Brought to you by

vmware

Container Network Security

for dummies®

A Wiley Brand

VMware Special Edition

Map attack vectors in microservices architectures

Discover the power of the NetworkPolicy API

Enforce Kubernetes best practices and policies

Haim Helman

Manish Chugtu

Jay Vyas

Susan Wu

About VMware

VMware software powers the world's complex digital infrastructure. The company's cloud, app modernization, networking, security, and digital workspace offerings help customers deliver any application on any cloud across any device. Headquartered in Palo Alto, California, VMware is committed to being a force for good, from its breakthrough technology innovations to its global impact. For more information, please visit www.vmware.com/company.html.

Container Network Security

for dummies®
A Wiley Brand

Container Network Security

VMware Special Edition

by Haim Helman, Manish Chugtu, Jay Vyas, and Susan Wu

for dummies®
A Wiley Brand

Container Network Security For Dummies®, VMware Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN 978-1-119-81121-3 (pbk); ISBN 978-1-119-81122-0 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Elizabeth Kuball

Acquisitions Editor: Ashley Coffey

Editorial Manager: Rev Mengle

Business Development Representative: Cynthia Tweed

Production Editor: Mohammed Zafar

Special Help: Pere Monclus, Stijn Vanveerdeghem, Yves Fauser

Table of Contents

Introduction

The trend of moving networking into software and compute layers started a decade ago, but containers and the use of Kubernetes for container orchestration has been accelerating this trend. Simply put, there are no Kubernetes clusters without container networking, there are no containerized applications in production without security, and there is no shared infrastructure without segmentation.

About This Book

Container Network Security For Dummies consists of five chapters that explore:

- How to recognize the attack vectors in the software supply chain (Chapter 1)
- How to use Kubernetes NetworkPolicy to secure the software development chain (Chapter 2)
- How to harden your application workloads (Chapter 3)
- How to secure network communications (Chapter 4)
- Helpful resources to learn more about container network security (Chapter 5)

Each chapter is written to stand on its own, so if you see a topic that piques your interest feel free to jump ahead to that chapter. You can read this book in any order that suits you (though we don't recommend upside down or backward).

Foolish Assumptions

It has been said that most assumptions have outlived their uselessness, but I assume a few things nonetheless!

Mainly, I assume that you're an application developer, a security engineer, or a cloud architect. As such, I assume you have at least some understanding of application development, containers and orchestration, and network fundamentals.

If any of these assumptions describes you, then this is the book for you! If none of these assumptions describes you, keep reading anyway! It's a great book and after reading it, you'll know quite a bit about container network security.

Icons Used in This Book

Throughout this book, I occasionally use special icons to call attention to important information. Here's what to expect:

REMEMBER

This icon points out important information you should commit to your nonvolatile memory, your gray matter, or your noggin!

TECHNICAL STUFF

If you seek to attain the seventh level of NERD-vana, perk up! This icon explains the jargon beneath the jargon.

TIP

Tips are appreciated, but never expected, and I sure hope you'll appreciate these useful nuggets of information.

WARNING

These alerts point out the stuff your mother warned you about (well, probably not — but they do offer practical advice).

Beyond the Book

There's only so much I can cover in this short book, so if you want to learn more check out `https://vmware.com`.

IN THIS CHAPTER

- Looking at the rise of container adoption and security risks
- Identifying attack vectors in the development environment
- Mapping attack vectors in microservices and networks

Chapter 1
Recognizing Attack Vectors in the Software Supply Chain

The Sonatype *2020 State of the Software Supply Chain Report* declared it's "open season" on open-source software projects, as evidenced by a 430 percent increase in next-generation cyberattacks actively targeting these projects since 2019. At a high-level, the modus operandi for adversaries attacking the software supply chain is to inject malicious code upstream into open-source projects and then target downstream applications after the malicious code has made its way through the supply chain.

In this chapter, I introduced you to the different attack vectors that adversaries use to compromise development environments, microservices architectures, and networks.

Understanding Security Risks Associated with the Rise of Container Adoption

Containerized applications can be very dynamic. Compared with VM-based applications, containerized applications typically have a high rate of change. According to industry analysts, nearly three-quarters of global organizations will be running three or more containerized applications in their production environments by 2023. The Cloud Native Computing Foundation (CNCF) also confirmed a similar pattern in its survey which found the use of containers in production has increased to 92 percent since 2019.

With Kubernetes being the prevalent container orchestration solution, 32 percent of respondents in the CNCF survey indicated that security is one of their top three challenges to using containers. You need to carefully evaluate the risks that may be introduced in your software supply chain through containers and take appropriate precautions to harden your workloads and secure the container environment — and the data in them.

Attack Vectors in the Development Environment

Adversaries often target integrated development environments (IDEs) and weak credentials used in repositories and registries to taint popular software development kits (SDKs) and code libraries, or to obtain hard-coded secrets (that is, passwords).

In the following sections, I describe several common attack vectors that adversaries frequently leverage to compromise development systems.

SecOps practitioners often map threats using the MITRE Adversarial Tactics, Techniques & Common Knowledge (ATT&CK) framework, which provides a thorough analysis of the techniques a threat actor may employ, as well as potential mitigations, during the last five stages of the seven-stage Cyber Attack Lifecycle (Recon, Weaponize, Deliver, Exploit, Control, Execute, Maintain) — in other words, everything but Recon and Weaponize.

Although a separate framework for container security does not exist today, the Linux Matrix within the MITRE ATT&CK Matrix for Enterprise discusses attack vectors for the Linux platform and is presently the most directly applicable model for containers.

Account credentials

Adversaries routinely scan public repositories in search of privileged account credentials that can be compromised, as well as sensitive log information and weak configuration settings that can be exploited.

Backdooring is a popular method used to infiltrate accounts. Adversaries insert malicious code into seemingly innocuous packages that create a backdoor for attackers to use after the host package is installed in an application.

Successful phishing attacks also can result in account hijacking. A compromised account can be used to introduce and execute malicious code in the repository (for example, to exfiltrate sensitive data or mine cryptocurrency on a large number of production containers).

Finally, although logs can be an essential debugging tool, they can also be a rich source of sensitive information and configuration settings, which adversaries can use to compromise downstream applications. For example, privileged access logs may expose account credentials to adversaries.

WARNING

These are not hypothetical risks: developer account takeovers, including backdoor admin accounts and backdoor Secure Shell (SSH) accounts, are happening more and more frequently.

Container images

Container images from an unsanctioned repository can introduce vulnerabilities into applications. One common way to exploit a microservices architecture is to use an official and popular *vanilla* (that is, not customized) base image, like Ubuntu or Alpine, which is typically not designed to deliver a payload or run malicious code. The payload is delivered and initiated by the entrypoint command, which in turn downloads malicious components during runtime.

CSO Online reported that, of the images hosted on the Docker Hub repository, more than half have at least one critical vulnerability. The popularity of public registries hosting premade software components and images has led to attackers publishing code on these package repositories, making it even easier to exploit the software supply chain.


REMEMBER

Maintaining image hygiene is critical to prevent container bloat and to ensure that outdated images don't become an attack vector that can be used to introduce vulnerabilities.

Application dependencies

Developers build distributed applications using various components, including libraries, frameworks, and other software modules. These application dependencies (and nested dependencies) can themselves pose a security risk. For example, dependencies can be backdoored and vulnerabilities or outdated components can be nested deep to hide their existence.

Modern applications often make use of 100 or more software components, of which as much as 90 percent can be open source. Research from the University of Darmstadt, published in August 2019, revealed that nearly 40 percent of all npm (originally, an abbreviation for Node Package Manager) packages rely on code with known vulnerabilities.


REMEMBER

Outdated components with known vulnerabilities should not be used in modern application development. It's important to maintain good security hygiene by removing outdated software versions that may have potentially vulnerable components that can be exploited by adversaries.

Image registries

Developers download free open-source component releases from public open-source repositories in order to build their applications. The use of public image repositories (for example, Docker Hub, Maven Central, and npm) introduces security risks because defective and known vulnerable components can make their way through software supply chains into production code.

Host-container relationships

In self-managed scenarios, direct access to the host operating system (OS) of the master node is possible via port exposure or an exploitable condition. A container or pod is allowed to escalate privileges, resulting in host access. Containers can't provide 100 percent isolation for your applications in the same way that running your applications on separate hosts or virtual machines (VMs) can. Containers can also be exposed to kernel privilege escalation attacks.

