attack and is useful to build an effective defense for a cyber attack. MITRE also provides an attack matrix tailored for cloud environments like AWS, Google cloud and Microsoft Azure.

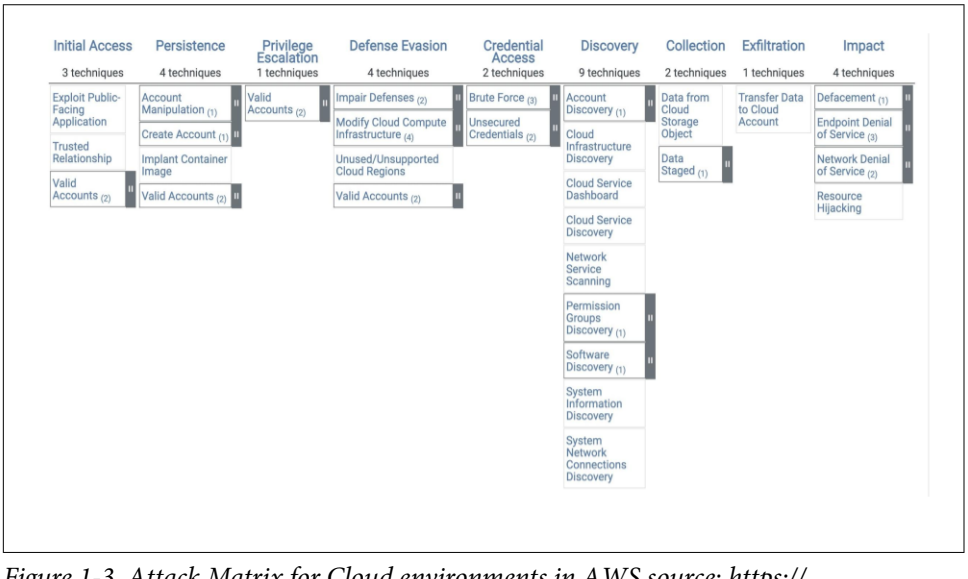


Figure 1-3. Attack Matrix for Cloud environments in AWS source: https://attack.mitre.org/matrices/enterprise/cloud/aws/

Figure 1-3 describes the MITRE ATT&CK® Matrix for AWS. We recommend that you review each of the stages described in the attack matrix as you build your threat model for securing your Kubernetes cluster.

Threat Matrix for Kubernetes

The following is a threat matrix that is a Kubernetes specific application of the generic MITRE attack matrix. It was published by the Microsoft team [ref: threat matrix for Kubernetes.] based on security research and real world attacks.This is another excellent resource to use to build and validate your security strategy

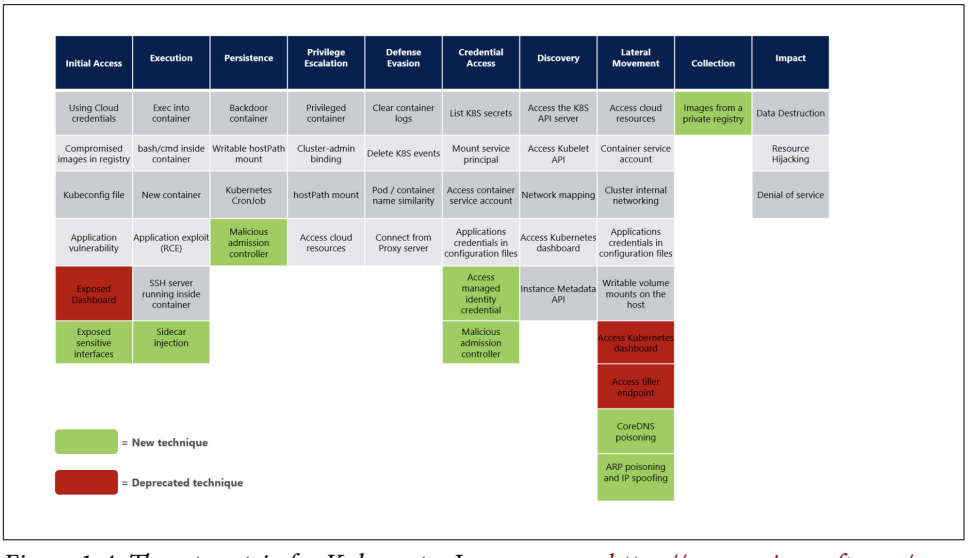


Figure 1-4. Threat matrix for Kubernetes Image source: https://www.microsoft.com/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/?_lrsc=2215f5af-27b7-4d0b-abd2-ad3fbd998797

Figure 1-4 provides the stages that are very relevant to your Kubernetes cluster. They map to the various stages we discussed in this chapter, for e.g. you should consider the Compromised images in registry in the Initial Access stage, Access cloud resources in the Privilege Escalation stage and Cluster internal network in the Lateral Movement stage for the build, deploy and runtime security respectively as described in this chapter.

Conclusion

By now you should have a high level overview of what Kubernetes security and observability entails. These are the foundational concepts that underpin this entire book. In short:

- Security for Kubernetes is very different from traditional security and requires a holistic security and observability approach at all 3 stages of workload deployment -- build, deploy and runtime.

- Kubernetes is declarative and abstracts details of workload operations, this means workloads can be running anywhere over a network of nodes. Also workloads can be ephemeral where they are destroyed and recreated on a different node. Securing such a declarative distributed system requires that you think about security at all stages.
- We hope you understand the importance of collaboration between the application, platform and the security teams to design and implement a holistic security approach
- MITRE and the Threat Matrix for Kuberntes are two security frameworks that are widely adopted by security teams.

It's important that you take all this together, because a successful security and observability strategy is a holistic one. In the next chapter, we will cover infrastructure security.

CHAPTER 2
Infrastructure Security

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Many Kubernetes' configurations are insecure by default. In this chapter we will explore how to secure Kubernetes at the infrastructure level. Kubernetes can be made more secure through the combination of host hardening to make the servers or VMs Kubernetes is hosted on more secure, cluster hardening to secure the Kubernetes control plane components, and the network security required to integrate the cluster with the surrounding infrastructure beyond the cluster boundary. Please note that the concepts discussed in this chapter apply to self hosted Kubernetes clusters as well as managed Kubernetes clusters

- **Host hardening** includes considering the choice of operating system, avoiding running non-essential processes on the hosts, and host based firewalling.
- **Cluster hardening** covers a range of configuration and policy settings needed to harden the control plane, including configuring TLS certificates, locking down etcd, encrypting secrets at rest, credential rotation, and user authentication and access control.

- **Network security** covers securely integrating the cluster with the surrounding infrastructure and in particular which network interactions between the cluster and the surrounding infrastructure are allowed, for control plane, host, and workload traffic.

Let's look at the details for each of these aspects and explore what is needed to build a secure infrastructure for your Kubernetes cluster,

Host hardening

A secure host is an important building block for a secure Kubernetes cluster. When you think of a host it is in the context of host for your workloads that comprise your Kubernetes cluster. We will now explore techniques to ensure a strong security posture for the host.

Choice of operating system

Many enterprises standardize on a single operating system across all of their infrastructure, which means the choice may have already been made for you. However, if there is flexibility to choose an operating system then it is worth considering a modern immutable Linux distribution specifically designed for containers. These distributions are advantageous for the following reasons:

- They often have newer kernels which include the latest vulnerability fixes, as well as up-to-date implementations of newer technologies such as eBPF, which can be leveraged by Kubernetes networking and security monitoring tools.
- They are designed to be immutable, which brings additional benefits for security. Immutability in this context means that the root filesystem is locked and cannot be changed by applications. Applications can only be installed using containers. This isolates applications from the root filesystem and significantly reduces the ability for malicious applications to compromise the host.
- They often include the ability to self-update to newer versions, with the upstream versions being geared up for rapid releases to address security vulnerabilities.

Two popular examples of modern immutable Linux distributions designed for containers are Flatcar Container Linux (which was originally based on CoreOS Container Linux) and Bottlerocket (originally created and maintained by Amazon).

Whichever operating system you choose, it is good practice to monitor upstream security announcements so you know when new security vulnerabilities are identified and disclosed, and to make sure you have processes in place to update your cluster to a newer version to address critical security vulnerabilities. Based on your assessment of these vulnerabilities you will want to make a decision on whether to upgrade your

cluster to a new version of the operating system. When you consider the choice of the operating system, you must also take into account shared libraries from the host operating system and understand their impact on containers that will be deployed on the host.

Another security best practice is to ensure that application developers do not depend on a specific version of the operating system or kernel as this will not allow you to update the host operating system as needed.

Non-essential processes

Each running host process is a potential additional attack vector for hackers. From a security perspective, it is best to remove any non-essential processes that may be running by default. If a process isn't needed for the successful running of Kubernetes, management of your host, or security of your host, then it is best not to run the process. How you disable the process will depend on your particular setup (e.g. systemd configuration change, or removing the initialization script from /etc/init.d/).

If you are using an immutable Linux distribution optimized for containers, then non-essential processes will have already been eliminated and you can only run additional processes/applications as containers.

Host based firewalling

To further lock down the servers or VMs Kubernetes is hosted on, the host itself can be configured with local firewall rules to restrict which IP address ranges and ports are allowed to interact with the host.

Depending on your operating system, this can be done with traditional Linux admin tools such as iptables rules or firewalld configuration. It is important to make sure any such rules are compatible with both the Kubernetes control plane and whichever Kubernetes network plugin you plan to use so they do not block the Kubernetes control plane, pod networking, or the pod network control plane. Getting these rules right, and keeping them up-to-date over time, can be a time consuming process. In addition, if using an immutable Linux distribution, you may not easily be able to directly use these tools.

Fortunately, some Kubernetes network plugins can help solve this problem for you. For example, several Kubernetes network plugins like Weave Net, Kube-router and Calico include the ability to apply network policies. You should review these plugins and pick one that also supports applying network policies to the hosts themselves (rather than just to Kubernetes pods). This makes securing the hosts in the cluster significantly simpler and is largely operating system independent, including working with immutable Linux distributions.

