O'REILLY®

Zero Trust Architecture in Kubernetes

Kim Crawley

Compliments of NGINX Part of F5

REPORT

Improve the Security Posture of Apps and Services in Kubernetes with F5 NGINX

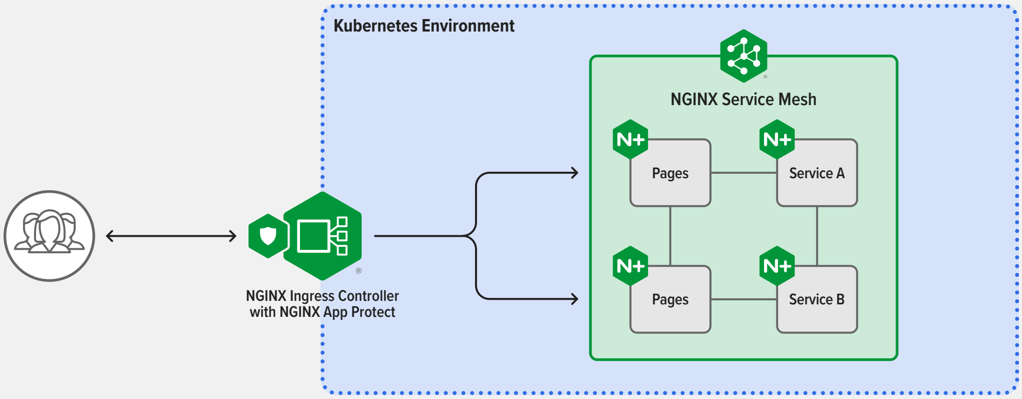
NGINX's Secure Kubernetes Connectivity simplifies deployment and operations with the infrastructure-agnostic tools you need to successfully implement Zero Trust in Kubernetes and protect your users, distributed applications, microservices, and APIs at scale across any environment – on-premises, hybrid, and multi-cloud. Powered by the most popular data plane in the world, the toolset includes:

NGINX Ingress Controller as the Ingress controller and a policy decision and enforcement point (PDP/PEP) that integrates with third-party identity providers. Based on NGINX Plus, NGINX Ingress Controller handles advanced connectivity, monitoring and visibility, authentication and SSO, and acts as an API gateway.

NGINX Service Mesh as a lightweight, turnkey, and developer-friendly service mesh that secures service connectivity within the Kubernetes cluster. The enterprise-grade sidecar that acts as the PDP/PEP colocated with each Kubernetes service is based on NGINX Plus.

NGINX App Protect for holistic protection of apps and APIs, built on F5's market-leading security technologies. It uses a modular approach for flexibility in deployment scenarios and optimal resource utilization:

- **NGINX App Protect WAF** – A strong, lightweight WAF that protects against the OWASP Top 10 and provides PCI DDS compliance
- **NGINX App Protect DoS** – Behavioral DoS detection and mitigation with consistent and adaptive protection across clouds and architectures



Get started by requesting your free 30-day trial of NGINX Ingress Controller with NGINX App Protect WAF and DoS, and download the always-free NGINX Service Mesh. Learn more today at nginx.com.

NGINX
Part of F5

Zero Trust Architecture in Kubernetes

Kim Crawley

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Zero Trust Architecture in Kubernetes
by Kim Crawley

Copyright © 2023 O'Reilly Media, Inc. 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*.

Acquisitions Editor: Simina Calin
Development Editor: Gary O'Brien
Production Editor: Christopher Faucher
Copyeditor: Audrey Doyle
Proofreader: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer

November 2022: First Edition

Revision History for the First Edition

2022-11-11: First Release

978-1-098-13861-5

[LSI]

Table of Contents

Introduction

Effective zero trust architecture is needed now more than ever. Even the United States government, through a White House Executive Order published in May 2021, acknowledges this:

> "The Federal Government must adopt security best practices; advance toward Zero Trust Architecture; accelerate movement to secure cloud services, including software as a service (SaaS), infrastructure as a service (IaaS), and platform as a service (PaaS); centralize and streamline access to cybersecurity data to drive analytics for identifying and managing cybersecurity risks; and invest in both technology and personnel to match these modernization goals."

What's needed to secure the public sector is just as necessary for the private sector. Kubernetes is now essential in the ever-growing SaaS, IaaS, and PaaS spaces. So creating a zero trust architecture designed to secure Kubernetes is vital to addressing enterprise cybersecurity needs in the 2020s and beyond.

It's time to get candid—securing Kubernetes is no walk in the park. Containers may have a lifespan of mere days. Different cloud platforms have different networking and security tools, and each platform has a unique API. Your public key infrastructure (PKI) in the cloud needs to assign and revoke machine identities at a rapid pace, most often in the form of certificates.

The State of the Cloud

The amount of cloud infrastructure your application needs can increase and decrease quickly, making scalable security solutions an absolute must. And although some developers and cybersecurity professionals have decades of experience, these technologies are still relatively new. Google Cloud Platform was launched in 2008 but has evolved immensely in the years since. The Amazon Web Services (AWS) cloud is a couple of years older, but the same applies. And Kubernetes was born in 2014. Even Craig McLuckie, Brendan Burns, and Joe Beda, Kubernetes' creators, don't have 10 years of experience with it as of this writing.

Perimeters, the Obsolete Way

As far as network security in general is concerned, threat boundaries used to be tied to machines. But now systems are much more complex and everyone is a potential threat actor.

In the traditional perimeter security model, your network is a collection of trusted servers, endpoints, applications, and users. Authentication vectors are set at the perimeter where your internal network ends and the internet and other external networks begin. That may have worked well enough in the 1990s, but now network computing and application design are radically different.

The Need to Adapt

The advent of the cloud and bring your own device (BYOD) was an immense sea change. Cloud platforms have made it much easier for enterprises to deploy applications and services to their end users and employees wherever they happen to be physically. The cloud has also empowered enterprises to deploy applications with scalability and agile-like flexibility.

It used to be that if an organization needed more bandwidth and capacity, it had to buy more servers and networking infrastructure, plus the building space to house all of it. And if an application needed less capacity or different hardware, an expensive number of machines and networking infrastructure would have to go unused. That's one of the reasons why organizations seldom have entirely on-premises networks these days. Enterprise networks that are completely on the cloud or on a hybrid cloud is how things are done now. That adaptability has revolutionized everything.

Zero trust is a security model and philosophy that makes it possible to secure today's and tomorrow's cloud networking environments. Your data assets in the cloud and through the internet need to be just as visible and securable as your data assets on premises! Therefore, no endpoint, server, application, or service should be automatically trusted based on its location in your network. Everything must be verified and authenticated as frequently as possible. This applies as much to east-west traffic (between internal applications) as to north-south traffic (between internal and external networks). And because zero trust is applied in all directions, it improves the security management of your applications and data assets regardless of their origin. Zero trust means no machine or user is trusted by default.

The Advent of Kubernetes

Kubernetes is such an important platform because it makes it possible for enterprises to deploy their applications in a fully scalable and dynamic way through the power of containerization. Before containerization and virtualization became common in the datacenter, the usual model was to run multiple applications on the same dedicated server hardware. It was the best that could be done with technology at the time. But that use of computing resources led to frequent downtime and overall poor performance. For instance, it was easy for one server application to use far more memory and CPU cycles than other applications.

When the Kubernetes platform was introduced in 2014, it made it much more feasible for developers to deploy their applications through containerization. That way, all applications (which should ideally have maximum uptime) could be assured of the resources they need in order to run well at all times.