Unsecured orchestrator platforms

Excessive developer privileges in orchestrator platforms, such as Kubernetes, can pose a significant risk. The Kubernetes community curates a collection of failure stories to share best practices on operations and security. One of these stories details the attack vectors in the Kubernetes platform. A video software player company found a cryptominer running as a binary on the `root` directory of one of their host machines (not from a container). This was due to an exposed load balancer using a specific Kubernetes cluster configuration that allowed arbitrary code to break out of the container and run on the host instance. The attacker was able to execute this exploit by masquerading the cryptominer as a legitimate filename to avoid detection. The cryptominer then exploited a monitoring process that was running in the privileged mode to move laterally through the Kubernetes deployment. Fortunately, in this case, the damage was limited to consuming 100 percent of the CPU core on their dev and test instances, and there were no successful exploits into their production clusters.

Adversaries can exploit the Kubernetes platform in many ways, including via the following methods:

- An exposed application programming interface (API) server that allows control of Kubernetes clusters.
- Unencrypted secrets harvested from the `etcd` key-value store, which contains encryption keys, database credentials, API keys, and other sensitive key materials or authentication data for container services.
- An exposed kubelet, which could allow for worker node control.

TECHNICAL STUFF

A *kubelet* is an agent that runs on each node in a Kubernetes cluster and ensures that containers are running in a pod.

- An unpatched authentication bypass vulnerability in the kube-proxy network proxy, or a kube-proxy that has anonymous access enabled.
- An unpatched vulnerability in a container runtime that enables container breakout and access to the host system.

Attack Vectors in Microservices Architectures and Networks

Unlike legacy monolithic and *n*-tier applications, modern cloud-native applications are composed of hundreds (sometimes thousands) of microservices that are often deployed as containers across numerous on-premises data centers and cloud environments. Every microservice that's part of an application can be a unique network endpoint that exposes an attack surface.

An inbound (north–south) request from an end-user application can result in hundreds of east–west transactions between containers in a microservices architecture. This east–west traffic — including inter-process communications, databases, and application layer protocols — is generally not mediated by API gateways or other security controls, so attackers take advantage of relatively unrestricted lateral movement within the environment.

Inter-process communications

Message brokers such as Apache Kafka, Rabbit MQ, and Redis are used as the back-end glue of a microservices architecture. However, after access to the queue is permitted, no further authorization checks are performed.

TIP

Additional messaging-level controls, like message signing, may not be necessary in the development environment, but they should be strongly considered in production environments to enhance security.

Databases

Monolithic applications tend to make use of a single, centralized database. In a microservices architecture, there's a tendency to use different database tables for each microservice to increase agility and promote independence. However, there are many attack techniques defined by MITRE that an adversary can use to exploit a microservices architecture, such as using a compromised container to perform malicious data queries and data exfiltration.

Unlike traditional application workloads, databases used in a microservices architecture can be stored practically anywhere and are, thus, challenging to protect.

Finally, whereas traditional relational databases have centralized controls (like private tables, private schemas, and private databases), in a microservices architecture there are no traceability tools (such as database audit logs and database management tools). So, an attacker can gain unauthorized access to sensitive data in a microservices architecture using a variety of techniques that take advantage of this relatively open environment.

Application layer protocols

Any application, whether a traditional *n*-tier or microservices application, can be susceptible to application layer attack techniques, such as SQL injection and cross-site scripting (XSS). Additionally, many containerized applications can interact with monolithic applications, thereby exposing a larger attack surface between the applications.

Attackers may also communicate with their command-and-control (C2) infrastructure using application layer protocols that are normally associated with web traffic. In this way, they can avoid detection and network filtering by blending their C2 traffic in with legitimate traffic. Commands from the remote C2 infrastructure (and often, the results of those commands) can be embedded in legitimate traffic between the client and server using common web protocols, like Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS). An attacker uses these protocols to communicate with the compromised systems within a targeted network, while also mimicking normal expected traffic to avoid detection.

North–south attacks

Attackers may also exploit weaknesses in public-facing web applications or databases. If the application is hosted on cloud-based infrastructure, an application exploit can compromise the underlying instance and take advantage of weak identity and access control policies in a north–south attack.

WARNING

The threat landscape for microservices architectures has shifted. Organized criminal organizations are increasingly attacking cloud-native application environments and investing heavily in attack infrastructure to find exposed and vulnerable hosts in order to achieve criminal objectives for financial gain.

IN THIS CHAPTER

- Creating your first network policy
- Understanding the role of Container Network Interface (CNI) providers
- Taking network policies to the next level
- Verifying your network policies work with end-to-end testing

Chapter 2
Securing the Software Development Supply Chain

The Kubernetes NetworkPolicy application programming interface (API) was created so that developers could write policies that define internal connectivity and security requirements for their applications. In this chapter, I dive right into a few examples to demonstrate how to use the NetworkPolicy API.

Getting Started with the NetworkPolicy API

Network policies created with the NetworkPolicy API generally include the ability to:

- Block (and later, allow) incoming connections from pods, namespaces, or Classless Inter-Domain Routing (CIDR) blocks
- Block (and later, allow) outgoing connections to pods or namespaces
- Block (and later, allow) communications to specific ports

There are three steps to creating network policies:

1. **Create a default policy.**

   First, select a pod you're applying a policy to. When you apply a policy to a pod, you're effectively blocking its ability to receive traffic from the outside world (if you create an "ingress" policy) or blocking its ability to send traffic to the outside world (if you create an "egress" policy).

   Next, open up firewall rules. After you apply a policy to a pod, you need to add ingress and/or egress rules to that policy.

   Let's create a network policy and apply it to a pod that has the `role: my-app-that-needs-to-be-more-secure` label:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: my-first-policy
 namespace: default
spec:
 podSelector:
  matchLabels:
   role: my-app-that-needs-to-be-secure
```

   Write this policy file out to `policy.yaml` on your local machine, and then run `kubectl create -f policy.yaml` to create the policy. Your pods will now be inaccessible from the outside world.

2. **Expand your policy to allow traffic from a namespace.**

   A network policy that totally isolates a pod from ingress traffic is not very practical for a web server. A web server should, of course, accept traffic from web clients. To allow this traffic, you need to add an ingress rule that defaults to `Ingress`, by including the `policyTypes:` field in your policy. You can re-create your original policy by rewriting the file and running `kubectl apply -f policy.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: my-first-policy
 namespace: default
```

```
spec:
 podSelector:
  matchLabels:
   role: my-app-that-needs-to-be-secure
 policyTypes:
 - Ingress
 ingress:
 - from:
  - namespaceSelector:
      matchLabels:
       podgroup: web-client
```

At this point, you now have a network policy that will block traffic to Pods matching the Pod selector, with the exception of namespaces that are labeled **`podgroup:`** `web-client`.

3. **Verify that your policy works.**

To test your policy, you can create a nginx-daemonset (these are ideal for network policy testing because you'll see the same firewall rules on all nodes). Save your daemonset file as `ds.yaml`, and then run `kubectl create -f ds.yaml`. In a few moments, you'll have a nginx daemonset running on all nodes of your cluster:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: nginx-ds
spec:
 selector:
  matchLabels:
   app: web
 template:
  metadata:
   matchLabels:
    role: my-app-that-needs-to-be-secure
    app: web
  spec:
   containers:
   - name: nginx
    image: quay.io/bitnami/nginx
```

Now, you can run a few experiments on your own because you have a web server running on port 443 of the pod's IP address for this nginx pod. For example, you can check whether your cluster has network policies enabled. Create a busybox pod, which you can use to run your experiments using the following deployment:

```
apiVersion: v1
kind: Pod
metadata:
 name: ubuntu
 namespace: default
spec:
 containers:
 - image: quay.io/baselibrary/ubuntu
  command:
   - sleep
   - "36000"
  name: ubuntu
 restartPolicy: Always
```

Next, create a busybox pod in the default namespace. Now, create a shell in your busybox pod (you can do this by running `kubectl get pods` and then running `kubectl exec -t -i <POD_NAME> -- /bin/sh`). Then, inside your busybox pod, you can try to curl the IP off your nginx pod, which can be obtained by running `kubectl get pods -o wide`. For example, you might run `kubectl exec -t -i <POD_NAME> -- curl 10.1.2.3:8080`, where 10.1.2.3 is the IP address for your nginx pods.

In any case, if this command hangs, then it means your policies are working.

What if you relabel that pod to contain **`podgroup: web-client`** in its pod specification? If you set up the preceding example correctly, then opening a new shell into one of your busybox pods and rerunning the preceding commands should result in traffic being allowed to the nginx pod.

Now, create a Kubernetes service that load balances traffic to the nginx daemonset. Can you curl traffic to the daemonset through the service IP address from this busybox pod?

As you can see from these examples, network policies can be powerful security tools for your applications. They allow you to place firewalls around your application — without modifying your application — because, in most cases, they're entirely enabled by your container networking plugin.

Looking at How Network Policies Work with Services

One of the experiments you (hopefully) ran in the previous section involved trying to access traffic from a protected pod from inside, and outside, of a namespace. In that case, traffic was blocked unless your busybox pod had the right labels. So, how is Kubernetes smart enough to block pod traffic even through a load balancer?

The answer is the Container Network Interface (CNI). Network policies are applied by a CNI provider, which means that regardless of how you accessed a pod — for example, whether you're using a translated IP address (using network address translation, or NAT) to that pod through a load balancer — your CNI provider will still be able to block traffic, because your CNI is acting directly on the machine that your pod is running on.

WARNING

Unless you're running a cluster with a CNI provider that supports the NetworkPolicy API (such as Antrea, Calico, or Cillium), the policies you create may not actually be applied and your containers will be wide open to traffic from anywhere.

Depending on your CNI provider, network policies are implemented using a different set of technologies. For example:

- If you use Calico, then your policies are written as iptables rules. This is accomplished by the calico/node container, which continuously writes new iptables rules when it sees changes to labels on pods, namespaces, or network policies.

» If you use Antrea, the exact same thing occurs except that instead of modifying iptables rules, the antrea-agent (which is the architectural equivalent of the calico/node) will continually update openvswitch (OVS) rules.

You can dig into your Antrea containers and see exactly how policies are implemented. Because you ran nginx in a daemonset in the previous examples, any node on your cluster will exhibit the same OVS rules for blocking traffic to an nginx pod. Specifically, you'll be able to see the conjunction rule on table 90 in the following example:

```
$> kubectl -n kube-system exec -it antrea-agent-
<POD_ID> -- ovs-ofctl dump-flows br-int | grep
table=90

Defaulting container name to antrea-agent.

Use 'kubectl describe pod/antrea-agent-<Pod_ID> -n
kube-system' to see all of the containers in this
pod.

 cookie=0x2000000000000, duration=344936.777s,
table=90, n_packets=0, n_bytes=0, priority=210,ct_
state=-new+est,ip actions=resubmit(,105)

 cookie=0x2000000000000, duration=344936.776s,
table=90, n_packets=83160, n_bytes=6153840,
priority=210,ip,nw_src=100.96.26.1
actions=resubmit(,105)

 ### This line shows that you have some pods which
are being allowed, via the ovs flow register, into
the cluster....

 cookie=0x2050000000000, duration=22.296s, table=90,
n_packets=0, n_bytes=0, priority=200,ip,reg1=0x18
actions=conjunction(1,2/2)
```

```
 cookie=0x2050000000000, duration=22.300s,
table=90, n_packets=0, n_bytes=0, priority=190,
conj_id=1,ip actions=load:0x1->NXM_NX_REG6[],
resubmit(,105)

 cookie=0x2000000000000, duration=344936.782s,
table=90, n_packets=149662, n_bytes=11075281,
priority=0 actions=resubmit(,100)
```

In OVS (the technology that powers Antrea), a series of tables is used to define routing rules, and table 90 is where the routing ingress rules occur. So, you'll see changes occurring at the OVS level in this table as you create or destroy network policies in an Antrea-powered cluster.


TIP

If you use Calico, you can do similar experiments to see the underlying network policy implementation by comparing the output of `iptables-save` before and after you create a policy on a pod.

Exploring Other Types of Network Policies

The Kubernetes.io documentation has several other examples of network policies that you can adapt and use as templates for your policies. The following example shows a complete policy that ties together ingress, egress, and multiple matching predicates for a policy selector:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: my-real-netpol
 namespace: default
spec:
 podSelector:
  matchLabels:
   role: db
 policyTypes:
 - Ingress
 - Egress
```

```
ingress:
- from:
 - ipBlock:
    cidr: 172.17.0.0/16
    except:
    - 172.17.1.0/24
 - namespaceSelector:
    matchLabels:
     Project: trusted-ns
 - podSelector:
    matchLabels:
     role: trusted-pod
 ports:
 - protocol: TCP
   port: 6379
egress:
- to:
 - ipBlock:
    cidr: 10.0.0.0/24
 ports:
 - protocol: TCP
   port: 5978
```

This policy has an `ipblock`, `namespacesSelector`, and `pod Selector`. This means that the only traffic that would be allowed into it must be from the `trusted-ns` namespace and also from the `trusted-pod` pod. Furthermore, incoming traffic must be in the 172.17 subnet. So, pods that are not in your cluster's 172.17 subnet would be rejected. Finally, the only traffic that would be allowed into this pod must be on Transmission Control Protocol (TCP) port 6379.

But what if you want to allow communication across a range of ports? Several improvements in Kubernetes 1.21 provide greater flexibility in network policies:

- **Port ranges:** In future Kubernetes releases, you'll be able to define port ranges, instead of just using individual ports. This will make it easier, for example, to create a network policy for a File Transfer Protocol (FTP) server or a virtual machine (VM) running as a Kubernetes pod. It will also allow

administrators to easily make default policies that only allow traffic in a specified port range.

- **Namespace name policies:** This is a new feature in the Kubernetes API that will add a default namespace label to any namespace in a cluster. This means you can easily build namespace label selectors for a policy, even if you don't know about the existing labels on that namespace. This problem is common in clusters where users don't have permission to read namespace labels but still want to be able to write network policies.

Using the Kubernetes End-to-End Tests

The Kubernetes community has released a powerful set of NetworkPolicy validation tests that use truth tables to tell you exactly how your cluster responds to a specific policy, with several example pods that probe one another.

To run these tests, you need to run the Kubernetes e2e.test library or use a tool (such as Sonobuoy) to filter out the Netpol testing suite. You can use these tests to verify that your cluster has a perfectly functioning, high-performance network policy provider, allowing you to easily switch between networking technologies over time without losing your security model.

Table 2-1 shows a sample output of the Kubernetes NetPol test suites for a cluster that is wide open (that is, a cluster where `pod a` in `namespace x` can talk to `pod a` in `namespace y`, and so on) for all pods in the namespaces being tested.

TABLE 2-1 Pod Connectivity before Policies on namespace x Are Created

	x/a	x/b	x/c	y/z
x/a	.	.	.	.
x/b	.	.	.	.
x/c	.	.	.	.
y/a	.	.	.	.

After applying a network policy (for example, one that blocks all traffic going into `namespace x`), the truth table output of the e2e. test program would look like Table 2-2.

TABLE 2-2 Pod Connectivity after Policies on namespace x Are Created

	x/a	x/b	x/c	y/z
x/a	x	x	X	.
x/b	x	x	X	.
x/c	x	x	X	.
y/a	x	x	X	.

REMEMBER

Different CNIs do different things for loopback policies, so the ability of x/a to connect to x/a when a policy blocks all ingress to namespace x will be different. The Kubernetes end-to-end tests don't "fail" if the expected connectivity from a pod to itself is different from what is defined by the NetworkPolicy API, but they do print out the result of this connection. In other words, the NetworkPolicy API behavior is not defined for loopback connections and varies across different CNIs.

To run the end-to-end testing NetworkPolicy validation suite from the source, follow these steps:

1. **Clone Kubernetes from** `https://github.com/kubernetes/kubernetes`.
2. **Ensure you have Golang 1.16+ and Docker installed.**

   You'll also need a C++ complier like g++ (GCC).
3. **Change directory (cd) to the Kubernetes directory and run** `make WHAT=test/e2e/e2e.test`.
4. **Export the** `KUBERNETES_SERVICE_HOST=https://<your cluster apiserver IP>` and `KUBECONFIG=<path to your kubeconfig>`.
5. **Run** `./_output/local/bin/linux/amd64/e2e.test --provider=local --dump-logs-on-failure=false --ginkgo.focus='Netpol'`.