Always research the latest best practices

As new vulnerabilities or attack vectors are identified by the security research community, security best practices evolve over time. Many of these best practices are well documented online and are available for free.

For example, the Center for Internet Security maintains free PDF guides with comprehensive configuration guidance to secure many of the most common operating systems. These guides, known as "CIS Benchmarks" are an excellent resource for making sure you are covering the many important actions required to secure your host operating system. You can find an up-to-date list of CIS Benchmarks here: https://www.cisecurity.org/cis-benchmarks/. Please note there are Kubernetes specific benchmarks and we will discuss them later in this chapter.

Cluster hardening

Kubernetes is insecure by default. So in addition to hardening the hosts that make up a cluster, it is important to harden the cluster itself. This can be done through a combination of Kubernetes component and configuration management, authentication and role based access controls (RBAC), and keeping the cluster updated with the latest versions of Kubernetes to ensure the cluster has the latest vulnerability fixes.

Secure the Kubernetes datastore

Kubernetes uses etcd as its main datastore. This is where all cluster configuration and desired state is stored. Having access to the etcd datastore is essentially equivalent to having root login on all your nodes. Almost any other security measures you have put in place within the cluster become moot if a malicious actor gains access to the etcd datastore. They will have complete control over your cluster at that point, including the ability to run arbitrary containers with elevated privileges on any node.

The main way to secure etcd is to use the security features provided by etcd itself. These are based around x509 Public Key Infrastructure (PKI), using a combination of keys and certificates. They ensure that all data in transit is encrypted using Transport Layer Security (TLS) and all access is restricted using strong credentials. It is best to configure etcd with one of credentials (key pairs and certificates) for peer communications within between the different etcd instances, and another set of credentials for client communications from the Kubernetes API. As part of this configuration, etcd must also be configured with the details of certificate authority (CA) used to generate the client credentials.

Once etcd is configured correctly, only clients with valid certificates can access it. You must then configure the Kubernetes API server with the client certificate, key, and certificate authority so it can access etcd.

You can also use network level firewall rules to restrict etcd access so it can only be accessed from Kubernetes control nodes (hosting the Kubernetes API server). Depending on your environment you can use a traditional firewall, virtual cloud firewall, or rules on the etcd hosts themselves (for example, using a networking policy implementation that supports host endpoint protection) to block traffic. This is best done in addition to using etcd's own security features as part of a defense in depth strategy, since limiting access using firewall rules does not address the security need for Kubernetes sensitive data in transit to be encrypted.

In addition to securing etcd access for Kubernetes, it is recommended to not use the Kubernetes etcd datastore for anything other than Kubernetes. In other words do not store non-Kubernetes data within the datastore and do not give other components access to the etcd cluster. If you are running applications or infrastructure (within the cluster or external to the cluster) that use etcd as a datastore, the best practice is to set up a separate etcd cluster for that. The arguable exception would be if the application or infrastructure was sufficiently privileged that a compromise to its datastore would also result in a complete compromise of Kubernetes. It is also very important to maintain backups of etcd and secure the backups so that it is possible to recover from failures like a failed upgrade or recovering from a security incident.

Secure the Kubernetes API server

One layer up from the etcd datastore, the next set of crown jewels to be secured is the Kubernetes API server. As with etcd this can be done using x509 Public Key Infrastructure (PKI) and Transport Layer Security (TLS). The details of how to bootstrap a cluster in this way vary depending on the Kubernetes installation method you are using, but most Kubernetes installation methods include steps that create the required keys and certificates and distribute them to the other Kubernetes cluster components. It's worth noting that some installation methods may enable insecure local ports for some components, so it is important to familiarize yourself with settings of each component to identify potential unsecured traffic so you can take appropriate action to secure these.

Encrypt Kubernetes secrets at rest

Kubernetes can be configured to encrypt sensitive data it stores in etcd such as Kubernetes secrets. This keeps the secrets safe from any attacker that may gain access to etcd or to an offline copy of etcd such as offline backup.

By default Kubernetes does not encrypt secrets at rest and when encryption is enabled it only encrypts when a secret is written to etcd. Therefore when enabling encryption at rest, it is important to rewrite all secrets (through standard kubectl apply or update commands) to trigger their encryption within etcd.

Kubernetes supports a variety of encryption providers. It is important to pick the recommended encryption based on encryption best practices. The mainline recommended choice is to use the AES-CBC with PKCS#7 based encryption. This provides very strong encryption using 32-byte keys and is relatively fast. There are two different providers that support this encryption:

- The local provider that runs entirely with Kubernetes, and uses locally configured keys.
- The KMS provider that uses an external Key Management Service (KMS) to manage the keys.

The local provider stores its keys on the API server's local disk. This therefore has the limitation that if the API server host is compromised then all of your secrets become compromised. Depending on your security posture this may be acceptable.

The KMS provider uses a technique called envelope encryption. With envelope encryption each secret is encrypted with a dynamically generated Data Encryption Key (DEK), the DEK is then encrypted using a Key Encryption Key (KEK) provided by the KMS, and the encrypted DEK is stored alongside the encrypted secret in etcd. The KEK is always hosted by the KMS as the central root of trust and is never stored within the cluster. Most large public cloud providers offer a cloud based KMS service which can be used as the KMS provider in Kubernetes. For on-prem clusters there are third-party solutions, such as HashiCorp's Vault, which can act as the KMS provider for the cluster. As the detailed implementations vary, it is important to evaluate the mechanism through which the KMS authenticates the API server and whether a compromise to the API server host could therefore in turn compromise your secrets, and therefore offer only limited benefits compared to a local encryption provider.

If exceptionally high volumes of encrypted storage read/writes are anticipated then using the secretbox encryption provider could potentially be faster. However, secretbox is a newer standard and at the time of writing has had less review than other encryption algorithms. It may therefore not be considered acceptable in environments that require high levels of review. In addition, secretbox is also not yet supported by the KMS provider and must use a local provider which stores the keys on the API server.

Encrypting Kubernetes secrets is the most common must-have encryption at rest requirement, but note that you can also configure Kubernetes to encrypt storage of other Kubernetes resources if desired.

It's also worth noting there are third-party secrets management solutions that can be used if you have requirements for secrets management beyond the capabilities of Kubernetes secrets. One such solution, already mentioned as a potential KMS provider for envelope encryption in Kubernetes, is Hashicorp's Vault. In addition to pro-

viding secure secrets management for Kubernetes, Vault can also be used beyond the scope of Kubernetes to manage secrets more broadly across the enterprise if desired. Vault was also a very popular choice for plugging the major gap in earlier Kubernetes versions which did not support encryption of secrets at rest.

Rotate credentials frequently

Rotating credentials frequently makes it harder for attackers to make use of any compromised credential they may obtain. It is therefore a best practice to set short lifetimes on any TLS certificates or other credentials and automate their rotation. Most authentication providers can control how long issued certificates or service tokens are valid for and it is best to use short lifetimes whenever possible, for example rotating keys every day or more frequently if particularly sensitive. This needs to include any service tokens used in external integrations or as part of the cluster bootstrap process.

To fully automate the rotation of credentials may require custom devops development work, but normally represents a good investment compared to attempting to manually rotate credentials on an on-going basis.

When rotating keys for Kubernetes secrets stored at rest (as discussed in the previous section) local providers support multiple keys. The first key is always used to encrypt on any new secret writes. For decryption, the keys are tried in order until the secret is successfully decrypted. As keys are only encrypted on writes, it is important to rewrite all secrets (through standard kubectl apply or update commands) to trigger their encryption with the latest keys. If rotation of secrets is fully automated then the write will happen as part of this process without requiring a separate step.

When using a KMS provider (rather than a local provider), the KEK can be rotated without requiring re-encryption of all the secrets, which can reduce the performance impact of re-encrypting all secrets if you have a large number of sizable secrets.

Authentication & RBAC

In the previous sections we primarily focused on securing programmatic / code access within the cluster. Equally important is to follow best practices for securing user interactions with the cluster. This includes creating separate user accounts for each user, and using Kubernetes RBAC to grant users the minimal access they need to perform their role, following the principles of least privilege security practices. Usually it is better to do this using groups and roles, rather than assigning RBAC permissions to individual users. This makes it easier to manage privileges over time, both in terms of adjusting privileges for different groups of users when requirements change, and for reducing the effort required to periodically review / audit the user privileges across the cluster to verify they are correct and up-to-date.

Kubernetes has limited built-in authentication capabilities for users, but can be integrated with most external enterprise authentication providers, such as public cloud provider IAM systems, or on-prem authentication services, either directly or through third-party projects such as Dex (originally created by CoreOS). It is generally recommended to integrate with one of these external authentication providers rather than using Kubernetes basic auth or service account tokens, since external authentication providers typically have more user friendly support for rotation of credentials, including the ability to specify password strength and rotation frequency timeframes.

Restricting cloud metadata API access

Most public clouds provide a metadata API accessible locally from each host / VM instance. The APIs provide access to the instance's cloud credentials, IAM permissions, and other potentially sensitive information about the instance. By default these APIs are accessible by the Kubernetes pods running on an instance. Any compromised pod can use these credentials to elevate its intended privilege level within the cluster or to other cloud provider services the instance may have privileges to access.

To address this security issue, the best practice is to:

- Provide any required pod IAM credentials following the cloud provider's recommended mechanisms. For example, Amazon EKS allows you to assign a unique IAM role to a service account, Microsoft Azure's AKS allows you to assign a managed identity to a pod, and Google Cloud's GKE allows you to assign IAM permissions via Workload Identity.
- Limit the cloud privileges of each instance to the minimum required to reduce the impact of any compromised access to the metadata API from the instance.
- Use network policies to block pod access to the metadata API. This can be done with per namespace Kubernetes network policies, or preferably with extensions to Kubernetes network policies such as those offered by Calico which enable a single network policy to apply across the whole of the cluster (without the need to create a new Kubernetes network policy each time a new namespace is added to the cluster). This topic is covered in more depth later in the Default Deny and Default App Policy section of the Network Policy chapter.