Once developers and networking professionals got more experience working with the cloud and Kubernetes, enterprises could deploy their applications much more effectively. The cloud and containerization provided developers not only with better uptime and reliability but also with the ability to be immediately responsive to the needs of their customers, clients, and supply chains with tremendous scalability. Given that applications can change rapidly, making sure every part of an application has as much hardware capacity and bandwidth as it needs at any given time has been nothing short of revolutionary. And an enterprise can see that in its bottom line.

How Kubernetes Works

Technological innovation has improved business, but the innovative nature of Kubernetes requires a different approach in order to secure it properly. Containerization is a relatively new technology, having only been in common use for about a decade. And Kubernetes has been a major influence in popularizing containerization. Some organizations have made mistakes, such as not integrating their PKI with as many network vectors as possible or forgetting to use security controls that cloud platforms (such as AWS) provide to developers.

While virtual machines simply virtualize a computer and run multiple applications within it, Kubernetes has a container runtime layer in which an unlimited number of containers can run. The container runtime simplifies how data is exchanged between applications. But once an external data transmission enters the network and the API, that transmission has seamless access to all the containers in the Kubernetes system. And just as that external transmission can access anything in the container runtime, so, too, can the containerized applications within the runtime interact with one another.

What makes Kubernetes so powerful also makes a completely different security philosophy absolutely necessary. You don't want an attacker to have easy access to everything! Ransomware attacks and data breaches can cost organizations millions of dollars per incident in application downtime, reputation damage, litigation, and regulatory fines.

That's why zero trust architecture is an absolute necessity in order to deploy Kubernetes securely. With zero trust architecture, all the components of your Kubernetes system are secured by default. Nothing interacts with anything else without proper authentication, authorization, and nonrepudiation.

The Kubernetes platform is designed to be compatible with Ingress controllers and service meshes. An Ingress controller manages network traffic between your internal network and external networks by load-balancing and authenticating the traffic. A service mesh manages network traffic internal to your network, allocating some network resources and managing authentication inside your application. These powerful tools within your Kubernetes network can manage all your machine and user identities to ensure consistent authentication of everything. Applications and APIs are freed from that work, making compliance and security policy application much more effective. Kubernetes gives enterprises the power to take full advantage of the cloud in their application deployments. And to prevent and mitigate a plethora of cyber threats in today's rapidly evolving threat landscape, the support of zero trust architecture is essential.

The National Institute of Standards and Technology (NIST) defined how zero trust architecture (ZTA) should work in its SP 800-207 standard. Let's summarize some of the standard's concepts and how they apply to Kubernetes.

In Section 3.1 of the SP 800-207 document, NIST defines three requirements for ZTA. The first is that ZTA is designed using network infrastructure and software-defined perimeters when it's applied to a Kubernetes-based application. As per the document:

> "When the approach is implemented at the application network layer (i.e., layer 7), the most common deployment model is the agent/gateway (see Section 3.2.1). In this implementation, the agent and resource gateway (acting as the single PEP and configured by the PA) establish a secure channel used for communication between the client and resource."

Kubernetes works best when Ingress controllers and service meshes manage authentication. Therefore, those components serve as the security gateways, keeping that work away from the Kubernetes API and the applications that are run from containers.

The second requirement, covered in section 3.4.1, concerns network requirements for supporting ZTA.

A fundamental requirement is basic network connectivity. Your application needs to acquire traffic through an external network and within your internal network. That usage, plus authenticating and authorizing their associated requests to the Kubernetes API, requires network connectivity. Some additional requirements, as they pertain to Kubernetes, will be covered in this report.

The third requirement is that the enterprise must be able to distinguish between what assets are owned or managed by the enterprise and the devices' current security posture. Authentication and authorization determine the identity of users and machines and decide accordingly what they're allowed to do. Integrity ensures that data and configuration settings within your Kubernetes application are only altered by authorized entities. And maintaining an observable state makes it possible to verify your application's security posture.

Table I-1 summarizes how the requirements in the NIST SP 800-207 standard can be applied to your Kubernetes application.

Table I-1. NIST SP 800-207 pragmatics

Requirements	Applications
"The enterprise can observe all traffic."	Maintain an observable state based on logging, metrics, and traces.
"Enterprise resources should not be reachable without accessing a PEP (policy enforcement point)."	This is assured with proper Ingress controller and service mesh configuration.
"The data plane and control plane are logically separate."	This is managed by Kubernetes' design. The Kubernetes control plane includes the API server, scheduler, and controller managers. The data plane consists of nodes and their containers.
"Enterprise assets can reach the PEP component. Enterprise subjects must be able to access the PEP component to gain access to resources."	This is managed by Ingress controllers and service meshes.
"The PEP is the only component that accesses the policy administrator as part of a business flow."	This is why your Ingress controller and service mesh should manage authentication and authorization within your ZTA Kubernetes network, not the nodes, containers, or other Kubernetes components.
"Remote enterprise assets should be able to access enterprise resources without needing to traverse enterprise network infrastructure first."	This is taken care of with a properly configured Ingress controller.
"The infrastructure used to support the ZTA access decision process should be made scalable to account for changes in process load."	This is another reason why the Ingress controller and service mesh should manage security in your Kubernetes system.
"Enterprise assets may not be able to reach certain PEPs due to policy or observable factors."	This can be assured with proper Ingress controller and service mesh configuration, and by assuring an observable state.

So let's get started!

CHAPTER 1
Authentication and Authorization

Zero trust is about making sure all data transmissions and interactions are authenticated at all possible vectors within your network. In order to deploy zero trust architecture (ZTA) for Kubernetes, we must understand how authentication and authorization work within the containerization platform.

A user in a Kubernetes cluster is defined as any user account that has authentication credentials and is granted a certificate that is controlled by either a human or a machine. A Kubernetes cluster can have two types of users: service accounts and normal users. Service accounts usually originate from within a Kubernetes-based system for use by internal services and are managed by Kubernetes; normal users are managed by an outside service. But because Kubernetes doesn't have an object to represent normal user accounts, these accounts cannot be added to a cluster through an API call. A Kubernetes object is any entity that can be defined or configured with a YAML configuration file. I'll expand on YAML configuration files in Chapter 3.

Authentication of Normal Users

You can set up Kubernetes to verify machine identities external to its cluster. Consumers will indirectly interact with machine identities (such as TLS certificates) wherever they go on the web, within and outside of Kubernetes. So external machine identity verification makes it possible for a consumer of your enterprise's services to use your application securely.

Normal users can be authenticated through machine identities, meaning that if they have a valid certificate signed by the cluster's certificate authority (CA), they are considered authenticated. This works much the same way CAs work in web apps without Kubernetes, through TLS and HTTP port 443 by default. When Kubernetes doesn't manage a normal user, it's assumed that the internal network has a cluster-independent service (a third-party service originating from outside the Kubernetes cluster) that manages a list of usernames and passwords, private keys from an administrator, and a user store such as Google Accounts.

Authentication of Kubernetes-Managed Users

Conversely, applications originating from within a cluster will authenticate with service accounts managed by Kubernetes and therefore can make calls to the API if they're authenticated properly. API requests by Kubernetes-managed service accounts can use multiple authentication methods, such as bootstrap tokens, JSON Web Tokens, or client certificates. Each authentication module will be tried in sequence until one succeeds. This makes Kubernetes' API compatible with a variety of different applications.