TIP

If you're running macOS, your final `_output/local/bin/` command might instead look like `./_output/local/bin/darwin/amd64/e2e.test` because the end-to-end testing binary will have been built for macOS rather than Linux.

This testing suite consists of about 30 tests and takes anywhere from 5 to 20 minutes, depending on how fast your network is and how quickly your cluster can create and destroy pods. If you see failures, make sure they're relevant. For example, many clusters don't support Stream Control Transmission Protocol (SCTP) or Internet Protocol Version 6 (IPv6), so any network policy tests involving SCTP or IPv6 might be expected failures. In conventional scenarios, you can skip these tests by using the following commands:

```
./_output/local/bin/linux/amd64/e2e.test --provider=
local --kubeconfig=/home/ubuntu/.kube/config --dump-
logs-on-failure=false --ginkgo.focus='Netpol'
--ginkgo.skip="SCTP|UDP|6"
```

TIP

Running the Kubernetes end-to-end tests can tell you a lot about your cluster, and it isn't just a tool for network policies. If you want to integrate this tool more deeply into your daily workflow, you may want to use a tool, such as Sonobuoy (go to `https://github.com/vmware-tanzu/sonobuoy`), which runs the entire end-to-end test suite inside of a pod and offers a convenient command line tool, which you don't have to compile.

IN THIS CHAPTER

- Securing Kubernetes resources
- Implementing best practices and addressing PodSecurityPolicy limitations

Chapter 3
Hardening the Workload

Kubernetes does much more than scheduling containers to run on a cluster of servers. It provides a very powerful application programming interface (API) that allows developers to specify the resources needed to run their application, including compute, network, and storage, as well the privileges the application will have in the server it's running on and in the cluster as a whole.

Before letting your developers unleash the power of the Kubernetes API, you should understand the potential risks that may be introduced into your environment with each configuration attribute and the Kubernetes mechanisms that help administrators control risk by setting security guardrails for their developers — both of which I cover in this chapter.

Addressing Risky Configurations in Kubernetes Resources

Many out-of-the-box Kubernetes resources and configurations (and potential misconfigurations) can introduce risk into an application environment. In the following sections, I take a closer look at these resources and configurations.

Container runtime

A *container runtime* is the software that manages and configures the components required to run containers. It makes it easier to securely execute and efficiently deploy containers. A container isn't a first-class object in the Linux kernel. Instead, it's a combination of multiple settings of kernel namespaces, cgroups (which define how much CPU and memory can be used), and Linux Security Modules (LSMs, such as selinux).

Open Container Initiative (OCI) runtimes are low-level runtimes in charge of configuring the kernel resources for running a container. RunC is by far the most commonly used. Instead of using the native constructs for isolating containers, some OCI runtimes use sandboxing (for example, gVisor) or virtualization (for example, Kata and Firecracker) to create more strict separation between the container and its host.

Another type of runtime implements the Container Runtime Interface (CRI). These high-level runtimes provide orchestrators, such as Kubernetes, a uniform way to manage the container life cycle and monitor running containers. The most popular examples of CRI implementations are `containerd` and CRI-O (Container Runtime Interface plus OCI).

Containers, like virtual machines (VMs), provide a consistent runtime environment for processes and isolate processes from each other and from the underlying operating system (OS). However, unlike VMs, which run a separate OS, containers generally share the host's OS and rely on kernel namespaces and resource groups (that is, Linux cgroups) for isolation and resource management. Although this "soft" isolation is configurable, there isn't an inherent property that guarantees any kind of isolation. Administrators must be aware of the level of privilege each container has in the server it's running on and the resulting risk of an escape from a compromised container to its hosting server and the rest of the cluster.

Let's take a closer look at some common pod configurations and risks associated with them:

- **Privileged containers:** Although they may run in separate namespaces, these containers have all the capabilities of a root user and full access to the host's kernel and devices. In

terms of security, they offer no isolation and should be treated as privileged host processes.

- **Run as user:** By default, processes in containers will run as the root user (user id 0), which is also the root user in the host. Although the namespace and cgroup isolations will still apply for those processes, the risk of container escape by leveraging some vulnerability is much higher for processes running as root. Running a container as root, while very easy to do, is almost always a misconfiguration.
- **Capabilities:** When a container is privileged or running as root, it has all the capabilities provided by the Linux kernel. In some cases, the container only needs a specific set of capabilities, such as access to the networking subsystem. It's always better to run the container as a non-root user and to specify the kernel capabilities it requires, if any.
- **Host namespace:** In general, each container should run in separate network, process ID (PID), and inter-process communication (IPC) namespaces. However, containers can be configured to share one or more of these namespaces with their host. Although useful at times, such configuration increases the chance of container escape and lateral movement. For example, in Kubernetes, containers running in the host's network namespace cannot be isolated using network policies.
- **Lack of resource quotas:** By default, there is no limit to the amount of CPU or memory that processes in a container can use. Without explicitly setting those limits, a compromised (or just misbehaving) container can easily disrupt the stability of the host it's running in and (in some cases) the entire cluster, by consuming all the host's resources, causing processes running in other containers and directly on the host to be starved or killed.

Network exposure

In Kubernetes, the service object determines how one or more containers will be exposed in the network. Each service has a:

- **Selector:** Identifies the pods (which encapsulate one or more containers) that will be exposed.

- **Type:** Determines whether the exposure is internal to the cluster (ClusterIP type), via a port in the host (NodePort type), or via a load balancer (LoadBalancer type). Although the ClusterIP service is always internal, NodePort and LoadBalancer can be internal or external and can be protected via firewall (on-premises) or security group (cloud) rules.

WARNING

Simply by changing the type of a service resource, a developer can significantly increase the security risk associated with an application by exposing it, and all its vulnerabilities, to the Internet.

Network policies are another Kubernetes resource that can be used to control how pods are exposed over the network. By default, pods in Kubernetes allow all inbound and outbound communication. NetworkPolicy objects tell the Container Network Interface (CNI) which connections should be allowed. A tight network policy will reduce both the probability of attack (by limiting inbound connections) and its impact (by limiting outbound connections).

Role-based access control

Because Kubernetes offers a powerful API that can provision compute, network, and storage resources, it requires a way to control who can call which API. For authorization, Kubernetes uses role-based access control (RBAC). A `Role` resource defines a set of Kubernetes resource kinds and a set of operations that are permitted on resources of any of those kinds. A `RoleBinding` resource associates a role with a subject. A subject could be a user, a group, or a service account. Whereas users and groups identify people, service accounts identify an application. By default, each namespace in Kubernetes has a service account and each pod in that namespace has a token mounted, which it can use to authenticate itself as part of that service account.

By default, service accounts aren't bound to a role, so there is no inherent risk in providing the token to each pod. However, as soon as the first application is deployed that requires some access to the cluster's resources, great care has to be taken to enforce the least privilege principle by providing only the necessary privileges to each application. This is accomplished in the following ways:

- **By enumerating the specific resource kinds and operations allowed by each role instead of using a wildcard (*):** It's unlikely that an application actually needs to access every kind of resource or perform every kind of operation.

- **By only binding explicitly defined service accounts to roles:** Binding the default service account may lead to new pods in the namespace getting unintended privileges.
- **By using `Roles` and `RoleBindings` instead of `ClusterRoles` and `ClusterRoleBindings` whenever possible:** The "Cluster" versions of these resources are not scoped to a specific namespace and are, therefore, much more powerful. They should be used sparingly and audited frequently.

Volumes

By default, containers can only access their local filesystem, which is ephemeral. This behavior isolates the application running in the container by preventing it from accessing data it isn't supposed to or modifying files that are used by other applications.

However, in some cases an application needs to access files that are not part of its image or its local filesystem. This could be files from the host's filesystem or files provided by the orchestrator (such as Kubernetes secrets or configuration maps).

Containers support this requirement by allowing volumes to be mounted to the running container. Volumes could be mounted as read-only or as writable.

When mounting read-only volumes, you need to ensure that they don't contain any sensitive data beyond what's needed by the application. For example, a developer may prefer to mount a top-level directory from the host, such as `/var` or `/etc`, in order to make sure that an application has all the information it needs. However, these directories contain many files, and it's nearly impossible to verify that none of them includes credentials that could be used to compromise the host or even the cluster. A secure volume mount should include a relatively small number of files required for a specific use.