Enable auditing

Kubernetes auditing provides a log of all actions within the cluster with configurable scope and levels of detail. Enabling Kubernetes audit logging and archiving audit logs on a secure service is recommended as an important source of forensic details in the event of needing to analyze a security breach.

The forensic review of audit log can help answer questions such as:

- What happened, when, and where in the cluster?
- Who or what initiated it and from where?

In addition, Kubernetes audit logs can be actively monitored to alert on suspicious activity using your own custom tooling or with third party solutions. There are many enterprise products that you can use to monitor Kubernetes audit logs and generate alerts based on configurable match criteria.

The details of what events are captured in Kubernetes audit logs are controlled using policy. The policy determines which events are recorded, for which resources, and with what level of detail.

Each action being performed can generate a series of events, defined as stages:

- RequestReceived - generated as soon as the audit handler receives the request
- ResponseStarted - generated when the response headers are sent, but before the response body is sent (only generated for long-running requests such as watches).
- ResponseComplete - generated when the response body has been completed.
- Panic - generated when a panic occurred.

The level of detail recorded for each event can be one of:

- None - does not log the event at all.
- Metadata - logs the request metadata (user, timestamp, resource, verb, etc.) but not the request details or response body.
- Request - logs event metadata and the request body but not response body.
- RequestResponse - logs the full details of the event including the metadata, and request and response bodies.

Kubernetes audit policy is very flexible and well documented in the main Kubernetes documentation. Included here are just a couple of simple examples to illustrate.

To log all requests at Metadata level:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

To omit RequestReceived stage, and log pod changes at RequestResponse level and configmap and secrets changes at Metadata level:

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
```

```
omitStages:
  - "RequestReceived"
rules:
  - level: RequestResponse
    resources:
    - group: ""
      resources: ["pods"]
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]
```

This second example illustrates the important consideration of sensitive data in audit logs. Depending on the level of security around access to the audit logs, it may be essential to ensure the audit policy does not log details of secrets or other sensitive data. Just like the practice of encrypting secrets at rest, it is generally a good practice to always exclude sensitive details from your audit log.

Restrict access to alpha or beta features

Each Kubernetes release includes alpha and beta features. Whether these are enabled can be controlled by specifying feature gate flags for the individual Kubernetes components. As these features are in development, they can have limitations or bugs that result in security vulnerabilities. So it is a good practice to make sure that all alpha and beta features you do not intend to use are disabled.

Alpha features are normally (but not always) disabled by default. They might be buggy, and the support for the feature could radically change without backwards compatibility, or be dropped in future releases. They are generally recommended for testing clusters only, not production clusters.

Beta features are normally enabled by default. They can still change in non-backward compatible ways between releases, but they are usually considered reasonably well tested and safe to enable, but as with any new feature they are inherently more likely to have vulnerabilities just because they have been used less and had less review.

Always assess the value an alpha or beta feature may provide against the security risk they represent, as well as the potential ops risk of non-backwards compatible changes of the features between releases.

Upgrade Kubernetes frequently

It is inevitable that new vulnerabilities will be discovered over time in any large software project. The Kubernetes developer community has a good track record of responding to newly discovered vulnerabilities in a timely manner. Severe vulnerabilities are normally fixed under embargo - meaning that the knowledge of the vulnerability is not made public until the developers have had time to produce a fix. Over

time the number of publicly known vulnerabilities for older Kubernetes versions grows, which can put older clusters at greater security risk.

To reduce the risk of your clusters being compromised, it is important to regularly upgrade your clusters, and to have in place the ability to urgently upgrade if a severe vulnerability is discovered.

All Kuberneters security updates and vulnerabilities are reported (once any embargo ends) via a public and free to join kubernetes-announce email group. Joining this group is highly recommended for anyone wanting to keep track of known vulnerabilities so they can minimize their security exposure.

Use a managed Kubernetes service

One way to reduce the effort required to act on all of the advice in this chapter is to use one of the major public cloud managed Kubernetes Services such as EKS, AKS, GKE, or IKS. Using one of these services moves security from being 100% your own responsibility to being a shared responsibility model. Share responsibility means there are many elements of security that the service includes by default, or can be easily configured to support, but that there are elements you still have to take responsibility for yourself in order to make the whole add up to being truly secure.

The details vary depending on which public cloud service you are using, but there's no doubt that all of them do significant heavy lifting which reduce the effort required to secure the cluster compared to if you are installing and managing the cluster yourself. In addition, there are plenty of resources available from the public cloud providers and from third parties which detail what you need to do as part of the shared responsibility model to fully secure your cluster. For example, CIS Benchmarks as discussed next.

CIS Benchmarks

As discussed earlier in this chapter, the Center for Internet Security (CIS) maintains free PDF guides with comprehensive configuration guidance to secure many of the most common operating systems. These guides, known as "CIS Benchmarks" can be an invaluable resource to help you with host hardening.

In addition to helping with host hardening, there are also CIS Benchmarks for Kubernetes itself, including configuration guidance for many of the popular managed Kubernetes services which help you implement much of the guidance in this chapter. For example the GKE CIS Benchmark includes guidance on ensuring the cluster is configured with auto-upgrade of nodes, and for using managing Kubernetes authentication and RBAC using Googe Groups. These guides are highly recommended resources to keep up-to-date with the latest practical advice on the steps required to secure Kubernetes clusters.

In addition to the guides themselves, there are third party tools available which can assess the status of a running cluster against many of these benchmarks. One popular tool is kube-bench (an open source project originally created by the team at Aqua Security). Or if you prefer a more packaged solution then many enterprise products have CIS Benchmark, and other security compliance tooling and alerting built into their cluster management dashboard. Having these kinds of tools in place, ideally running automatically at regular intervals, can be very valuable for verifying the security posture of a cluster and ensuring that careful security measures that might have been put in place at cluster creation time are not accidentally lost or compromised as the cluster is managed and updated over time.

Network security

When securely integrating the cluster with the surrounding infrastructure outside the scope of the cluster, network security is the primary consideration. There are two aspects to consider: how to protect the cluster from attack from outside the cluster; and how to protect the infrastructure outside of the cluster from any compromised element inside the cluster. This applies at both the cluster workload level (i.e. Kubernetes pods) and the cluster infrastructure level (i.e. the Kubernetes control plane and the hosts on which the Kubernetes cluster is running on).

The first thing to consider is whether or not the cluster needs to be accessible from the public internet, either directly (e.g. one or more nodes have public IP addresses) or indirectly (e.g. via a load balancer or similar which is reachable from the internet). If the cluster is accessible from the internet then the number of attacks or probes from hackers is massively increased, so if there is not a strong requirement for the cluster to be accessible from the internet then it is highly recommended to not allow any access at a routability level (i.e. ensuring there is no way of packets getting from the internet to the cluster). In an enterprise on-prem environment this may equate to choice of IP address ranges to use for the cluster, and their routability within the enterprise network. If using a public cloud managed Kubernetes service, you may find these settings can only be set at cluster creation. For example, in GKE, whether the Kubernetes control plane is accessible from the internet can be set at cluster creation..

Network policy within the cluster is the next line of defense. Network policy can be used to restrict both workload and host communications to/from the cluster and the infrastructure outside of the cluster. It has the strong advantage of being workload aware (i.e. the ability to limit communication of groups of pods that make up an individual microservices) and being platform agnostic (i.e. the same techniques and policy language can be used in any environment, whether on-prem within the enterprise, or public cloud). Network policy is discussed in-depth later in a dedicated chapter.

Finally, it is highly recommended to use perimeter firewalls, or their cloud equivalents such as security groups, to restrict traffic to/from the cluster. In most cases these are not Kubernetes workload aware, so they don't understand individual pods, and are therefore usually limited in granularity to treating the whole of the cluster as a single entity. Even with this limitation they add value as part of a defense in-depth strategy, though on their own they are unlikely to be sufficient for any security conscious enterprise.

If stronger perimeter defense is desired, there are strategies and third-party tools which can make perimeter firewalls or their cloud equivalents more effective.

- One approach is to designate a small number of specific nodes in the cluster as having a particular level of access to the rest of the network, which is not granted to the rest of the nodes in the cluster. Kubernetes taints can then be used to ensure that only workloads that need that special level of access are scheduled to those nodes. This way perimeter firewall rules can be set based on the ip addresses of the specific nodes to allow desired access and all other nodes in the cluster are denied access outside the cluster.
- In an on-prem environment, some Kubernetes network plugins allow you to use routable pod IP addresses (non-overlay networks) and control the IP address ranges the group of pods backing a particular microservice use. This allows perimeter firewalls to act on IP address ranges in a similar way as they do with traditional non-Kubernetes workloads. For example, you need to pick a network plugin that supports non-overlay networks on-prem that are routable across the broader enterprise networks, and has flexible IP address management capabilities that can facilitate such an approach.
- A variation of the above, which is useful in any environment where it is not practical to make pod IP addresses routable outside of the cluster (e.g. when using an overlay network). In this scenario, the network traffic from pods appears to come from the ip address of the node as it uses source network address translation (SNAT). In order to address you can use a Kubernetes network plugin which supports fine-grained control of egress NAT gateways. The egress NAT gateway feature supported by some Kubernetes network plugins, allows this behavior to be changed so that the egress traffic for a set of pods is routed via specific gateways within the cluster which perform the SNAT, so the traffic appears to be coming from the gateway, rather than from the node hosting the pod. Depending on the network plugin being used the gateways can be allocated specific IP address ranges or to specific nodes, which in turn allows perimeter firewalls rules to act more selectively than treating the whole of the cluster as a single entity. There are a few options that support this functionality, Red Hat's OpenShift SDN, Nirmata and Calico support egress gateways.

- Finally, some firewalls support some plugins or third-party tools which allow the firewall to be more aware of Kubernetes workloads. For example, automatically populating IP address lists within the firewall with pod IP addresses (or node IP addresses of the nodes hosting particular pods). Or in a cloud environment, automatically programming rules that allow security groups to selectively act on traffic to/from Kubernetes pods, rather than only operating at the node level. This integration is very important for your cluster to help complement the security provided by firewalls. There are several tools in the market that allow this type of integration. It is important to choose a tool that supports these integrations with major firewall vendors and is native to Kubernetes.

In the section above we discussed the importance of network security and how you can use network security to secure access to your Kubernetes cluster from traffic originating outside the cluster and also how to control access for traffic originating from within the cluster destined to hosts outside the cluster.

Conclusion

In this chapter we discussed the following key concepts that you should use to ensure you have a secure infrastructure for your Kuberentes cluster.

- You need to ensure that the host is running an operating system that is secure and free from critical vulnerabilities.
- You need to deploy access controls on the host to control access to the host operating system and deploy controls for network traffic to and from the host
- You need to ensure a secure configuration for your Kubernetes cluster, securing the datastore and API server are key to ensuring you have a secure cluster configuration.
- Finally you need to deploy network security to control network traffic that originates from pods in the cluster and is destined to pods in the cluster.

CHAPTER 3

Network Policy

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In this chapter, we will describe network policy and discuss its importance in securing a Kubernetes cluster. We will review various network policy implementations and tooling to support network policy implementations. We will also cover network policy best practices with examples.

What is network policy?

Network policy is the primary tool for securing a Kubernetes network. It allows you to easily restrict the network traffic in your cluster so only the traffic that you want to flow is allowed.

To understand the significance of network policy, let's briefly explore how network security was typically achieved prior to network policy. Historically in enterprise networks, network security was provided by designing a physical topology of network devices (switches, routers, firewalls) and their associated configuration. The physical topology defined the security boundaries of the network. In the first phase of virtuali-

zation, the same network and network device constructs were virtualized in the cloud, and the same techniques for creating specific network topologies of (virtual) network devices were used to provide network security. Adding new applications or services often required additional network design to update the network topology and network device configuration to provide the desired security.

In contrast, the Kubernetes network model defines a "flat" network in which every pod can communicate directly with all other pods in the cluster by default. This approach massively simplifies network design and allows new workloads to be scheduled dynamically anywhere in the cluster with no dependencies on the network design.

In this model, rather than network security being defined by network topology boundaries, it is defined using network policies that are independent of the network topology. Network policies are further abstracted from the network by using label selectors as their primary mechanism for defining which workloads can talk to which workloads, rather than IP addresses or IP address ranges.

Network policy enforcement can be thought of as each pod being protected by its own dedicated virtual firewall that is automatically programmed and updated in real time based on the network policy which has been defined. The figure below shows network policy enforcement at a pod using its dedicated virtual firewall.

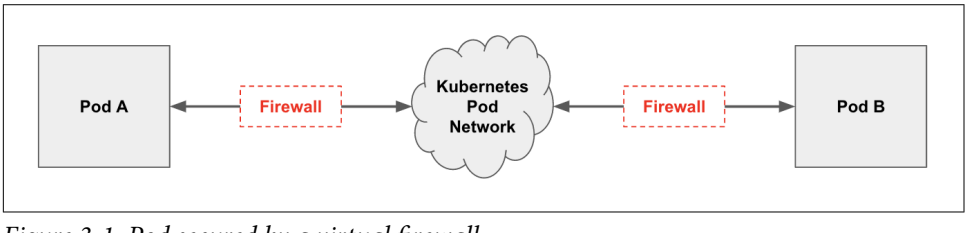


Figure 3-1. Pod secured by a virtual firewall

Why is network policy important?

In an age where attackers are becoming more and more sophisticated, network security as a line of defense is more important than ever.

While you can (and should) use firewalls to restrict traffic at the perimeters of your network (commonly referred to as north-south traffic), their ability to police Kubernetes traffic is often limited to a granularity of the cluster as a whole, rather than to specific groups of pods, due to the dynamic nature of pod scheduling and pod IP addresses. In addition, the goal of most attackers once they gain a small foothold inside the perimeter is to move laterally (commonly referred to as east-west) to gain access to higher value targets, which perimeter based firewalls can't police against. With application architectures evolving from monoliths to microservices, the amount

of east-west traffic, and therefore attack surface for lateral movement, is continuing to grow.

Network policy on the other hand is designed for the dynamic nature of Kubernetes by following the standard Kubernetes paradigm of using label selectors to define groups of pods, rather than IP addresses. And because network policy is enforced within the cluster itself, it can secure both north-south and east-west traffic.

Network policy represents an important evolution of network security, not just because it handles the dynamic nature of modern microservices, but because it empowers dev and devops engineers to easily define network security themselves, rather than needing to learn low-level networking details. Network policy makes it easy to define intent, such as "only this microservice gets to connect to the database", write that intent as code (typically in YAML files), and integrate authoring of network policies into git workflows and CI/CD processes.

Network policy implementations

Kubernetes defines a standard network policy API, so there's a base set of features you can expect on any cluster. But Kubernetes itself doesn't do anything with the network policy other than store it. Enforcement of network policies is delegated to network plugins, allowing for a range of implementations. Most network plugins support the mainline elements of Kubernetes network policies, though many do not implement every feature of the specification. In addition, it's worth noting that most implementations are coupled with the network plugin's specific pod networking implementation. However, some network policy implementations can enforce network policy on top of a variety of different pod networking plugins. The figure below shows network policies stored in the Kubernetes datastore being used by network plugins for enforcement.

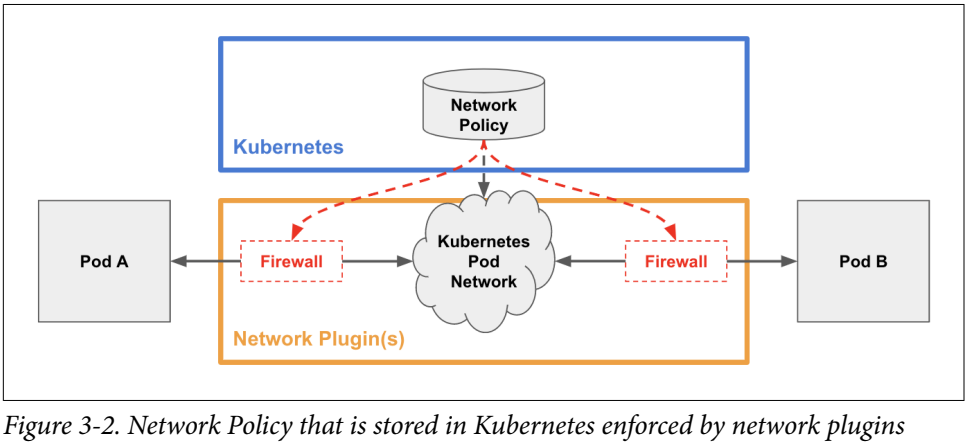


Figure 3-2. Network Policy that is stored in Kubernetes enforced by network plugins

There are a number of networking and network policy implementations to choose from, as shown in Figure 3-3.

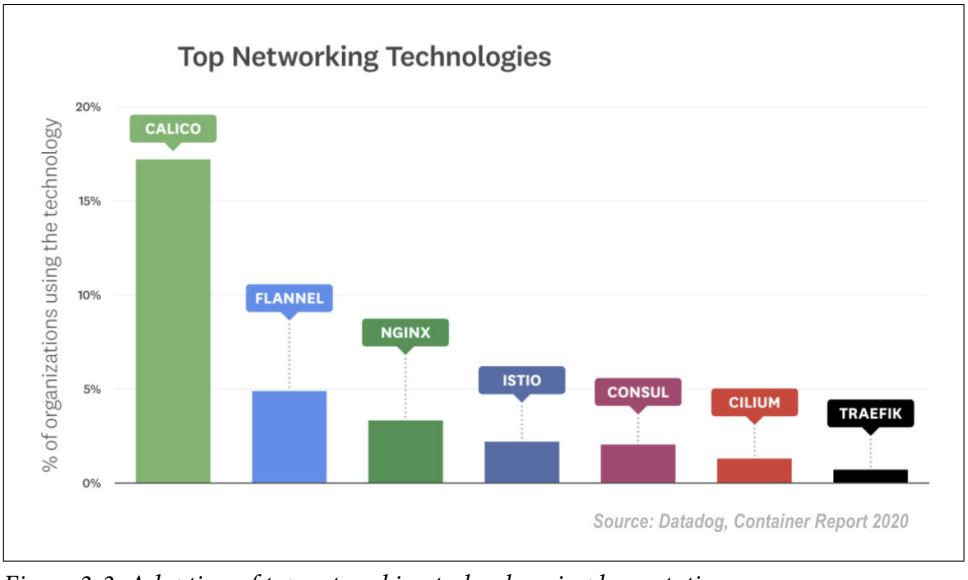


Figure 3-3. Adoption of top networking technology implementations

No matter what network policy implementation you choose, we recommend this tool for the following reasons:

- It implements the complete Kubernetes network policy specification.
- In addition to supporting the Kubernetes network policy specification, Its own policy model provides additional capabilities, which can be used alongside Kubernetes network policies to support additional enterprise security use cases.
- A few network plugins like Weave Net, Kube-router and Calico can enforce network policy on top of either their own rich set of networking capabilities, or on top of several other networking options, including the network plugins used by Amazon's Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE). This makes them a particularly strong choice as part of a multi-cloud strategy because it gives you the flexibility to select the best networking for your environment from a broad range of options with the same rich set of network policy features available across all environments.
- The network policy can be applied to host endpoints/interfaces, allowing the same flexible policy model to be used to secure Kubernetes nodes or non-cluster hosts/VMs.
- It supports network policy that is enforced both at the network/infrastructure layer and the layers above, including supporting L5-7 match criteria with its pol-

icy rules such as HTTP methods and paths. The multiple enforcement points help protect your infrastructure from compromised workloads and protect your workloads from compromised infrastructure. It also avoids the need for dual provisioning of security at the application and infrastructure layers, or having to learn different policy models for each layer

- It needs to be production grade, which means it must perform very well in clusters of any size from single node clusters to several thousand node clusters.
- It provides the ability for enterprises to add new capabilities and serves as a building block for an enterprise grade Kubernetes network security solution.

Network policy best practices

In this section we will explore how to implement network policy with examples and cover best practices for network policy implementation. The examples below use the Calico network policy schema which extends the Kubernetes network policy schema. We're using these examples due to our familiarity with Calico network policy, but these best practices can be implemented with other available network policy models as well.

Ingress and egress

When people think about network security, the first thought is often of how to protect your workloads from north-south external attackers. To help defend against this you can use network policy to restrict ingress traffic to any pods which are reachable from outside the cluster.

However, when an attacker does manage to find a vulnerability, they often use the compromised workload as the place from which to move laterally, probing the rest of your network to exploit additional vulnerabilities that give them access to more valuable resources, or allow them to elevate privileges to mount more powerful attacks or exfiltrate sensitive data.

Even if you have network policies to restrict ingress traffic on all pods in the cluster, the lateral movement may target assets outside of the cluster, which are less well protected. Consequently the best practice is to always define both ingress and egress network policy rules for every pod in the cluster.

While this doesn't guarantee an attacker cannot find additional vulnerabilities, it does significantly reduce the available attack surface, making the attackers job much harder. In addition, if combined with suitable alerting of policy violations, the time to identify that a workload has been compromised can be massively reduced. To put this into perspective, in the 2020 IBM Cost of a Data breach Report, IBM reported that on average it took enterprises 207 days to identify a breach, and a further 73 days to contain it! With correctly authored network policies and alerting of violations the breach

can be prevented or reduced potentially to minutes or seconds, and even opens the possibility of automated responses to quarantine the suspect workload if desired.

Not just mission critical workloads

The above best practice already recommends ensuring every pod has network policy that restricts its ingress and egress traffic. What this means is that when you are thinking about how to protect your mission critical workloads, you really need to be protecting all workloads. If you don't then some seemingly unimportant innocuous workload could end up being used as the base for attacks across the rest of your networking, ultimately leading to the downfall of your most critical workloads.

Policy and label schemas

One of the strengths of Kubernetes labels and network policies is there is a lot of flexibility in how you use them. However, as a result there are often multiple different ways of labelling and writing policies that can achieve the same particular goal. So another best practice is to consider standardizing the way you label your pods and write your network policies using a consistent schema or design pattern. This can make authoring and understanding the intent of each network policy much more straightforward, especially if your clusters are hosting a large number of microservices.

For example, you might say every pod will have an "app" label that identifies which microservice it is, and every pod will have a single network policy applied to it using that app label, with the policy defining ingress and egress rules for the microservices it is expected to interact with, again using the app label:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: back-end-policy
  namespace: production
spec:
  selector: app == 'back-end'
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: app == 'front-end'
    destination:
      ports:
        - 80
  egress:
  - action: Allow
    protocol: TCP
    destination:
      selector: app == 'database'
```

```
      ports:
        - 80
```