If a user fails authentication (regardless of its origin), it's rejected with HTTP status code 401. If a user is authenticated, the next step is authorization. We know who you are. Now let's determine what you're allowed to do!

Authorization

To determine what a user is authorized to do, Kubernetes expects attributes that are common to REST API requests. This makes Kubernetes compatible with external access control systems that may handle APIs other than the Kubernetes API. Examples of API request attributes that work with Kubernetes include user, group, request path, and namespace, among others.

Once attributes are reviewed, it's time to determine what the API request wants to do—POST, GET, and DELETE are examples of common API requests. With that, the Kubernetes API server can authorize a request with a particular authorization mode that suits your network's security needs. Kubernetes supports role-based access control (RBAC) mode, attribute-based access control (ABAC) mode, Webhook mode (an HTTP callback), and node authorization mode, which grants permissions to kubelets based on the Pods they are scheduled to run.

Zero Trust Authentication and Authorization

In order for zero trust authentication and authorization to run properly in Kubernetes, your network needs service meshes and Ingress controllers. These components manage authentication through machine identities and are used to authorize requests to the Kubernetes API rather than delegate those security controls to the containerized applications and API server. There are also other useful functions that Ingress controllers and service meshes provide as a segment between the application and the API, including (but not limited to) service discovery, traffic management, and reading the state of your Kubernetes deployment.

Assuming that your Kubernetes system handles interactions between applications and from external networks (such as the internet), both service meshes and Ingress controllers are needed. The service mesh manages the east-west traffic between applications, and the Ingress controller manages the north-south traffic from your internal Kubernetes network to external networks.

Service meshes are designed to be used with Kubernetes to implement zero trust security. East-west traffic between applications within your Kubernetes cluster is often governed by a control plane, which in turn manages sets of data planes (otherwise known as sidecars in many implementations), which are responsible for traffic delivery.

Proper configuration of your service mesh, Ingress controller, CAs, and overall public key infrastructure (PKI) is crucial for automating authentication and authorization in Kubernetes effectively. It's integral to your ZTA, facilitating robust security.

By acting as a gateway between other networks, your application, and the Kubernetes API, Ingress controllers and service meshes provide lots of useful functionality that makes cloud-driven applications run more smoothly and securely. By modifying the settings in your Ingress controllers and service meshes, you can change how your application manages network traffic, access control (more about that in Chapter 4), and authentication without having to change any application code. Developers can really appreciate that, because changing code in one script may mean having to change code in lots of other places and update the codebases of multiple services. Smaller application deployments may only use meshes and controllers for authentication, but there are a lot of additional automation features that can be very useful, even to the most complex DevOps.

Proper authentication and authorization management is crucial to effective application and network security. Controllers and meshes can make these functions much more effectively when implemented properly.

Let's Review

Establishing effective authentication and authorization functionality is crucial to securing all computer systems, and Kubernetes is no exception. Zero trust architecture requires robust authentication and authorization at as many vectors as possible, regardless of the origin of users and traffic. Here are some tips to keep in mind:

- Review your current authentication contexts and roles. The principle of least privilege must apply whenever privileges are assigned to any human- or machine-controlled account. That means accounts have only the privileges they require in order to perform their work.
- Once your cluster's API server is secured, make sure your application endpoints are secured. Review access via service NodePorts (an open port on every node of your cluster), port forwarding, or insecure Ingress routes. Kubernetes roles can secure the cluster but not necessarily your application.

CHAPTER 2
Integrity

Integrity is a trait that can sometimes be overlooked. But if someone does what they say they're going to do and lives according to their well-founded values, that's a person who you'd be better off doing business with!

You might be wondering why I bring that up, since you're here to make zero trust architecture (ZTA) work effectively with Kubernetes. Data with integrity is much like a person with integrity: it does what it says it will do and can be trusted to live according to its values. And that's the type of data you're better off doing business with!

Ultimately, you must ensure that the data stored by and transmitted through your Kubernetes application hasn't been altered by malicious threat actors nor by well-intentioned people making honest mistakes. Effectively encrypted and monitored access can prevent all the data within your Kubernetes system from being tampered with and therefore can protect its integrity. We'll get to access encryption and monitoring in Chapters 6 and 7. For now, we'll focus on how to verify the integrity of your data in your zero trust Kubernetes application system.

The NIST SP 800-53 standard requires cloud applications, such as your Kubernetes applications, to prevent the installation of any components that haven't been digitally verified with a signed certificate that's recognized by your organization. Certificates can be managed by your enterprise's own certificate authority (CA) or by a CA your organization trusts.

IBM researchers Ruriko Kudo, Hirokuni Kitahara, Kugamoorthy Gajananan, and Yuji Watanabe have identified the challenges of implementing signatures within a Kubernetes cluster for the sake of integrity verification. There wasn't a way to do this through the Kubernetes API until the dry-run feature was added to the API's library (more on dry-run in a bit). They presented an effective solution at the 2021 IEEE 14th International Conference on Cloud Computing. Because data integrity is crucial for your ZTA, their methodology can be very useful. I'll show you their methodology later in this chapter, but first we must understand how Kubernetes objects work, because those are the entities you can use to ensure the integrity of your data.

Integrity Verification Through Kubernetes Objects and Signed Resources

A Kubernetes object is a type of data entity that can have a specific configuration within Kubernetes, such as a CronJob (explained in Chapter 5) or a Kubernetes-managed user (explained in Chapter 1). All Kubernetes objects are configured with YAML. These YAML files are how your Kubernetes cluster and containers store their configurations. They can describe which containerized applications are running and on which nodes, their behavior policies, and which resources are available to your applications.

Kubernetes objects are considered to be a "record of intent," and the system will work to ensure that the objects are persistent. Kubernetes objects are managed by the Kubernetes API, and when used properly they can function as signatures to ensure and verify data integrity within your ZTA.

An admission controller in Kubernetes intercepts authenticated and authorized API requests prior to establishing object persistence. However, it can be difficult for an admission controller to verify signatures, for a few reasons. One reason is because Kubernetes resources are rewritten internally (for the sake of resource allocation by control plane entities such as the kube-scheduler) before the admission controller receives them signed and in YAML format. The Kubernetes objects in the YAML files are used to create signatures for the resources associated with them, to be used for integrity verification (as long as the hashing algorithm is properly referenced in the YAML files).

Signed resources are passed to the admission controller as admission requests when they're installed in your cluster. But the resources in admission requests differ from the state of the original resource when the signature is generated because of the resource allocation work done in the control plane. Oops! In order to verify the signature of the original resource, we need to start by handling the difference between the original resource and the resource in the admission request that occurs due to the internal Kubernetes operation.

Another reason that handling the signatures of admission requests can be difficult is because requests can originate both outside and inside clusters. In the normal operation of your Kubernetes cluster, admission requests are generated frequently. Requests generated by internal operations have expected mutations that can't be attached to signatures.

The admission controller can block unsigned requests from your cluster, which can cause problems with your cluster's operation. To make sure the integrity of your Kubernetes data can be verified within your ZTA, admission requests caused by internal cluster operations must be filtered from admission requests that originate externally from your cluster. The majority of admission requests will likely originate outside your cluster.

Integrity Protection Based on Digital Signature

As mentioned earlier, IBM researchers Kudo, Kitahara, Gajananan, and Watanabe developed a solution that works, outlined as follows:

1. An administrator generates a signature for a Kubernetes object defined in a YAML file, which is then attached to the object. An encoded message representing the entire resource is also attached. Signatures can be generated using RSA (Rivest-Shamir-Adleman) or GPG (GNU Privacy Guard) ciphers.
2. When the signed resource is applied to the cluster, the admission controller receives an admission request with it. The Kubernetes cluster then rewrites it.