Mounting volumes as writable should be done with even greater care because the container and any application running in it can create risk by modifying files that will be used by the host or other container. By modifying executable files or configuration files, malware could be injected or application behavior could be altered to serve an attacker. Therefore, any data in a volume that is mounted as writable to a container should not be trusted.

Secrets

Often, applications running in containers require credentials in order to authenticate themselves when accessing other services. These credentials can't be stored securely in the container image or the workload configuration manifest, where access is hard to monitor and control. Instead, they should be provided to the container only when it runs. In Kubernetes, the `Secret` resource is used to store such credentials. Secrets can be exposed to a container as files in a mounted volume or as environment variables.


TIP

The most common examples of secrets are the service account tokens, which are used by applications in containers when accessing the Kubernetes API server.

Because secrets hold sensitive information, special care must be taken to monitor who can access a secret, either by mounting the secret in a new pod, entering an existing container that uses that secret, or directly via the Secrets API provided by the Kubernetes API server.

Custom resource definitions

The Kubernetes API is based on the concept of a resource. An API call declares the desired state of a resource, and a controller in the Kubernetes control plane performs the necessary actions to get to that state. The way to extend the Kubernetes APIs is to define custom resources beyond the built-in ones (such as pods, deployments, and services). An `Operator` is a common pattern for a user-defined application, which acts as a controller handling a specific set of custom resources.

Although the security implications of the different attributes of built-in resources can be researched and understood to help you create a policy that governs the configuration of built-in resources, this is not the case for custom resources.

Minimizing the number of custom resource definitions (CRDs) and limiting the privileges of the controller handling each CRD is necessary, to avoid introducing new attack vectors to a cluster that may result from misconfiguring or manipulating the configuration of these resources.

Other Kubernetes APIs that are risky at runtime

Most of the APIs provided by Kubernetes have to do with declaring the desired state of resources. However, a couple of APIs affect pods in runtime and, therefore, introduce another kind of security concern. The exec and port-forward APIs are very useful for debugging purposes, but in production environments they could be used by an attacker to compromise an application, either from within its container or through its network interfaces:

- **Exec API:** This API, most commonly used via the `kubectl exec` and `kubectl cp` commands, replaces Secure Shell (ssh) as the way to remotely run commands in a running container inside Kubernetes. Such commands can be used to read secrets that are exposed to the pod or to attempt lateral movement inside the cluster (or to other servers that are accessible from the cluster). However, unlike using ssh to connect directly to the container, the privilege to perform `exec` can be eliminated or limited via RBAC, and these calls are logged in the cluster's audit log.
- **Port-forward API:** Port forwarding allows redirection of inbound connections from a pod to a local host and vice versa. It bypasses the service resource definition and any network policies that apply for that pod. Like `exec`, port-forwarding is meant to help when developing, testing, and debugging applications, but it's a dangerous tool in the hands of an attacker because it can intercept or initiate network requests to the target pod, even when that pod doesn't expose its network interface outside the cluster.

Enforcing Best Practices and Policies in Kubernetes

The first major effort to identify best practices for securely deploying container runtimes and Kubernetes clusters, and hardening applications running in those clusters, was done by the Center for Internet Security (CIS). The CIS Kubernetes Benchmark and CIS Docker (Containers) Benchmark are the foundation for many of the controls required by compliance standards and provided by

open-source and commercial security tools. Additional resources are available from the Kubernetes community through the Cloud-Native Computing Foundation (CNCF) and from the commercial providers of Kubernetes, such as the major public cloud providers.

PodSecurityPolicy is a kind of resource offered by Kubernetes to enforce polices and best practices for new pods in a cluster. Most rules in these policies have to do with the pod's runtime attributes such as `privileged`, sharing the host's namespaces and additional Linux capabilities. Other rules address the volumes mounted to the pod.

PodSecurityPolicy can also be used to define more secure defaults for new pods, such as preventing privilege escalation or turning off some default capabilities. When applying these kinds of rules, the PodSecurityPolicy doesn't just validate, but actually mutates, the pod's configuration.

Although the PodSecurityPolicy is a powerful tool that's offered natively in Kubernetes, it does have some shortcomings that limit its adoption:

- **It only applies to pods.** It doesn't cover other kinds of resources such as services, nodes, or roles. Also, if a deployment is configured to create pods that will fail the policy check, the deployment resource will be applied successfully, but it won't be able to create any pods.
- **Scope is hard to define.** The way to define which policies apply to a pod is by using RBAC. However, these configurations can be complicated to create, maintain, and debug in clusters where many users run many applications in many namespaces.
- **Pod rejection events may be hard to find and understand.** There is no easy way to monitor all the events of pod rejection, and when such an event is found, it may be hard to tell which rule in which policy is responsible for the rejection.
- **There is no "audit-only" mode.** There is no way to test which pod would fail a policy check before actually applying that policy and re-creating the pods, which may disrupt the operation of the application running in the cluster.
- **It's not extensible.** There is no way to add new rules other than updating Kubernetes itself.

In order to overcome the limitations of the PodSecurityPolicy tool, Kubernetes supports custom admission control via a validating webhook. In this approach, one or more servers are defined (known as *admission controllers*), which will either approve or reject Kubernetes API calls. The Kubernetes API server is configured to send a webhook to each admission controller on some, or all, incoming API calls. If all webhooks return as approved, the API call will be executed; otherwise, it will be rejected.


TIP

A validating webhook admission controller configured to enforce workload security best practices, combined with a tight RBAC policy are the best ways to minimize the security and operational risks introduced to a cluster via the Kubernetes API.

IN THIS CHAPTER

- Making Ingress access secure
- Securing east–west communications in a microservices architecture

Chapter 4
Securing Network Communications

Modern cloud-native applications are dynamic, adaptive, and highly distributed, with hundreds of microservices deployed as containers servicing the requirements of rapid feature releases, high resilience, and on-demand scalability. These containers are deployed on multiple nodes in a single Kubernetes cluster, and quite often across multiple clusters and clouds.

In this chapter, we explain how to manage and secure access to and from your application (north–south traffic), as well as communications between the microservices of the application itself (east–west traffic).

Managing and Securing Ingress Access

External access to an application running on a Kubernetes cluster is done by exposing the service using the following options:

- `Service.Type = NodePort`: When you declare a service with the `NodePort` option, it exposes the application on a

port across each of the nodes in the cluster. The application can then be accessed from outside the cluster using `<NodeIP>:<NodePort>`.

- `Service.Type = LoadBalancer`: When you declare a service with the `LoadBalancer` option, it creates an external load balancer that points to a Kubernetes service in the cluster. Most cloud providers support this option to provision the cloud load balancer and route external traffic to the service.
- **Ingress API:** The Ingress application programming interface (API) is a high-level abstraction that allows you to manage external access to the services inside a Kubernetes cluster. Ingress is not a service type; it's another Kubernetes resource that acts as a reverse-proxy (typically, Layer 7) and manages traffic routing to different services within a cluster. Even though, Ingress is a core concept of Kubernetes, the implementation is done using an external third-party proxy, which is known as an *Ingress controller.* Some examples of Ingress controllers are Contour, nginx, and VMware NSX Advanced Load Balancer (using Avi Kubernetes Operator, or AKO).

  Different Ingress controllers have extended the Ingress specification in various ways to support additional use cases. Some Ingress controllers support custom resources apart from Ingress, and other controllers extend the functionality using custom annotations. The most noteworthy of these controllers is Contour, which supports both the HTTPProxy API and the Ingress API. One of the notable differentiators of the HTTPProxy API is that it allows administrators to configure top-level ingress settings (for example, Virtual Hosts per team) and then allows lower-level configuration to be delegated to an individual team.


TIP

  The desire for advanced use cases has led to issues with different implementations and features among Ingress controllers. The Kubernetes community is currently working on several proposals to provide a more standardized and highly configurable set of APIs, as an alternative to Ingress.
- **External API gateway:** Because of additional service/API requirements at the Ingress level — like rate limiting, multi-protocol support, authorization and authentication, and so on — another option is to use an external API

gateway solution — such as Spring Cloud API Gateway, Kong Gateway, and others — to expose the service/API. Some API gateways (like Ambassador and Gloo) also function as an Ingress controller.