Or you might decide to use permission style labels in the policy rules, so rather than listing the microservices that are allowed to access each service in its ingress rules, any microservice that has the permission label is allowed:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: database-policy
  namespace: production
spec:
  selector: app == 'database'
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: database-client == 'true'
    destination:
      ports:
        - 80
  egress:
  - action: Deny
```

This could make it easier for individual microservice teams to author their own network policies, without needing to know the full list of other microservices that need to consume the service.

There are plenty of other ways you could go about it, and there is no right or wrong here. But taking the time to define how to approach labelling and defining network policies upfront can make life significantly easier in the long run.

If you are not sure which approach will work best for you, then following a simple "app" approach is a good place to start. This can always be expanded later to include the ideas of permission style labels for microservices which have a lot of clients if maintaining the policy rules becomes time consuming.

Default deny and default app policy

The Kubernetes network policy specification allows all ingress pod traffic, unless there is one or more network policy with an ingress rule that applies to the pod, and then only the ingress traffic that is explicitly allowed by the policies is allowed. And likewise for egress pod traffic. As a result if you forget to write a network policy for a new microservice, it will be left unsecured. And if you forget to write both ingress and egress rules for the microservice then it will be left partially unsecure.

Given this, a good practice is to put in place a "default deny policy" that prevents any traffic that is not explicitly allowed by another network policy. The way this is nor-

mally done is to have a policy that specifies it applies to all pods, with both ingress and egress rules, but does not explicitly allow any traffic itself. As a result, if no other network policy applies which explicitly allows the traffic then the traffic will then be denied.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  Namespace: my-namespace
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

As Kubernetes network policy is namespaced, the above network policy needs repeating for each namespace and ideally needs to be built into the standard operating procedure for provisioning new namespaces in the cluster. Alternatively, some network policy implementations go beyond Kubernetes network policy and provide the ability to specify cluster-wide network policies (that are not limited to a single namespace). The example below shows how to create a policy that switches the whole cluster to default deny behavior, including any namespaces that are created in the future.

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-deny
spec:
  selector: all()
  types:
  - Ingress
  - Egress
```

However, it's worth noting that the above policy applies to all pods, not just application pods, including control plane pods for Kubernetes. If you do not have the right network policies in place or failsafe ports configured before you create such a policy, you can break your cluster in pretty bad ways.

A much less high stakes best practice is to define a network policy that applies only to pods, excluding control plane pods. As well as triggering default deny behavior, this policy can also include any rules that you want to apply to all application pods. For example, you could include a rule which allows all application pods to access kube-DNS. This helps simplify any per microservice policies that need writing so they can focus solely on the desired per-microservice specific behaviors.