Integrity Protection Based on Digital Signature | 7

3. For the admission controller to control requests for signature verification, the verification server gets the signature and message from the Kubernetes resource in the admission request.
4. The signature goes through the verification process. If it fails to be verified, the request is rejected. If the signature is verified, it moves to the next step.
5. Dry-run is a feature that was introduced to the Kubernetes API server. It's designed to test whether modified requests to the API can be processed successfully. A dry-run is performed on the verified signature using the decoded message to receive a simulated Kubernetes resource from the API server. This is only a test!
6. The original message is compared to the simulated Kubernetes resource generated by dry-run, and the fields are calculated by the cluster.
7. The authentic Kubernetes resource masked by the system-inserted attribute is checked to determine whether it matches the simulated resource with the same attribute. If the resources match, the request is permitted to be processed by the Kubernetes API. If the resources don't match, the request is rejected.

This solution is one way to ensure the integrity of your YAML Kubernetes objects, which is essential to maintaining effective zero trust security in your cloud-hosted Kubernetes application. There wasn't a reliable way to ensure integrity verification before dry-run was added to the Kubernetes API, as entities in the control plane rewrite YAML files for resource allocation purposes.

Let's Review

To properly secure a Kubernetes application in ZTA, integrity verification is a must. Now that dry-run is available and IBM's team has done essential research, it's much more feasible to design integrity verification into your Kubernetes system by taking full advantage of digital signatures based on YAML files. It takes a bit of tinkering to set up, but that effort will be worth it in the long run.

Here are some tips:

- Follow up with and review IBM's research paper, and give it a try!
- Signing Kubernetes objects isn't the only thing you can do to ensure integrity. There are many options to ensure integrity for your node images, as well as digitally signing container images.
- Have you reviewed the source of your container images? Consider moving to a private registry where all images can be scanned and vetted.

CHAPTER 3

Observable State

It's impossible to secure what cannot be seen. Anyone who has ever worked in a security operations center (SOC) can tell you that their everyday work entails using lots of log analysis software, alerting SOC analysts to any anomaly that appears. Effective defensive security requires logging as many devices and applications in the network as possible, keeping an eye on those logs, and staying informed about performance metrics. If an event isn't logged, it will be missed by security monitoring software and the human professionals using it. Zero trust architecture (ZTA) is something a secure Kubernetes deployment must have, but it doesn't come with Kubernetes software out of the box. Establishing an observable state means that all the events and metrics within your application are visible to security controls and the human beings who manage them.

Sounds simple, doesn't it? Especially since intrusion detection systems and intrusion prevention systems (IDSs and IPSs), endpoint detection and response (EDR) systems, firewalls, and security information and event management (SIEM) systems can be configured to automate most of the log analysis and anomaly detection processes in enterprise networks. A typical enterprise network logs hundreds or thousands of events every minute, so automating as much as possible is an absolute necessity.

Cloud-driven application networks are integral to most enterprises these days. But it's important to recognize that those innovative technologies have even more complex security monitoring requirements.

So perhaps it's not so simple after all. A tried-and-true defensive security approach starts by inventorying every single data asset in your network. From there, those assets can be logged and monitored. But how can you maintain an accurate network data asset inventory if a container in a Kubernetes cluster might live for only a couple of days? And in the ZTA that Kubernetes requires, nothing is trusted until it has been authenticated, regardless of its origin. How can user access to a container be managed when containers have very brief lifespans?

Managing the machine identities and PKI that a zero trust environment for Kubernetes needs is delegated to Ingress controllers and service meshes. But some functions, such as integrity verification (discussed in Chapter 2), require methods that aren't completely built into Kubernetes.

As with integrity, ensuring an observable state in a zero trust Kubernetes environment also requires a bit of clever invention. Kubernetes has no security controls on its own, and the platform is designed in a way that makes logging a challenge. We need a thoroughly observable state in order to secure the entirety of our Kubernetes-based applications. It's not an easy problem to solve, but there is a way to do it!

Observable Data Sources

First you need to determine all the possible observable data sources within a Kubernetes deployment. There are many. A multitude, in fact. It can get overwhelming.

The most fundamental data source is your cloud provider's API. Then there's the control plane, which is the base of your Kubernetes cluster. The control plane generates observable data through its Kubernetes API server, controller manager, and scheduler. The control plane has several nodes as its children, and the number of nodes will vary according to what your Kubernetes application needs. Each node runs kubelets and kube-proxies. The nodes' children are Pods, and the Pods run the containers, which are the entities that execute the various parts of your applications.

Cluster-Level Logs and Metrics

The first important data source for observability is cluster-level resource metrics. These can be found through the Kubernetes Metrics API. Then there are cluster-level logs for entities such as the scheduler. These are found in the filesystems of your master nodes, which are the nodes that are directly operated through the control plane.

Application and Operating System Logs

Application logs belong to containers, which are the entities that run your applications. Whether your applications were developed by the vendors in your supply chain or they were developed in-house, there's a lot of variety in how applications manage logging. But the vast majority of the time, logs should be collected inside your containers. Applications will seldom send logs to an external location unless they were deliberately designed that way. If that's the case, the external source will also need to be monitored, making this operation even more complicated. Kubernetes containers are known to have a variety of lifespans. Sometimes they run for mere hours. That's part of Kubernetes' agile scalability, but it also makes logging those containers a bit tricky. Therefore, all application logs must be exported to some sort of persistent storage.

The operating system logs belong to the master and worker nodes. You will get those directly from each node's filesystem.

Metrics

In addition to the available logs, metrics are another data source that will help you maintain visibility of the observable state of your Kubernetes system. Kubernetes generates cluster-level resource consumption metrics. Similar metrics can be acquired through each individual node. Metrics need to be monitored directly through the Kubernetes API, as per the official documentation.

Traces

Traces are one of the most flexible data sources. A trace is a network analysis function that can find evidence of activity between points of a network and points within applications, such as connected features, interactions, loops, and paths of action. Traces can be performed in a Kubernetes network in much the same way they're performed in other types of computer networking environments. If there's some sort of error or other sort of anomalous event in your application, a trace can map interactions between various resources and services.

Log analysis and metrics analysis are almost always automated, whereas traces can sometimes be executed manually to investigate a bug or failure.

Establishing an Observable State

When you establish an observable state in your Kubernetes system, you must gather all of these data sources responsively. Containers sometimes only live for hours, and Pods can quickly move from one node to another. All the system resources also expand, contract, and move around quickly and mutably.

Even the applications within your containers can change frequently, and as one version of an application is updated to another, that change doesn't necessarily occur at the same rate as the life and death of the containers or the ever-shifting locations of Pods. Everything is temporary, and yet everything must leave a record that can be monitored and analyzed. Your historical data from yesterday may reflect a very different network than what exists today. But acquiring all that data is crucial to maintaining your ZTA.

Kubernetes, by its design, doesn't make establishing thorough observability simple. You're going to have to do the work. There's no master Kubernetes log; you're going to have to implement every possible logging source. Kubernetes makes acquiring metrics data possible through its Metrics API, but Kubernetes doesn't provide any tools for streaming metrics data; you'll need to use your own, such as Prometheus. Application logs come from within each and every container. And because containers can live fast and die young, exporting data to a persistent source is essential.