- **Ingress gateway:** In some service mesh implementations (like Istio and VMware Tanzu Service Mesh), Ingress functionality is provided by a gateway proxy, which is a part of the service mesh itself and is configured using APIs provided by the mesh. Ingress gateways in a service mesh provide more flexibility than the Ingress API and also allow service mesh features like encryption and routing rules to be applied to the incoming traffic.

Securing Ingress traffic

The basic security configuration provided within the Kubernetes Ingress API uses Transport Layer Security (TLS) to secure your ingress traffic, which assumes that TLS is terminated at the Ingress.

As described in the Kubernetes documentation, "Ingress is secured by specifying a 'Secret' resource that contains a TLS private key and certificate. The TLS secret must contain keys named 'tls.crt' and 'tls.key' that contain the certificate and private key to use for TLS," as shown in the following example:

```
apiVersion: v1
kind: Secret
metadata:
  name: example-secret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

The secret is referenced in the Ingress API, which tells the Ingress controller to secure communications from the client to the load balancer using TLS. This configuration requires that the certificate used to create the TLS secret contain a Common Name (CN) — also

known as a fully qualified domain name (FQDN) — for the host, such as `example-tls.acme.com` in the following example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-tls-ingress
spec:
  tls:
  - hosts:
      - example-tls.acme.com
    secretName: example-secret-tls
  rules:
  - host: example-tls.acme.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

Depending on the Ingress controller being used, other encryption methods are also supported, including the following:

- **Mutual TLS (mTLS):** Using mutual authentication, both server and client authenticate each other's identity by presenting valid certificates to each other. This method is considered more secure than the TLS option shown in the previous example.
- **Secure Sockets Layer (SSL) passthrough:** In cases where you would like TLS to be terminated by the application itself rather than at the Ingress, you can enable the SSL passthrough option, supported by some advanced Ingress controllers like nginx and VMware NSX Advanced Load Balancer. In this scenario, the incoming SSL request is not decrypted at the load balancer. Instead, it's passed along to a server for decryption. This option is preferred when application security is of top concern.

TIP

Because TLS encryption requires certificates, it's important to look at options that would enable users to manage these certificates in an easy and automated way. One option is to use a certificate management controller, like cert-manager, which is native to Kubernetes and automatically provisions SSL certificates for your services.

Implementing authentication and authorization

Apart from traffic encryption, you may also need to validate an incoming request and provide access to allowed services or an API, based on who the end user is. Various methods are supported (depending on the Ingress controller being used or, in some cases, by using API/Ingress gateways). Some of these methods include the following:

- **Basic Auth:** Hypertext Transfer Protocol (HTTP) Basic Auth is a simple method that creates a username and password authentication for HTTP requests. Using this type of authentication mechanism, user credentials are encoded and sent along with the request in a standard header.
- **API keys:** API keys provide a way to authenticate an application accessing an API, without the need for referencing an actual user. The application adds the key to each API request, which is used to identify the application and authorize the request. Depending on the API, the mechanism that sends the keys may differ. For example, some APIs may use authorization headers, query parameters, and/or the body parameters, and so on.
- **Open Authorization (OAuth):** OAuth is a token-based architecture that relies on the fact that the service receives a token issued by a trusted third-party as proof that the application can connect to the service. For example, in OAuth 2.0 flows, an external Identity Provider is used to authenticate and provide user identity through an Access Token.
- **OpenID Connect (OIDC):** OIDC is an extension that is built on OAuth 2.0 and provides some additional standardizations, where the identity provider is required to also return an ID token apart from an access token. The ID token is a JavaScript Object Notation (JSON) Web Token (JWT)

containing identity information. The format of the ID token is digitally signed, self-contained, and compact. The information provided in the ID token can be passed down the transaction chain and used to apply fine-grained attribute-based access policies. Service Mesh implementations, described later in this chapter, use this mechanism for implementing authorization policies within the mesh.

Configuring other Ingress security features

Additional security-related features can be configured at an Ingress layer. Some of these features are included in advanced Ingress controllers, while others may require a custom gateway (like an API Gateway). Some of these features include the following:

- **Web application firewall (WAF):** A WAF protects web applications by intercepting and inspecting network packets for threats, such as distributed denial-of-service (DDoS) attacks and Open Web Application Security Project (OWASP) vulnerabilities. A WAF processes traffic based on a set of rules, such as the OWASP rule set or custom rules, to determine if access to the application needs to be blocked. Advanced Ingress controllers, like VMware NSX-ALB with iWAF, offer a comprehensive set of web application security features and helps organizations achieve and maintain regulatory compliance, for example, with the General Data Protection Regulation (GDPR), Health Insurance Portability and Accountability Act (HIPAA), and Payment Card Industry Data Security Standards (PCI DSS).
- **Rate limiting:** To prevent your API from being overwhelmed by too many requests, some Ingress controllers and API Gateways also include options for rate shaping and throttling of the traffic.

Depending on the Ingress controller or API Gateway implementation, the position of the different security functions (shown in Figure 4-1) can be interchanged. Some implementations, like VMware NSX Advanced Load Balancer, consolidate all or some of these functions in a single Ingress solution.

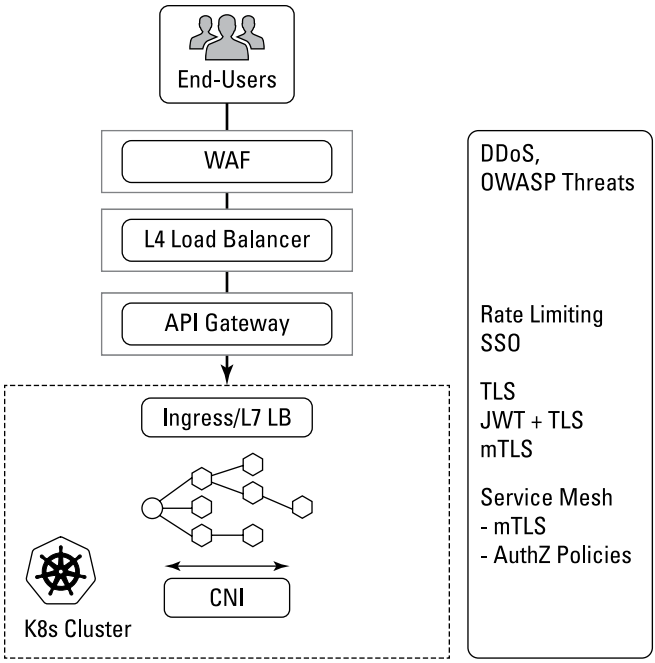


FIGURE 4-1: Additional security functions of a WAF, load balancer, and API Gateway can be used to secure the Ingress API.

Managing and Securing Traffic between Microservices

With a microservices-based architecture, securing communication between the services within an application becomes as important as securing ingress traffic, particularly given that these containers may be deployed across multiple clusters and clouds. This is where a service mesh is needed.

A service mesh is a platform layer that provides applications with features like service discovery, resilience, observability, and security, without requiring application developers to integrate or modify their code to use these services. All these features are abstracted from developers using an architecture in which the communication between all the services happens via a sidecar proxy, which sits alongside each service, creating a service mesh.

Some examples of service mesh solutions include Istio, Linkerd, VMware Tanzu Service Mesh, Consul, and Kuma. In most service mesh implementations, sidecars are managed and configured by the centralized control plane for traffic routing and policy enforcement. In Kubernetes, the injection of the sidecar alongside an application container is transparent to the user and happens automatically (see Figure 4-2).

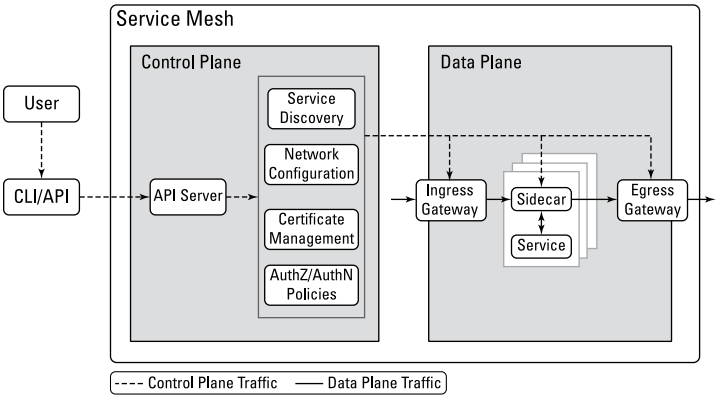


FIGURE 4-2: High-level architecture of a service mesh.

TIP

Learn more about the sidecar injection model in Istio at `https://istio.io/latest/blog/2019/data-plane-setup`.

A service mesh provides secure connectivity by encrypting the traffic between services. It also manages authentication and authorization of service communications at scale. Authentication policies are provided at two levels: service-to-service authentication (mTLS) and request authentication (at Ingress).

Service-to-service authentication (mTLS)

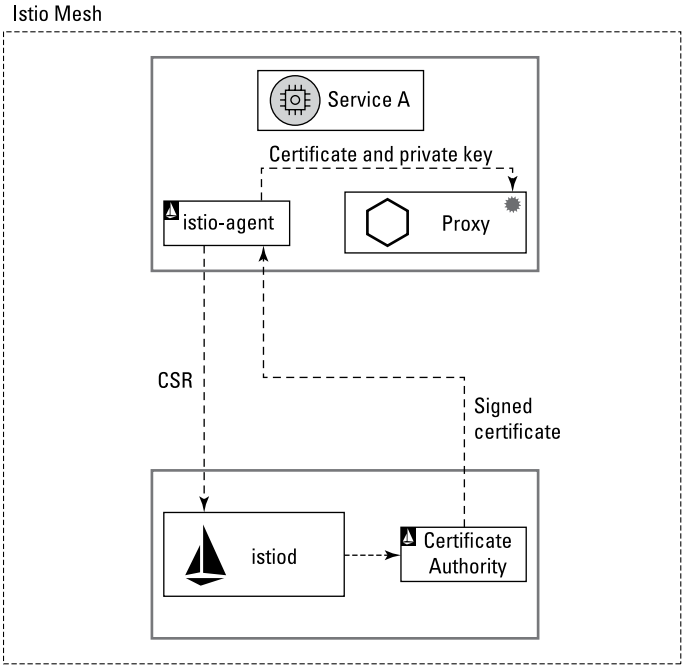
A service mesh offers transparent mTLS between the services inside the mesh. The service mesh provisions strong identities to every service within a mesh with certificates, which are used to establish mutual authentication. For this purpose, the service mesh control plane also includes a Certificate Authority (CA) for key and certificate management and automates their generation, distribution, and rotation at scale. The service identity becomes very important in securing the mesh, as it may also represent

its role. There are multiple ways of determining the identity of a service.

Istio's identity model uses the first-class service identity to determine the identity of a service. Some examples that Istio can use for service identities on various platforms (per Istio's documentation) include the following:

- Kubernetes service account on Kubernetes
- Google Cloud Platform (GCP) service account on Google Compute Engine (GCE)
- On-premises non-Kubernetes (Istio can use other identities like user account, custom service account, or even service names that can group workload instances)

Figure 4-3 shows how identity provisioning is done in Istio.



Source: https://istio.io/latest/docs/concepts/security

FIGURE 4-3: Identity provisioning workflow.

After the identities are determined, Istio provisions keys and certificates through the Envoy secret discovery service (SDS) using the following flow (as written in the Istio documentation):

1. `istiod` (Istio's control plane) offers a Google Remote Procedure Call (gRPC) service to take certificate signing requests (CSRs).
2. Envoy (sidecar proxy) sends a certificate and key request via the Envoy SDS API.
3. Upon receiving the SDS request, the Istio agent (which runs alongside each sidecar proxy) creates the private key and CSR before sending the CSR with its credentials to `istiod` for signing.
4. The CA validates the credentials carried in the CSR and signs the CSR to generate the certificate.
5. The Istio agent sends the certificate received from `istiod` and the private key to Envoy via the Envoy SDS API.

This CSR process repeats periodically for certificate and key rotation.

The following is an example of an Istio peer authentication policy that specifies that transport authentication for the workloads with the `app:acme` label must use mTLS (that is, STRICT mode).

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "example-acme-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: acme
  mtls:
    mode: STRICT
```