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-app-policy
```

```
spec:
  namespaceSelector: has(projectcalico.org/name) && projectcalico.org/name not
in {"kube-system", "calico-system"}
  types:
  - Ingress
  - Egress
  egress:
    - action: Allow
      protocol: UDP
      destination:
        selector: k8s-app == "kube-dns"
        ports:
        - 53
```

As the above policy deliberately excludes control plane components, to secure the control plane you can write specific policies for each control plane component. It is best to do any such policy creation at cluster creation time before the cluster is hosting workloads since getting these policies wrong can leave your cluster in a broken state that might result in a significant production outage. In addition, it is highly recommended you always make sure you have the correct failsafe ports for the network plugin you are using in place before you start trying to create any policies for the control plane.

Policy tooling

In this section we explore tools at your disposal to effectively add network policies to your Kubernetes cluster

Development processes & microservices benefits

One of the advantages of network policy compared to traditional network security controls is that defining network policy does not require networking or firewall expertise. Network policies use the same concepts and paradigms as other Kubernetes resources. In theory, any team that is familiar with deploying microservices in Kubernetes, can easily master network policies. As a result, network policy represents an opportunity to adopt a "shift left" philosophy for network security, where network security is defined earlier in the development cycle, rather than being defined late in the process.This is a great opportunity for the security and development teams to collaborate to secure your Kubernetes cluster.

At the same time, many organizations are moving from monolith application architectures to microservice architectures, often with one of the goals being to increase development and organizational agility. In such an approach, each microservice is typically maintained by a single development team, with that team having significant expertise on the microservice, but not necessarily the whole of the application that the microservice is a part of. The move to microservices complements the shift-left

opportunity of network policy. The team responsible for the development of a microservice normally has a good understanding of which other microservices they consume and depend on. They may also have a good understanding of which microservices consume their microservice.

When coupled with a well defined standardised approach to policy and label schemas, this puts them in a strong position to implement network policies for their microservice as part of the development of the microservice. In this model network policy is treated as code, built into and tested during the development process, just like any other critical part of a microservice's code.

An equally valid approach is to have development teams focus purely on the internals of the microservices they are responsible for, and leave responsibility for operating the microservices with devops teams. However, the same ideas still apply. Such a devops team typically needs a good understanding of the dependencies between the microservices they are responsible for operating in order to manage the operation of the application and lifecycle of the microservices. Network security can be defined as code by the devops team, and tested just like they would any other operational code or scripts they develop before using in production.

The reality today of course is that many organizations are some way off achieving this nirvana of microservices, agility, and shift-left security. Network security may come much later in the organization's processes, or even as an afterthought on a system already in production. In such scenarios, defining network policies may be significantly more challenging, and getting network policies wrong could have significant production impacts. The good news is that there are a range of tools to help with network policy lifecycle management to make this easier, including policy recommendations, policy impact previews, and policy staging / audit modes.

Policy recommendations

Policy recommendation tools are a great help in scenarios where the team responsible for network security does not have a good confident understanding of all the network dependencies between the applications or microservices they need to secure. These tools also help you get started with authoring network policies the right way and make the creation of network policy significantly easier than writing them by hand.

The way recommendation tools usually work is to analyze the network traffic to and from each microservices over a period of time. This means to get recommendations the microservice needs to be running in production, or a staging or test environment that accurately reflects the production interactions between the microservice and the rest of the application.

There are many policy recommendation tools available to choose from, often with varying levels of sophistication, degrees of Kubernetes awareness, and different policy

schema approaches. It is recommended that you use a Kubernetes aware policy recommendation engine built into your network policy solution.

Policy impact previews

Policy impact preview tools provide a way of sanity checking a network policy before it is applied to the cluster. Like policy recommendations, this is usually done by analyzing the cluster's historical network traffic over a period of time and calculating which network flows would have been impacted by the new policy. For example, to identify any flows which were previously allowed that would now be denied, and any flows were previously denied that would now be allowed.

Policy impact previews are a great help in any scenarios you are not relying 100% on policy recommendations. For example, if defining network policies by hand, or if modifying a policy recommendation to align with a particular standardised approach to policy and label schemas. Even if the team defining the network policy for a microservice has a high confidence in their understanding of the microservice's network dependencies, policy impact previews can be invaluable to help catch any accidental mistakes, such as hard to spot typos, that might otherwise have significantly impacted legitimate network traffic.

Policy impact preview tools are less common than policy recommendations, It is very useful to use a tool that provides a visual representation of the impact based on analysis of the flow log data it collects over any desired time period.This will help in reducing issues due to incorrectly authored policies or outages due to operator error.

Policy staging / audit modes

Even less common that policy impact previews, but potentially even more valuable, is support for policy staging, sometimes called policy audit mode.

Policy staging allows network policy changes to be applied to the cluster without impacting network traffic. The staged policy then records full details of all the flows it would have interacted with, without actually impacting any of the flows. This is incredibly useful in scenarios where a policy impact preview of an individual policy against historical data may be overly simplistic given the complexity of the applications running in the cluster. For example, if multiple interdependent policies need to be updated in unison, or where there's a desire to monitor the policy impact with live rather than historical network flows.

In order to make the task of authoring effective network policies less daunting, you need to use policy recommendation and then stage policies to understand the impact of the policy before you promote the policy for enforcement. This cycle of policy recommendation (based on historical network flows), followed by staging (applying policies to current and future network flows) ,followed by making desired adjust-

ments and then finally enforcing the policy is the best way to ensure the policy change would do exactly what you want.

Conclusion

In this chapter we discussed the importance of Network policy and various network policy implementations and tooling to help you with implementing network policy. The following are some key aspects of network policy

- Network Policy should be used to secure a Kubernetes network and it complements the firewalls that are implemented at the perimeter of your cluster.
- It is recommended that you choose a Kubernetes aware implementation of Kubernetes that extends the basic Kubernetes network policy
- There are a lot of network policy implementations that offer tooling to help with implementation of network policy in a Kubernetes cluster.

CHAPTER 4
Managing Trust Across Teams

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the previous chapter we explored how network policy represents an opportunity to adopt a "shift left" philosophy for network security, where network security is defined by teams earlier in the development cycle rather than being defined and maintained by a security team late in the process. This approach can bring a lot of benefits, but to be viable, there needs to be a corresponding degree of trust and split of responsibilities between the teams involved.

In most organizations, it's not practical to shift 100% of the responsibility for security all the way to the left, with all other teams (platform, network and the security) washing their hands of any responsibility for security. So for example, while the responsibility for lower level details of individual microservice security may be shifted-left, the security team may still be responsible for ensuring that your Kubernetes deployment has a security posture that meets internal and external compliance requirements.

Some enterprises handled this by defining internal processes, for example, to ensure the security team reviews all security changes before they are applied. The down side

of this approach is it can reduce agility, which is at odds with one of the motivations for shifting-left being to increase agility.

Fortunately, there are various types of guardrails that can be put in place across a Kubernetes environment that reduce the need for these kinds of traditional process controls. In this chapter we will explore some of these capabilities and how they can be used to control the degree of trust being delegated from one team to another in the context of a shift-left approach to security.

Role based access control

Kubernetes role based access control (RBAC) is the primary tool for defining the scope of what individual users or groups of users are permitted to do in a Kubernetes cluster. RBAC permissions are defined using roles and granted to users or groups of users using role bindings. Each role includes a list of resources (specified by resource type, cluster wide, within a namespace, or even a specific resource instance) and the permissions for each of the resources (e.g. get, list, create, update, delete, etc).

Many Kubernetes resources are namespaced, including deployments, daemonsets, pods and Kubernetes network policies. This makes the namespace an ideal trust boundary between teams. There are no set rules for how to use namespaces but one common practice is to use a namespace per microservice. RBAC can then be used to grant permission to manage the resources in the namespace to the team responsible for operating the corresponding microservice.

If security has been shifted-left, this would normally include permissions to manage the network policies that apply to the microservice, but not to manage any network policies that apply to microservices they are not responsible for.

If default-deny style best-practices are being followed for both ingress and egress traffic, then the team cannot forget to write network policies because the microservice will not work without them. In addition, since other teams will have defined equivalent network policies covering both ingress and egress traffic for the microservices they are responsible for, traffic is only allowed between two microservices if both teams have to have specified network policy that says the traffic is allowed. This further controls the degree of trust being delegated to each team.

Of course, depending on the degree to which security has been shifted-left, the responsibility for defining network policies may fall to a different team than the team responsible for operating the microservice. Again, Kubernetes RBAC can be used to easily reflect this split of responsibilities.

Limitations with Kubernetes network policies

There are a couple of limitations it is worth being aware of when using RBAC with Kubernetes network policies in a shift-left environment.

- Default deny style policies need to be created per namespace at the time the namespace is provisioned. The team responsible for defining network policies for the microservice would also have the ability to modify or delete this default policy if they wanted to.
- Network Policies are IP based and you cannot use fully qualified domain names (FQDN). This can be a limitation especially when defining policies to resources external to the cluster.
- Kubernetes RBAC controls access to resources but does not constrain the contents of resources. Of particular relevance in the context of network policies is pod labels since these are used as the primary mechanism for identifying other microservices in network policy rules. So for example, if one team has written a network policy for their microservice with a rule allowing traffic to it from pods with a particular label, then in theory any team with permission to manage pods could add that label to their pods and get access to the microservice. This exposure can be reduced by always using namespace sectors within policy rules and being selective as to which teams have permissions to change namespace labels.

If standardized policy and label schemas have been defined and the teams are trusted to follow them then these limitations are more of a theoretical rather than practical issue. However, for some organizations, they may represent genuine issues not sufficient for their security needs.

These organizations may therefore want to leverage additional capabilities beyond Kubernetes RBAC and Kubernetes network policies. In particular:

- Richer network policy implementations that support additional network policy types, match criteria, and non-namespaced network policies which open up more options for how to split responsibilities and RBAC across teams.
- Admission controllers to enforce controls on a per field level within resources, for example to ensure a standardized network policy and label schemas as followed, including limiting teams to using particular labels.

We will now review network policy implementations that extend the Kubernetes network policy and how you can use the same to manage trust.

Richer network policy implementations

Some network policy implementations support both Kubernetes network policies and their own custom network policy resources that can be used alongside or instead of Kubernetes network policies. Depending on the implementation, these may open up additional options for how to split responsibilities and use RBAC across teams. There are vendors that support richer network policy implementations that support the Kubernetes network policy and add more features (e.g., Weave, Kube-router, Antrea, Calico). We encourage you to review these and choose the best one that meets your needs. In this section we will look at the concrete example using Calico as it is the most widely deployed container network plugin.

Calico supports the Kubernetes network policy feature set, plus its own Calico network policy resources that can be used alongside Kubernetes network policies. There are two types of Calico network policy, both under the `projectcalico.org/v3` API group:

- NetworkPolicy - These policies are namespaced (just like Kubernetes network policies)
- GlobalNetworkPolicy - These policies apply across the whole of the cluster independent of namespace.

Both types of Calico network policy support a common set of capabilities beyond Kubernetes network policies, including:

- A richer set of match criteria than Kubernetes network policies, for example with the ability to match on Kubernetes Service Accounts.
- Explicit allow, deny or log actions for policy rules, rather than Kubernetes network policy actions which are implicitly always allow.
- Precedence ordering to define the evaluation order of the network policies if multiple policies apply to the same workload. (Note that if you are just using Kubernetes network policies, or Calico policies only with allow actions in them, then evaluation order doesn't make any difference to the outcome of the policies. However, as soon as there are any policy rules with deny actions, ordering becomes important.)

We want to mention that there are other network policy implementations that extend the Kubernetes network policy like Antrea which offers ClusterNetworkPolicy (similar to GlobalNetworkPolicy)

The following sample shows how you can implement network policies using Kubernetes RBAC. In the example below you can control network access based on the labels assigned to a service account. In Kubernetes pods have service accounts associated

with them and therefore pods can be identified by service accounts. You should use RBAC to control which users can assign labels to service accounts. The network policy in the example below uses the labels assigned to service accounts to control network access. In the following example, pods with an intern service account can communicate only with pods with service accounts labeled, `role: intern`:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: restrict-intern-access
  namespace: prod-engineering
spec:
  serviceAccountSelector: 'role == "intern"'
  ingress:
    - action: Allow
      source:
        serviceAccounts:
          selector: 'role == "intern"'
  egress:
    - action: Allow
      destination:
        serviceAccounts:
          selector: 'role == "intern"'
```

This way you can extend the concept of RBAC which controls service accounts access to a Kuberntes resource to network access. It is a 2-step process, RBAC is used to control label assignment to service accounts and a label based service account selector is used to control network access. These additional capabilities can be leveraged alongside Kubernetes network policies to more cleanly split responsibilities between higher-level cluster ops or security teams, and individual microservice teams.

For example:

- Giving the cluster ops or security team RBAC permissions to manage Calico network policies at the cluster-wide scope so they can define basic higher-level rules that set the overall security posture of the cluster. For example, a default deny style app policy (as discussed earlier in the "Network policy best practices" section) and policies to restrict cluster egress to specific pods.
- Giving each microservice team RBAC permissions to define Kubernetes network policies in the microservice's namespaces so they can define their own fine-grained constraints for the microservices they are responsible for.

On top of this basic split in network policy RBAC permissions, the cluster ops or security team can delegate different levels of trust to each microservice team by defining rules using namespaces or service accounts labels, rather than simplifying matching on pod labels. For example, defining policies to restrict cluster egress to specific pods using service account labels and giving the individual microservice teams to use,

but not edit, any service accounts assigned to their namespace. Through this mechanism some microservice teams may be granted permission to selectively allow cluster egress from some of their pods, while not offering other teams the same permissions.