Don't overlook any of the sources; acquire as many as possible. Don't just aggregate logs, don't just watch applications, and don't just monitor metrics. You'll need to implement everything for effective observability. Popular cloud platforms provide some Kubernetes-specific tools, such as Azure Kubernetes Service and Amazon Elastic Kubernetes Service. These tools can be very useful, and it's important to study the documentation about how to use them properly. But if your observability strategy implements them, they must be used as helpful components in a diverse system and not as your only strategy. With or without the tools that cloud platforms provide, as many data sources as possible must be monitored, both internal and external to your cloud platform.

Let's Review

Establishing an observable state in your Kubernetes network is necessary to make sure its entirety can be kept secure. But because there are multiple sources of logs and metrics in a Kubernetes application, it takes a bit of work and thorough exploration to ensure that everything is being monitored and analyzed properly. Make sure you don't miss any possible source of security and network information.

Here are some tips:

- Observability is often described as consisting of three pillars: logging, metrics, and tracing. Your next steps are to identify what insights you and your organization want to gain from these pillars.
- Familiarize yourself with the Cloud Native Computing Foundation (CNCF) projects that are at the sandbox and graduated maturity levels. There are many options for metrics and for logging and tracing.

CHAPTER 4

Minimal Access in Size

Now let's discuss how to reduce your cyberattack surfaces as much as possible. A useful paradigm is the principle of least privilege. Every single user, application, and machine within a computer network must only have access to what's absolutely necessary for operation and nothing more. For example, in a role-based access control (RBAC) system, only the financial user groups can access the financial data servers, only administrators can modify configuration files, and so on.

RBAC isn't merely an effective security control in computer networks in general. It's strongly recommended in Kubernetes networks specifically. Sometimes accounts for human users will be assigned roles. But in Kubernetes, you'll also be focused on the roles assigned to Pods, applications, machines, and other nonhuman entities. I'll get into RBAC implementation in Kubernetes in greater detail soon. For now, let's get back to the basics.

Granting any entity more access than is absolutely necessary increases the possible cyberattack surface if a malicious agent acquires unauthorized control of said entity. In addition to malicious activity, security incidents can be caused by human error and application bugs. Therefore, we design our systems so that if something goes wrong, as little damage as possible will result. And every experienced network administrator and application developer knows that lots of things will inevitably go wrong.

Reducing Access in Size

In implementing zero trust architecture (ZTA) for your Kubernetes-based applications, one important way to ensure the least amount of damage from a security incident is to allow access to the smallest amount of data as possible. The most fundamental way to do this is to limit access to hardware resources. Namespaces are an instrument for this. In Kubernetes, namespaces are used to delegate hardware resources to clusters. A cluster can have multiple namespaces if necessary. If your Kubernetes deployment needs to serve lots of users spread among multiple projects or teams, your clusters will need more than one namespace.

Namespaces aren't only for hardware resource management. They are also used as a fundamental unit of isolation for name collisions, authorizations, and the like. Make sure your namespaces have a defined limit for how much memory they can access. Memory footprints will grow and shrink frequently, and you don't want them to completely take over your hardware. Imagine if something went haywire in one of your clusters. It could happen!

You can create namespaces for your clusters by using kubectl at the command line. Kubectl is one of the most important programs for managing all Kubernetes deployments. Keep in mind that each node in your cluster must have at least 2 GB of memory. Don't be *too* stingy!

Kubernetes uses YAML for all configuration files. `LimitRange` is a Kubernetes object that can be defined in a YAML file to restrict memory usage. It's also prudent to limit the CPU usage of your namespaces. `LimitRange` can be used in a YAML file for that purpose as well.

Those are the basics of restricting the size of what can be accessed as far as your clusters and their hardware usage is concerned. Next, we'll talk about limiting the amount of data through RBAC.

Role-Based Access Control

You can limit the amount of data that can be accessed at the filesystem and application levels through effective implementation of RBAC as it's defined in the Kubernetes API. There's an API group within the Kubernetes API specifically for making authorization decisions through RBAC: *rbac.authorization.k8s.io*. In order to use RBAC, it must be included in a list in the authorization mode flag when you start the API server at the command line.

Four types of Kubernetes objects are declared in the RBAC API: `Role`, `ClusterRole`, `RoleBinding`, and `ClusterRoleBinding`. The `Role` object is used to set permissions to namespaces, of which a cluster can have one or many. The `ClusterRole` object sets permissions for—you guessed it—the clusters themselves.

Permissions associated with those objects are defined by combining verbs (get, create, delete) with resources (Pods, services, nodes). Kubernetes comes with a set of out-of-the-box roles that can be used for common Kubernetes RBAC needs, but you may also need to set up your own custom roles. Remember that you'll have to set up node authorizers in addition to authorizers of your roles in the Kubernetes API.

The `RoleBinding` and `ClusterRoleBinding` objects work with the `Role` and `ClusterRole` objects to grant the permissions defined by the latter two objects to users or groups of users.

As with authentication, simple and broad roles may be appropriate for smaller clusters, but as more users interact with the cluster, it may become necessary to separate teams into distinct namespaces with more limited roles. As a general rule, the fewer users you have, the broader the permissions granted to your clusters and namespaces can be. But if your Kubernetes implementation has a large number of users, clusters, and namespaces, you will need to define roles with more specific permissions and restrictions. No matter what, the principle of least privilege applies: no user, cluster, or namespace should have permission to access more than is necessary for proper operation.

To complement RBAC in Kubernetes, it also helps to understand Pod Security admission. It's possible and often necessary to define different isolation levels for Kubernetes Pods.

Pod Security admission is based on three security levels defined by the Pod Security Standards: privileged, baseline, and restricted. The privileged level is unrestricted; use this level as sparingly as possible, because threat actors can exploit privileged Pods for privilege escalation attacks. The baseline level applies the default privileges as defined in the Pods' parent security policies. The restricted level is often the best to use as long as it doesn't conflict with your application's needs. This level assigns heavily restricted privileges following current Pod-hardening best practices.

Exemptions in your Pod Security admission can be applied to `Usernames`, `RuntimeClassNames`, and `Namespaces` in your Admission Controller configuration. Refer to the official Kubernetes documentation for more help with applying RBAC to Pods.

Let's Review

RBAC is possibly the most straightforward way to design the principle of least privilege in a computer network. Fortunately, Kubernetes has RBAC support built in. Understanding how you can assign privileges to Pods, users, and other Kubernetes entities is key.

Here are some things to consider:

- Have you reviewed your use of resources? Not only can you use `LimitRanges`, but there are resource limits for each Pod in the pod spec and ResourceQuotas to explore.
- Pods can also be assigned various levels of isolation. The original Pod Security Policies were deprecated in v1.21. Familiarize yourself with Pod Security admission and refer to Kubernetes documentation for further help.

CHAPTER 5

Minimal Access in Time

In the previous chapter we talked about why an important step in establishing effective zero trust architecture (ZTA) in which to run Kubernetes is to minimize the size of the resources your namespaces, clusters, and users can access. Doing so lowers the cyberattack surface if a threat actor acquires unauthorized access to a Kubernetes entity, and it also reduces any potential damage from user error or software bugs. But there's another way to apply the principle of least privilege to minimize access for effective security.

Do you remember when you started learning about physics in school? Or perhaps you're a real physics nerd and know way more about that branch of science than I do. Either way, we both know that matter is one of the fundamental physical properties. Matter and size are related concepts. So the next step is to think of another fundamental physical property: time! Both time and size are ways to define access. It's time (ahem) to talk about limiting resource access time.