TIP

Some service mesh implementations, like Istio, also allow you to either disable mTLS or configure a PERMISSIVE mode, which allows the services to accept both plaintext and mTLS traffic. This capability helps to easily migrate applications to mTLS in a non-disruptive manner.

Request authentication (at Ingress)

Most service meshes provide their own gateway implementation for controlling Ingress traffic to the mesh. In some cases, these implementations also integrate with API gateways to provide advanced API management features. Because the Ingress Gateway in a service mesh is a part of the mesh, the policies of the mesh (like encryption and traffic routing) can also be applied to traffic from the Ingress Gateway to the back-end services.

Based on the service mesh implementation, the Ingress Gateway provides similar security capabilities as described in the "Securing Ingress Traffic" section in this chapter, whether it's securing traffic using TLS/mTLS at the Ingress or letting users provide authentication and authorization policies.

For example, in Istio, users can configure authentication policies to validate a JSON Web Token (JWT) in the request based on certain values, which among others, may include the following:

- The token location
- The issuer
- The public JSON Web Key Set (JWKS)

Based on the authentication policy, Istio validates the token to accept or reject the request.

The following is an example that requires a valid JWT for all requests for workloads that have the `app:frontend` label issued by `issuer-acme`.

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: frontend
  namespace: acme
spec:
  selector:
    matchLabels:
      app: frontend
  jwtRules:
  - issuer: "issuer-acme"
```

If no token is found, Istio accepts the request by default but won't provide an authenticated identity. To reject requests without tokens, a user needs to provide authorization rules that specify the restrictions for specific operations (for example, paths or actions).

Authorization

Using service mesh, users can enforce access control for the workloads within the mesh by configuring authorization policies. These policies are enforced at runtime by a server-side sidecar proxy on the incoming traffic. For each request that is passed through the proxy, the request context is evaluated by the authorization engine against the authorization policies, to either allow or deny the request.

Authorization policies can be configured based on the various attributes, which could either be based on service attributes like its identity, name, namespace, and so on, or even request attributes which are fetched at the authentication phase (for example, from JWT).

The following example of Istio's AuthorizationPolicy requires valid request principals (derived from JWT Authentication) for a request to `/admin` path. If not present, the request is denied.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: frontend-admin
  namespace: acme
spec:
  selector:
    matchLabels:
      app: frontend
  action: DENY
  rules:
  - to:
    - operation:
        paths: ["/admin"]
    from:
    - source:
        notRequestPrincipals: ["*"]
```

Advanced use cases

Security and infrastructure markets are rapidly evolving, and a service mesh can't assume it has access to all data for all attributes natively or expect to understand the relevant semantics. Some advanced service mesh implementations, like VMware Tanzu Service Mesh, also provide an extensible data integration framework to which third-party solutions can write a plugin. Using the additional data, Tanzu Service Mesh has the ability to access and aggregate this context and enable policy decisions against it. The context spans intrinsic data such as an inventory of users, services, data from various infrastructure platforms, and extrinsic data such as user and application life-cycle behavior (see Figure 4-4).

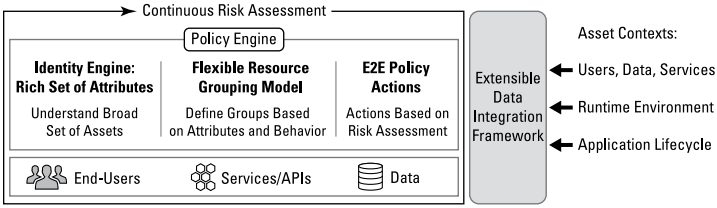


FIGURE 4-4: VMware Tanzu Service Mesh policy framework.

This context awareness in VMware Tanzu Service Mesh is accompanied by a flexible query language that allows resource groups to be defined based on a richer set of attributes (for example, all workloads exhibiting a certain type of vulnerability, or showing risky behavior based on their security posture scores, and so on). Using these resource groups, users can define more evolved access control policies that go beyond defining control on a hop-to-hop basis to specifying end-to-end transactions that need to be protected.

Also, as applications are becoming more distributed and can span multiple clusters or even clouds, it's becoming a requirement to be able to apply the security policies not only within the mesh microservices but also across them.