Figure 4-1 provides a simplified illustration of how these ideas can be combined together.

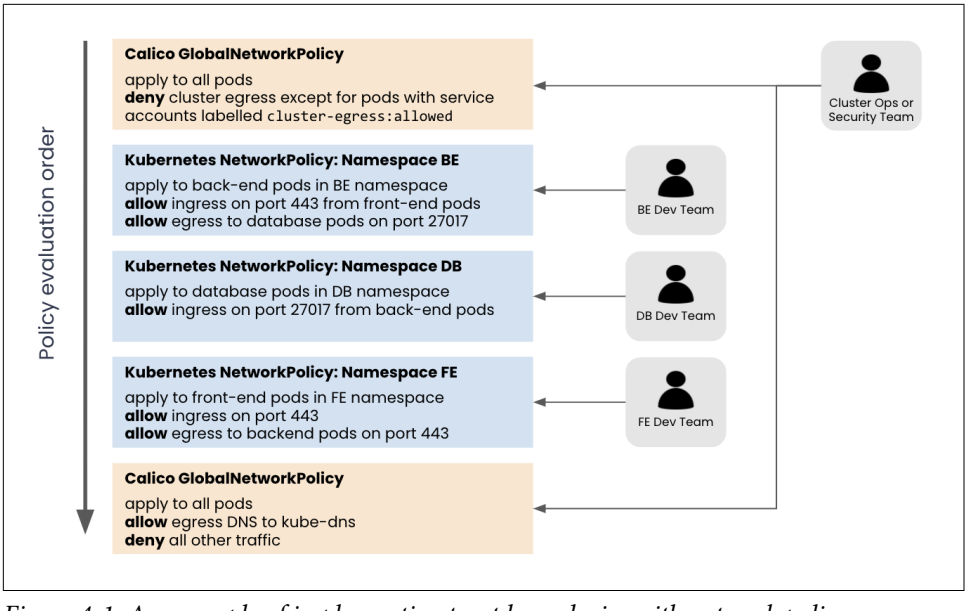


Figure 4-1. An example of implementing trust boundaries with network policy

While the above capabilities are reasonably powerful, in some organizations the required split of responsibilities across teams may be more complex, particularly where there are more layers of teams. For example, a compliance team, security team, cluster ops team, and individual microservice teams, all with different levels of responsibility. One way to more easily meet these requirements is to use a network policy implementation that supports the notion of hierarchical network policies.

There are some commercial implementations that support hierarchical network policy using policy tiers. A similar concept (Hierarchical namespaces and policies) is also being discussed in the Kubernetes community. RBAC for each tier can be defined to restrict who can interact with the tier. In this model, network policies are layered in tiers which are evaluated in a defined order, with as many tiers as required to match the organizational split of responsibilities. RBAC for each tier can be defined to restrict who can interact with the tier. The network policies in each tier can make allow or deny decisions (that terminate evaluation of any following policies), or can pass the decision on to the next tier in the hierarchy to be evaluated against the policies in that tier.

Figure 4-2 provides a simplified illustration of how these capabilities can be used to split responsibilities across three distinct layers of responsibility within an organization.

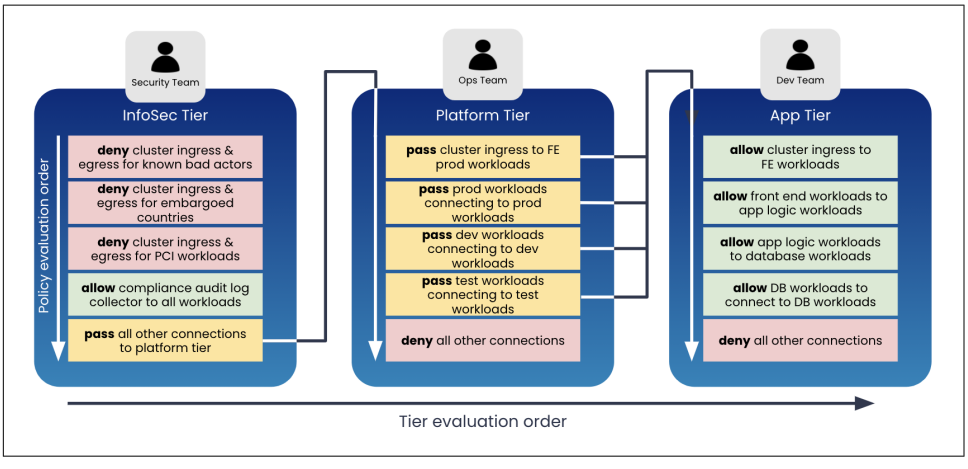


Figure 4-2. Implementing hierarchical network policies using tiers.

Admissions controllers

We have already discussed the value of defining and following standardised network policy and label schemas. The approaches for splitting responsibilities between teams discussed above are oriented around resource and namespace level RBAC, with teams having freedom to whatever they want within the resource and namespace scopes they are allowed to manage. As such they do not provide any guarantees that any such schemas are being followed by all teams.

Kubernetes itself does not have a built in ability to enforce restrictions at this granular level, but it does support an Admission Controller API which allows third-party admission controllers to be plugged into the Kubernetes API machinery to perform semantic validation of objects during create, update, and delete operations. You can also use admission controllers, known as mutating admission controllers for modifying objects that are admitted.

For example, in the context of network policy:

- Validate that network policies have both ingress and egress rules to comply with the best practices the organization is trying to follow.
- Ensure every pod has a specific set of labels to comply with the labelling standards the organization has defined.
- Restricting different groups of users to using specific label values.

But admission controllers have security use cases beyond network policy too. For example, Kubernetes Services include support for specifying an arbitrary IP address to be associated with the service using the Service's `ExternalIP` field. Without some level of policing this is a very powerful feature which could be used maliciously to intercept pod traffic to an IP address and redirect it to the Kubernetes service by anyone with RBAC permissions to create and manage Kubernetes Services. Policing this with an admission controller might be essential depending on the level of trust within the teams involved.

There are a few options for admission controller implementations, depending on the skill sets and specific needs of the organization:

- Using a pre-existing third-party admission controller that specializes in the specific controls the organization needs, if one exists.
- Writing a custom admission controller optimized for the organization's needs.
- Using a general purpose admission controller with a rich policy model that can map to a broad range of use cases.

For many scenarios choosing a general purpose admission controller gives a good balance of flexibility versus coding complexity. For example, Kyverno which has a policy engine specifically designed for Kubernetes, or an admission controller built around Open Policy Agent (OPA), where the policy model has flexible matching and language capabilities defined using Rego.

While admission controllers are very powerful, it is generally recommended to only implement them if you genuinely need them. For some organizations, using admission controllers in this way is overkill and not required given the levels of responsibility and trust across teams. For other organizations, they can be essential to meet internal compliance requirements, and the case for using them will be very clear.

Conclusion

Kuberentes security is different as it needs to be implemented by various teams and needs collaboration between teams. We covered the following key concepts:

- You should use RBAC and network policy to define boundaries that will help you manage activities across teams.
- You can extend the concept of RBAC to control network access by leveraging service accounts in network policy to help you manage trust
- How to use Admission controllers to help control access and implement trust boundaries across various teams.

- The importance of collaboration between the development, platform and the security teams, working together to implement security.

CHAPTER 5
Encryption of Data in Transit

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

As you move mission critical workloads to production, it is very likely that you will need to implement encryption of data in transit. It is a very important requirement for certain types of data to meet compliance requirements.and also a good security practice.

Encryption of data in transit is a requirement defined by many compliances standards, such as HIPAA, GDPR, and PCI. The specific requirements vary somewhat, for example PCI DSS (Payment Card Industry Data Security Standard) has rules around encryption of cardholder data while in transit. Depending on the specific compliance standard, this may mean you need to ensure data in transit between the applications or microservices hosted in Kubernetes must be encrypted using a recognized strong encryption algorithm.

Depending on the architecture of your application or microservices, it may be that not all data being sent over the network is classified as sensitive, so theoretically you might strictly only need to encrypt a subset of the data in transit. However, from the perspective of operational simplicity and ease of compliance auditing, it often makes

sense to encrypt all data in transit between your microservices, rather than trying to do it selectively.

Even if you do not have strong requirements imposed by external compliance standards, it can still be a very good practice to encrypt data in transit. Without encryption, malicious actors with network access could see sensitive information. How you assess this risk may vary depending on whether you are using public cloud or on-prem / private cloud infrastructure, and the internal security processes you have in place as an organization. In most cases, if you are handling sensitive data, then you should really be encrypting data in transit.

If you are providing services which are accessed by clients on the public internet, then the standard practice of using HTTPS apply to Kubernetes. Depending on your microservice architecture these HTTPS connections can be terminated on the destination microservice, or they may be terminated by a Kubernetes Ingress solution, either as an in-cluster Ingress pods (e.g. as when using the NGINX Ingress Controller) or an out-of-cluster application load balancer (e.g. when using the AWS Load Balancer Controller). Note that if using an out-of-cluster application load balancer, it's important to still make sure that the connection from the load balancer to the destination microservice uses HTTPS to avoid an unencrypted network hop.

Within the cluster itself, there are three broad approaches to encrypting data in transit:

- Build encryption capabilities into your application / microservices code.
- Use side-car or service mesh based encryption to encrypt at the application layer without needing code changes to your applications / microservices.
- Use network level encryption, again without the need to code changes to your applications / microservices.

We will explore the pros and cons of each approach.

Building encryption into your code

There are libraries to encrypt network connections for most programming languages, so in theory you could choose to build encryption into your microservices as you build them. For example, using HTTPS SSL/TLS or even mTLS (mutual TLS) to validate the identity of both ends of the connection.