CronJobs

The main way to implement time management in Kubernetes is with the CronJob object. CronJobs automate a schedule for performing regular tasks, such as creating data backups or generating reports. Backups are particularly important when it comes to the "A" component of the CIA triad of cybersecurity: availability. There are various threats to the availability of your network's data assets—incidents like ransomware attacks, denial-of-service (DoS) attacks, and data storage failure. Backups are one of the most important use cases for a CronJob.

Like all other Kubernetes configurations, CronJobs are defined in YAML files. The `schedule` attribute is used to define the times when a CronJob is executed. The Cron schedule syntax takes account of minutes, hours, days of the month, months, and days of the week (such as Monday, Tuesday, etc.):

```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                       7 is also Sunday on some systems)
# │ │ │ │ │                    OR sun, mon, tue, wed, thu, fri, sat
# │ │ │ │ │
# * * * * *
```

There are five variables in a Cron schedule, and each has either a wildcard (*) or a one- or two-digit numeral that may contain / or – (divide or subtract) for some basic arithmetic. If you need help generating the syntax, there's a handy "crontab guru" web app at crontab.guru.

Here are some simple examples: the syntax for referring to every time it is 14:15 (the 24-hour clock representation of 2:15 p.m.) on the first of each month is `15 14 1 * *`, and the syntax for referring to every time it's 22:00 on a weekday is `0 22 * * 1-5`.

Here's a more complicated example: the syntax to refer to 23 minutes after every other hour from 0 (midnight) to 20 (8 p.m.) hours is `23 0-20/2 * * *`.

Of course, it's not the same time all around the world. By default, the time zone a CronJob will refer to is the one in which your kube-controller-manager operates. But you can also specify a different time zone in the YAML file with the `spec.timeZone` attribute.

CronJobs can be created to schedule many different kinds of automated tasks. CronJobs, when designed properly, can be an effective way to limit the time during which entities in your Kubernetes system perform different actions. You'll want to perform some tasks frequently, such as data backups. Others you may want to restrict, like letting an entity write data to some other entity on the first Saturday of November.

But there's one thing you must keep in mind when you use CronJobs to limit access according to a schedule. You can specify the time when a CronJob begins to execute, but once it starts, the CronJob will run for as long as it needs to in order to complete the task. For example, you can't tell Kubernetes to execute the CronJob for only 30 seconds. The task will run for as long as it takes to complete.

Now, what if, instead of defining the time to perform actions according to a clock or calendar, you need to perform actions according to other conditions? That's where Jobs—CronJobs without the "Cron" part—can be used.

Jobs

In Kubernetes, a Job is a type of Pod controller. A Job will create one Pod or multiple Pods as needed, and it will execute them until the task has been successfully completed.

Jobs are so useful that it pays to dig deep into all the attributes that can be used in their YAML definitions. You can make many kinds of tasks execute under a set variety of conditions within your Kubernetes cluster. Maybe a container should be killed if a particular user account accesses the API. Maybe `RoleBinding` should only grant a `Role` to a user when they've done X, Y, and Z. These aren't time-related limits; rather, they're related to whether or not an action was performed under certain conditions. Like CronJobs, a Job will execute a task until it's complete, no matter how short or long it takes. Therefore, regardless of whether you use a Job or a CronJob, you can only specify when a task begins.

Actions can be performed under limitations defined by a schedule or conditional factors, augmenting how zero trust demands authentication at all points with an even more detailed security design.

Let's Review

By taking full advantage of the Jobs and CronJobs features built into Kubernetes, you can ensure that facets of your application are only accessed as necessary, reducing the cyberattack surface in your zero trust–based Kubernetes application.

Here are some more tips:

- Not all workloads must be Pods and long-lived, restarted entities. Review your workloads to use Jobs and CronJobs effectively. Jobs and CronJobs use `PodTemplateSpecs`, too, and in doing so they create Pods governed by different rules. In addition, not all workloads must be controlled by `Deployments`, `DaemonSets`, and `StatefulSets` as long-lived, restarted entities.
- Each workload type has settings to control its lifecycle. Use lifecycle controls to minimize runtimes and have applications execute only when they're needed.

CHAPTER 6

Monitor All Access

We mentioned that authenticating users and machines at as many points as possible, regardless of whether they originated internal or external to your network, is key to a zero trust environment. Zero trust means nothing is trusted by default. Today's cloud and hybrid cloud networks have no easily definable security perimeter, and it's better for security in general to authenticate accounts as frequently as possible.

Now that it's common wisdom in the tech industry (based on my peer interactions) that zero trust, when implemented properly, is a superior security model, it makes me imagine the havoc that the traditional perimeter security model facilitated for decades. "Ah ha, I breached through the DMZ and network perimeter with this decryption key some developer left stored in cleartext in a database! That's right. I *am* the company's network administrator. Let's make some *interesting* changes to the *.htaccess* file on this web app's Apache server..." Often, just one successful authentication exploit was all it took. Actually, let's shift to the present tense. There are still lots of traditional perimeter security networks running today. Legacy tech can often be difficult to replace. That's why Fortran programmers are paid so well.

One authentication vector at a perimeter is woefully insufficient. But in zero trust, we get rid of the perimeter concept and challenge users to authenticate (or present proof of authentication) every time they access anything. More authentication vectors mean authentication exploits are less likely to threaten your data assets. A threat actor can get lucky once, but can they get lucky dozens of times in one session?

We all know how zero trust is a more effective way to implement authentication vectors in a network. But once an entity can prove its identity multiple times and in multiple ways, is that enough to trust them? No way. Proving an identity is one thing. How that identity is used is another. That's why an effective zero trust architecture (ZTA) must also have robust access monitoring systems. Monitoring how access control runs in your network is just as important as authenticating everything. Here's how that should work in your Kubernetes-based network.

Your cloud-driven network needs to run a service mesh to facilitate connections between applications that are internal to your Kubernetes system and as a tool for deploying end-to-end encryption. Kubernetes will deploy without service meshes if you leave them out, so you must remember to put service meshes in your ZTA. Some service meshes allow traffic to flow through your internal applications without access control by default.

Remember, zero trust means nothing is automatically trusted, regardless of its origin. Network security tools are needed internally for east-west traffic as much as for the north-south traffic between your internal network and an external network (most often the internet).

If your Kubernetes system also connects to external networks, you should use an Ingress controller. Your Ingress controller can be configured to run smoothly with your service mesh so that traffic in all directions flows well, with robust authentication and access control throughout.

When we first discussed authentication, I mentioned how service meshes and Ingress controllers can automate access control. All access monitoring is done with the logs and metrics feeds they generate. This offloads all that work from the rest of your Kubernetes system.

If you use third-party cloud network tools such as NGINX Plus, NGINX Ingress Controller, and NGINX Service Mesh in your Kubernetes network, your Ingress controller and service mesh can work together. Some Ingress controllers can be set up as the egress endpoint of the mesh, ensuring secured communication throughout the ingress and egress routes.

Integrating north-south and east-west traffic in this way streamlines access control monitoring in your Kubernetes system, giving your administrators and security analysts fewer panels to watch. They'll definitely appreciate that! You can set all access monitoring through HTTP and TCP/UDP at your Ingress controller. Here's how you can do so and also use live activity monitoring to make it easier to manage security responsively.