However, this approach has a number of drawbacks:

- In many organizations, different microservices are built using different programming languages, with each microservice development team using the language that is most suited for that particular microservice and team's expertise. For

example, a front-end web UI microservice might be written using node.js, and a middle-layer microservice might be written in Python or Golang. As each programming language has its own set of libraries available for encryption, this means that the implementation effort increases, potentially with each microservices team having to implement encryption for their microservice rather than being able to leverage a single shared implementation across all microservices.

- Building on this idea of not having a single shared implementation for encryption, the same applies to configuration of the microservices. In particular, how the microservice reads its credentials required for encryption.
- In addition to the effort involved in developing and maintaining all this code, the more implementations you have, the more likely that one of the implementations will have bugs in it that lead to security flaws.
- It is not uncommon for older versions of encryption libraries to have known vulnerabilities which are fixed in new versions. By the time a new version is released to address any newly discovered vulnerability, the vulnerability is public knowledge. This in turn increases the number of attacks targeted at exploiting the vulnerability. To mitigate against this it is essential to update any microservices that use the library as soon as possible. If you are running many microservices, this may represent a significant development and test effort, since the code for each microservice needs to be updated and tested individually. On top of that, if you don't have a lot of automation built into your CI/CD process then there may also be the operational headache of updating each microservice version with the live cluster.
- Many microservices are based on third-party open source code (either in part or for the whole of the microservice). Often this means you are limited to the specific encryption options supported by the third-party code, and in many cases the specific configuration mechanisms the third-party code supports. You also become dependent on the upstream maintainers of the third-party code to keep the open source project up to date and address vulnerabilities as they are discovered.
- Finally it is important to note that there is often operational overhead when it comes to provisioning encryption settings and credentials across disparate implementations and their various configuration paradigms.

The bottom line then, is that while it is possible to build encryption into each of your microservices, the effort involved and the risk of unknowingly introducing security flaws (due to code or design issues or outdated encryption libraries) can make this approach feel pretty daunting and unattractive.

Side-car or service mesh encryption

An alternative architectural approach to encrypting traffic between microservices at the application layer is to use the side-car design pattern. The side-car is a container that can be included in every Kubernetes pod alongside the main container(s) that implement the microservice. The side-car intercepts connections being made to/from the microservice and performs the encryption on behalf of the microservice, without any code changes in the microservice itself. The side-car can either be explicitly included in the pod specification, or it can be injected into the pod specification using an admission controller at creation time.

Compared to building encryption into each microservice, the side-car approach has the advantage that a single implementation of encryption can be used across all microservices, independent of the programming language the microservice might have been written in. It means there is a single implementation to keep up to date, which in turn makes it easier to roll out vulnerability fixes or security improvements across all microservices with minimal effort.

You could in theory develop such a side-car yourself. But unless you have some niche requirement then it would usually be better to use one of the many existing free open-source implementations already available which have had a significant amount of security review and in-field hardening.

One popular example is the Envoy proxy, originally developed by the team at Lyft, which is often used to encrypt microservice traffic using mTLS (mutual TLS). Mutual TLS means that both the source and destination microservices provide credentials as part of setting up the connection, so each microservice can be sure it is talking to the other intended microservice. Envoy has a rich configuration model, but does not itself provide a control or management plane, so you would need to write your own automation processes to configure Envoy to work in the way you want it to.

Rather than writing this automation yourself, an alternative approach is to use one of the many service mesh solutions which follow a side-car model. For example, the Istio service mesh provides a packaged solution using Envoy as the side-car integrated with the Istio control and management plane. Service meshes provide many features beyond encryption, including service routing and visibility. While service meshes are becoming increasingly popular, a widely acknowledged potential downside of their richer feature set is it can introduce operational complexity, or make the service mesh harder to understand at a nuts and bolts level with a greater number of moving parts. Another downside is the security risk associated with the sidecar design pattern where the sidecar is part of every application pod and the additional complexity of managing sidecars (for e.g. a CVE may require you to update sidecars and this is not a trivial update as it impacts all applications)

Network layer encryption

Implementing encryption with the microservice or using a side-car model is often referred to as application layer encryption. Essentially the application (microservice or the side-car) handles all of the encryption and the network is just responsible for sending and receiving packets, without being aware the encryption is happening at all.

An alternative to application layer encryption is to implement encryption within the network layer. From the application's perspective, it is sending unencrypted data, and it is the network layer that takes responsibility for encrypting the packets before they are transmitted across the network.

One of the main standards for network layer encryption that has been widely used throughout the industry for many years is IPsec. Most IPsec implementations support a broad range of encryption algorithms, such as AES encryption, with varying key lengths. IPsec is often paired with IKE (Internet Key Exchange) as a mechanism for managing and communicating the host credentials (certificates and keys) that IPsec needs to work. There are a number of open source projects, such as the popular strongSong solution, that provide IKE implementations and make creating and managing IPsec networks easier.

Some enterprises choose to use solutions such as strongSwan as their preferred solution for managing IPsec, which they then run Kubernetes on top of. In this case Kubernetes is not really aware of IPsec. Even with projects such as strongSwan helping make IPsec easier to set up and manage, many regard IPsec as being quite heavyweight and tricky to manage from an overall operational perspective.

One alternative to IPsec is WireGuard. WireGuard is a newer encryption implementation designed to be extremely simple yet fast, using state-of-the-art cryptography. Architecturally it is simpler, and initial testing indicates that it does out-perform IPsec in various circumstances. It should be noted though that development continues on both Wireguard and IPsec, and in particular as advances are made to cryptographic algorithms, the comparative performance of both will likely evolve.

Rather than setting up and managing IPsec or WireGuard yourself, an operationally easier approach for most organizations is to use a Kubernetes network plugin with built-in support for encryption. There are a variety of Kubernetes network plugins that support different types of encryption, with varying performance characteristics.

If you are running network intensive workloads then it is important to consider the performance cost of encryption. This cost applies whether you are encrypting at the application layer or at the network layer, but the choice of encryption technology can make a significant difference to performance. For example, the following chart shows

independent benchmark results for four popular Kubernetes network plugins (the most recent benchmarks available at the time of writing, published in 2020).

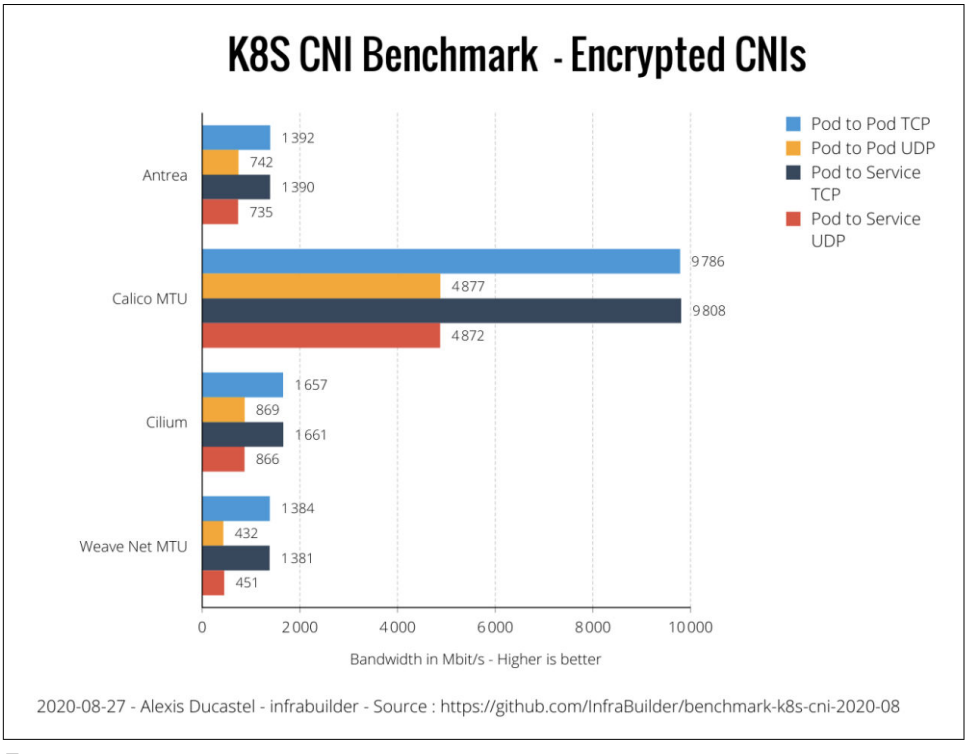


Figure 5-1.

Using a Kubernetes network plugin that supports encryption is typically significantly simpler from an operational standpoint, with many fewer moving parts than adopting a service mesh, and significantly less effort than building encryption into your application / microservice code. If your primary motivation for potentially adopting a service mesh is around security through encryption, then using a Kubernetes network plugin that supports network layer encryption along with Kubernetes network policies is likely to be significantly easier to manage and maintain. Please note that we will cover other aspects of service mesh like Observability in the chapter on Observability.

Conclusion

In this chapter we presented various options to implement encryption of data in transit and various approaches to implement encryption in a Kubernetes cluster. We hope this enables you to pick the option most suited for your use case.

- As you move mission critical workloads to production, for certain types of data, you will need to implement encryption for data in transit. We recommend implementing encryption of data in transit even if compliance requirements do not require you to encrypt all data.
- We covered the well known methods of how you can implement encryption, application layer encryption, sidecar based encryption using a servicemesh, network layer encryption.
- Based on operational simplicity and better performance, we recommend network layer encryption.

About the Authors

Alex Pollitt is CTO and co-founder of Tigera, and was one of the original co-creators of Calico, one of the most widely deployed open source networking and security solutions for Kubernetes. He has been an active member of the Kubernetes community since its inception and helped define many of the networking and security constructs that make Kubernetes what it is today.

Manish Sampat is Vice President of Engineering at Tigera, where he is responsible for all engineering operations, including Calico Cloud (commercial product) and Project Calico (open-source Kubernetes networking project). Manish has over a decade of experience building security products, with deep experience in networking, network security and observability.