Your Ingress controller and service mesh can generate two sets of logs: a runtime log (an error log) and an access log. Events are written to the runtime log when network operation problems occur at various severity levels. But for access monitoring specifically, you need to implement the access log. Each time a request is processed, an event is written there, regardless of whether the request was successful or not.

If you use log analysis and monitoring software from different vendors or platforms, you can send some logs through many of those third-party systems through syslog. Syslog is a tried-and-true Unix-based logging utility that's been used since the 1980s. Syslog is a platform-independent standard that can be useful for cross-platform compatibility. Just configure your logging to use the `syslog:` prefix in each log record, perhaps like this: `syslog:server=<`*`location address here`*`>`.

Some logging solutions can be fed into live monitoring applications, which makes it easier for security professionals to watch for network anomalies. Effective monitoring can improve the security posture in your ZTA.

Let's Review

Implementing effective monitoring tools and processes in your Kubernetes application helps tremendously in maintaining effective zero trust security.

Here are some tips:

- Monitoring logs is vital to your zero trust posture. You can't secure what you can't see, and you can't identify problems if your application is opaque.

- Now that you can produce access logs, it's important to aggregate that information. Once again, the Cloud Native Computing Foundation ecosystem provides many options for these services.
- Observability specifications are collapsing into one standard. Familiarize yourself with OpenTelemetry, and understand how its architecture can help you in your monitoring journey.

CHAPTER 7

Encrypt All Access

One of the main objectives in the cybersecurity field is for applications and networks to encrypt as much data as possible, both in storage and in transit. End-to-end encryption ought to be the standard for data in transit through networks. Thorough encryption helps protect the confidentiality and integrity components of the CIA triad of cybersecurity. As nothing should be trusted by default, regardless of its origin, establishing end-to-end encryption in all directions is essential for zero trust security.

Ultimately, all access in your Kubernetes-based cloud network must be encrypted. Here's how we can make sure all network encryption is automated effectively.

East-West Traffic

The service mesh can be set up to offload encryption and identity management work within your internal network from your Kubernetes clusters and their applications. Mutual TLS (mTLS) is the encryption protocol that can be used for that purpose, ensuring that all the data traveling within your internal network is ciphertext.

Configure mTLS in your service mesh with the "Strict" setting. "Off" disables mTLS between your Pods, and communication from all sources would be sent as cleartext. Unless you're testing something in a development environment, that's a terrible idea. The default setting in many Kubernetes-compatible service meshes is "Permissive," which enables mTLS between Pods but also permits cleartext traffic where mTLS cannot be established. Avoid that if at all possible, and make sure you use "Strict" in your production network. Specific settings may differ based on your service mesh. Please check it carefully!

All TLS implementations use certificates as identities, mTLS included. Check for self-signed root certificates. That's another default you must change! Instead, integrate proper PKI with an upstream root CA. The SPIRE open source toolchain can be used as an identity management interface, and from there you can include an upstream root CA with SPIRE's UpstreamAuthority mechanism.

Here are the SPIRE-compatible upstream authorities that the NGINX Service Mesh supports. Service meshes from other vendors may also support these SPIRE-compatible upstream authorities. Please read the vendor's documentation to verify that!

- If your cloud uses the AWS infrastructure, you can choose *aws_pca* for the Amazon Certificate Manager Private Certificate Authority (ACM PCA). Alternatively, *awssecret* can be used for CA credentials from AWS Secrets Manager.
- If you use the Vault PKI, *vault* is an option. Choose *cert-manager* use an instance of cert-manager running in Kubernetes to request intermediate signing certificates for the SPIRE server.
- Finally, if your enterprise handles PKI in-house, choose *disk* as your upstream authority. That, of course, requires certificates and private keys to be on disk!

North-South Traffic

In addition to setting up encryption for your east-west traffic, you should set it up for your north-south traffic as well, especially if your Kubernetes network accesses the internet.

This can be configured through Kubernetes APIs and NGINX Ingress Controller APIs (even a cert-manager integration). Refer to Kubernetes and NIC documentation for help with this.

Of course, all network traffic should be encrypted, which is possible with proper configuration of the Ingress controller and service mesh. You'll also need to obtain SSL server certificates and SSL client certificates.

In addition, your upstream server (or each server in the upstream group) will need to get server certificates and a private key. If it's possible for your enterprise to run its own CA and PKI, an SSL library, such as OpenSSL, can be used to generate certificates. Otherwise, it's best to choose a trusted third-party certificate authority (CA).

SSL client certificates are used as identities between NGINX and upstream servers. Client certificates are stored on the NGINX platform and need to be signed by a trusted CA. Let's Encrypt, Comodo, and other popular CAs can be used for client certificate signing. Once you have PKI access and identities set up, you can enable SSL in the NGINX configuration file.

Let's Review

To ensure effective zero trust security in your Kubernetes application, you must encrypt as much data as possible, both in transit and in storage. Now that you understand how you can apply mTLS to north-south and east-west traffic, you can look beyond your network. Here are some things to think about:

- Where else do you have data in transit?
- Where do you have data at rest?
- Extending beyond just encryption, where else in your system do you need to apply identity management?

CHAPTER 8

Effective Zero Trust Security Scenarios with Kubernetes

Massively Multiplayer Online Robbery Pwning Game

Deploying a backend for a massively multiplayer online (MMO) game isn't easy. It's difficult to determine how many players a game's servers will need the capacity and bandwidth to support until the game has had a successful stable launch and has grown its player base.

In the early days of MMO development, before cloud platforms became popular, a game development company often had to purchase its own bare-metal servers to run in on-premises network infrastructure. If an MMO exploded in popularity because *IGN* or *PC Gamer* gave it a positive review—as great as that would be for revenue—the company might have a sudden capacity nightmare. When thousands of new players find they can't log on or they're entering a super-slow game world, the company soon loses business.

But then Amazon, Google, Microsoft, and many other tech giants began offering infrastructure as a service (IaaS), which solved many scalability and responsiveness issues. By the mid-2010s, Docker debuted, and Kubernetes followed a couple of years later. Containerization empowered developers to deploy applications that are as scalable as the cloud environments in which they operate.

In our example, Alex and Jack started their MMO careers at indie developers that were restricted by the technological limitations of the early 2000s. But by the 2020s, the industry had acquired enough experience with Kubernetes and cloud platforms to provide plenty of support to indie developers who wanted to make MMOs that were as big and flashy as *The Elder Scrolls Online* and *Fortnite*.

With help from Kubernetes, Agones game servers, and Unreal Engine, Alex and Jack deployed their first cloud-driven MMO, *Shinjuku Shinto World*. The beta-testing release ran for nearly eight months. The game had a total of 5,246 unique players the first month, 15,349 the second, and 2,349 the third. As bandwidth and capacity demands fluctuated frequently, the developers were grateful for the scalability that Kubernetes made possible.

But by the fourth month, something unexpected happened. Hundreds of users reported that the credit cards they used to pay for their monthly subscriptions were being exploited for financial fraud.

Alex and Jack hired a business lawyer and compensated their customers who were financially victimized by the cybercriminals who penetrated their gaming backend. The incident was an expensive wake-up call about the importance of deploying Kubernetes with zero trust security.

The developers announced on their website, social media, and email list that the *Shinjuku Shinto World* beta would go offline for a few weeks so that they could security-harden their backend and redeploy.

During that time, they hired a third-party cybersecurity firm to investigate the incident and determine which vulnerabilities were exploited and how. Alex and Jack had a good PKI, but there were many vectors in their network where it wasn't being used. There were a lot of vectors with no authentication capabilities, and there was a route to their point-of-sale system that lacked proper TLS encryption.

They redeployed the Kubernetes network with an Ingress controller to authenticate and encrypt user data coming into their network through the internet, and a service mesh to secure traffic between their point-of-sale system and their game servers. Their network was now zero trust, and the Ingress controller also helped with load balancing. All traffic flowing through *Shinjuku Shinto World* was now encrypted, and robust logging and monitoring were making malicious exploits much easier to detect and mitigate responsively.

Months later, Alex and Jack presented *Shinjuku Shinto World* at the Game Developers Conference (GDC) to promote the game's official stable launch. Thanks to that event and loads of word-of-mouth advertising, their game now has over 100,000 unique players each month, and they've left that expensive PR nightmare behind them.

Stay at 127.0.0.1, Wear a 255.255.255.0

Mississauga Medical Supply has been selling nonpharmaceutical medical supplies directly to consumers and smaller clinics through their website since 2005. Glucometer testing strips, vinyl gloves, bandages, oral thermometers—you name it, they've got it. In 2011, they relaunched their ecommerce site with Shopify's platform and through a traditional web hosting provider. Shopify's platform made it so much easier to present their inventory on the site and manage point of sale.

Total annual revenue grew from $2.3 million in 2012 to $50.2 million by 2019. Mississauga Medical Supply utilized ecommerce to become the little business that could!

When the COVID-19 pandemic erupted in March 2020, Mississauga Medical Supply was hit with a new problem and a new opportunity. CEO and founder Faye Cassar was an epidemiologist before she became an entrepreneur, so she takes pandemic safety seriously. Mississauga Medical Supply's 78 employees worked out of a small office space and warehouse. But now she understood the need for her workforce to go remote.

The warehouse needed at least two employees at all times from 9 a.m. to 5 p.m. on weekdays. But everyone else—the buyers, web developers, office administrators, and customer service team—could do their jobs from home.

Throughout the second quarter of 2020, Mississauga Medical Supply transitioned from a simple retail website with a traditional web hosting provider to a cloud provider in order to run Shopify, their web UI, and Kubernetes, as well as to deploy an internal web app with the Vue.js JavaScript framework to support their newly remote employees.

The ecommerce site runs in a vanilla AWS instance, and the employee web application runs in Amazon EC2, both as separate networks. Each has an Ingress controller and a service mesh. For the ecommerce site, the Ingress controller authenticates and manages user connections while customers browse and shop, while a service mesh secures and balances traffic between the Shopify deployment and the point-of-sale system. For the employee web app, an Ingress controller gives workers a secure way to access the company's internal web forum, webmail, group messaging, and sensitive documents. A service mesh operates between the company's communication services and documentation database as well.

Whenever a customer uses the internet to buy something from Mississauga Medical Supply's site or an employee has a chat with another employee, every single network vector is part of the company's zero trust security architecture.

Business boomed with the new consumer demand for masks and Rapid Antigen Test kits. Total annual revenue grew from $90.2 million in 2020 to $110.5 million by 2021. Kubernetes and AWS were elegantly scalable to the company's sudden growth.

Kubernetes and ZTA: Empowering Business in the 21st Century

There are probably thousands, if not millions, of businesses worldwide and across industries that have utilized Kubernetes in zero trust architecture (ZTA) to conduct ecommerce with the utmost of efficiency and security. The private sector, the public sector, small businesses, and big corporations around the world are deploying internet interfaces for their customers, clients, stakeholders, employees, and contractors through these technologies.

Regardless of the kinds of products or services your organization produces, Kubernetes is a powerful tool for deploying responsive and scalable applications through the cloud. Using ZTA effectively is necessary to keep your valuable applications and data resources secure.

Conclusion

Kubernetes has quickly become the industry-leading container orchestration platform. Before Kubernetes became popular, most enterprises that were implementing cloud services in their networks and internet-driven applications would install virtual machines without containerization, similar to how they'd run virtual machines on bare-metal servers on premises. The advent of cloud platforms like AWS and Azure gifted enterprises with immense scalability, addressing a lot of infrastructure management woes that are inevitable when dealing with on-premises networks. But scaling the applications inside the scalable hardware resources that the cloud provides was easier said than done.

Kubernetes certainly didn't invent containerization. Docker has been around since 2013, Solaris Containers beta was released in 2004, and FreeBSD "jails" were a hosting provider's revolutionary experiment in 2000. But there's no doubt that Kubernetes has a design that accelerated the adoption of containerization and made it possible for businesses to take full advantage of the power of the cloud.

Kubernetes works well within a zero trust environment. The Kubernetes API has built-in role-based access control (RBAC) and TLS support. Kubernetes has plenty of ways to be integrated into zero trust architecture (ZTA). Kubernetes documentation helps developers learn how to integrate the platform into their security controls, but no security controls will work without deliberately adding them to your Kubernetes deployment and configuring them carefully. Kubernetes can be secure, but it's never secure out of the box. So this report is your guide to the core concepts of deploying Kubernetes with the zero trust security model.

There's more to learn about zero trust security and Kubernetes than can fit into this report. With that in mind, here are some excellent online resources you can reference to learn more.

A lot of the best resources for support specific to how Kubernetes works are straight from the source:

- Official Kubernetes documentation
- Kubernetes debugging documentation
- Official Kubernetes Community Forums
- Kubernetes' Slack channel

These unofficial resources can also be very helpful:

- The [Kubernetes] tag on StackOverflow
- Kubernetes on Reddit

Some of the top cloud providers have their own Kubernetes support resources. This can be extra helpful if you need support specific to using Kubernetes on their platforms:

- Kubernetes on AWS
- Google Kubernetes Engine (GKE) documentation
- Azure Kubernetes Service (AKS) documentation
- VMware Tanzu Kubernetes Grid documentation
- Oracle Cloud Infrastructure Container Engine for Kubernetes documentation

There are also some other resources you may want to explore:

- *Cloud Native DevOps with Kubernetes*, 2nd edition, by Justin Domingus and John Arundel
- NGINX documentation
- *Kubernetes: Up & Running*, 3rd edition, by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson
- *Networking & Kubernetes* by James Strong and Vallery Lancey
- *Zero Trust Networks* by Evan Gilman and Doug Barth
- *Cloud Native Security Cookbook* by Josh Armitage

Deploying Kubernetes in ZTA is often the most effective way to deploy fully scalable applications on the cloud while integrating security that works. Cloud networks and containerization platforms like Kubernetes require a security model that's much better suited to networks with "software-defined perimeters," or no perimeter at all, depending on your point of view.

The zero trust security model and Kubernetes were both experimental ideas a few years ago, but they have now matured into useful tools that enterprises in all industries rely upon to serve clients, customers, and entire supply chains every minute of every day. If you work in DevOps, software as a service, or web application development, experience in zero trust security and Kubernetes is now a necessary skill set. It's all pretty cool technology, but some knowledge is required to get the most out of it. You can do it!

About the Author

Kim Crawley is dedicated to researching and writing about cybersecurity issues. She's honored to be featured in the first volume of *Tribe of Hackers*. She's worked for Sophos, AT&T Cybersecurity, BlackBerry Cylance, Tripwire, and Venafi. She's currently working on cybersecurity content for IOActive and her upcoming O'Reilly Media title *Hacker Culture: A to Z*. In her spare time, she loves playing Japanese video games and cohosting the HACKERverse podcast with Craig Ellrod.