For example, VMware Tanzu Service mesh provides a strong and flexible construct called Global Namespaces (GNS), which can slice a cluster into many service mesh zones and extend each of them to multiple clusters and/or infrastructures (see Figure 4-5). Each GNS manages its own service discovery, observability, encryption, traffic and security policies, and service-level agreements (SLAs).

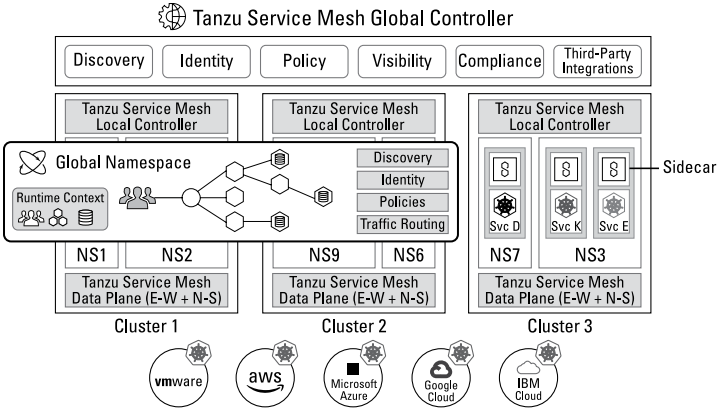


FIGURE 4-5: A GNS having workloads spanning across multiple clusters in Tanzu Service Mesh.

This enables enterprises to be able to apply consistent security policies within a GNS boundary, irrespective of where their application services are deployed (for example, mTLS between services within a GNS across multiple clusters or clouds).

Ensuring observability

The ability to monitor connectivity between services is the main driver for adoption of service meshes. The proxies provide statistics about the rate, latency, and failures of API calls.

More advanced observability, called *distributed tracing*, can be gained by enabling and monitoring span headers across API calls. These headers are propagated from an incoming API call to the resulting outbound calls. Distributed tracing allows developers and operators to identify the root cause of a slow or failed request to an application by evaluating each interaction between the different microservices involved in serving that request.

When it comes to security, observability serves multiple purposes, including the following:

- **Support for access control policy management:** Service-to-service traffic statistics are used to build a map of connections between services. This map can be translated to a least privilege access control policy, where only traffic that has been observed in the past is allowed in the future.

- **Threat detection:** Layer 7 traffic statistics can be used to create a behavior profile for each microservice. When the profile is stable, any significant deviation could be interpreted as an anomalous behavior that may be due to a compromise of that workload.
- **Attack investigation:** After a possible attack has been identified, the SecOps team starts investigating to identify the origin of the breach and its extent. In cloud-native environments where east–west traffic is dominant, distributed tracing capabilities are crucial for a quick and comprehensive investigation.

Pattern-based intrusion prevention systems (IPS) and deep packet inspection (DPI)

To prevent lateral movement between containers by an attacker, you need to insert IPS at the network level, but this requires the ability to monitor east–west traffic at scale. The challenge is that traditional IPS requires traffic to be backhauled to physical or virtual appliances, creating major scalability and latency issues while requiring you to completely re-architect your network — a time-intensive and error-prone process that introduces complexity. In addition, traditional IPS solutions do not have the application context to allow certain types of traffic to pass through while blocking other types of traffic.

Enterprise security in the microservices world requires a distributed and services-defined approach to IPS. Your IPS solution should provide visibility into all port-to-port traffic, allowing you to detect malicious traffic and use:

- Behavioral analysis such as network traffic analysis (NTA) and anomaly detection to recognize and stop abnormal behavior
- Deep packet inspection to decrypt and examine malicious traffic that may be hidden in Transport Layer Security (TLS) encrypted sessions

Delivering security as code to every workload at the hypervisor level ensures that all traffic — whether it's contained inside the data center or flows across multiple cloud providers, and whether segmentation has been applied — can be monitored for malicious content to prevent the lateral spread of attacks.

CUSTOMER SUCCESS STORY

Increased competition and rising customer expectations were putting pressure on a Fortune 50 financial services company. The company's leadership knew that it needed to be more agile while providing powerful, frictionless experiences to consumers and adhering to the company's strict security, audit, and compliance requirements.

How could it enable business agility while meeting robust security and regulatory requirements in a consistent manner?

The agility part was obvious. The company needed to embrace modern, cloud-native applications in a microservices architecture. However, making sure modern applications follow enterprise security policies and compliance is another issue altogether. Security is typically bolted on after the fact and tied to infrastructure rather than the actual workload, making it nearly impossible to know whether policies are being enforced appropriately at the app or user level.

The financial services company worked with VMware to modernize its security operations for multi-cloud, embracing a software-defined approach with central management across all environments. The idea was to tie security to each individual workload, so that each time a workload comes online a set of tags associated with an application will trigger the deployment of specific policies at each layer of the infrastructure stack across any underlying infrastructure. Policies stay with the workload and evolve as needed throughout its life cycle — whether it's in the public cloud or on premises, or it flows across multiple environments.

The financial services company followed five steps to successfully migrate its legacy apps to the cloud:

1. Define applications and security dependencies up front using GitOps.
2. Automate standard known dependencies, connectivity, and security posture through a workflow.
3. Immediately reject or flag applications that do not adhere to standard security posture. Exceptions can still be approved but need human interaction.

4. Use a service mesh for monitoring and enforcement to ensure that the correct policy has been implemented.

This intrinsic security framework creates a foundation for how users and applications connect and interact together while giving the IT organization an end-to-end understanding of security. The framework is simple, it's integrated seamlessly with legacy systems, and it can be extended to any environment.

IN THIS CHAPTER

- Reading analyst research, blogs, and books
- Taking courses and getting certified
- Watching demos and presentations
- Looking at documentation and product pages
- Referencing frameworks and participating in special interest groups
- Viewing videos and webinars

Chapter 5
Ten Resources to Help You Get Started with Container Network Security

Ready to get started? The following resources and tutorials will enhance your understanding of container network security and help you get started.

Analyst Research

Get an independent analyst's view on the state of container security:

- **Forrester Research: Best Practices for Container Security:** `https://hello-tanzu.vmware.com/best-practices-for-container-security/`

- **Gartner's Best Practices for Running Containers and Kubernetes in Production:** www.gartner.com/en/documents/3988395/best-practices-for-running-containers-and-kubernetes-in-

Blogs

Many container network security experts are blogging about lessons learned and sharing their knowledge on how to secure modern applications. Follow their conversations:

- **Multi-Cloud Connectivity and Security Needs of Kubernetes Applications:** https://blogs.vmware.com/networkvirtualization/2021/05/multi-cloud-connectivity-security-kubernetes.html
- **Announcing the General Availability of Container Security in VMware Carbon Black Cloud:** www.carbonblack.com/blog/announcing-the-general-availability-of-container-security-in-the-vmware-carbon-black-cloud/
- **VMware to Help Customers Make Modern Apps More Secure with Intent to Acquire Mesh7:** https://blogs.vmware.com/networkvirtualization/2021/03/vmware-announces-mesh7.html
- **Forging a Path to Continuous, Risk-based Security with VMware Tanzu Service Mesh:** https://blogs.vmware.com/networkvirtualization/2020/03/risk-based-security.html

Books

When you're looking for ways to secure container networks in your organization, get practical guides from technical experts with previous delivery experience:

- ***Core Kubernetes*** by Jay Vyas and Chris Love (www.manning.com/books/core-kubernetes)

- ***Learn Kubernetes Security*** by Kaizhe Huang and Pranjal Jumde (www.amazon.com/dp/1839216506)
- ***Container Security*** by Liz Rice (https://www.amazon.com/dp/1492056707)

Courses and Certifications

Developers and platform operators alike need to learn how to secure applications and platforms. Why not take a class to enrich your understanding? There are many free and low-cost options, including the following:

- **Kubernetes Security Essentials:** https://training.linuxfoundation.org/training/kubernetes-security-essentials-lfs260
- **Kubernetes Security Fundamentals:** https://training.linuxfoundation.org/training/kubernetes-security-fundamentals-lfs460
- **Networking in Kubernetes:** https://kube.academy/courses/networking-in-kubernetes
- **Kubernetes Platform Security:** https://kube.academy/courses/kubernetes-platform-security
- **Kubernetes Networking Deep Dive:** www.ipspace.net/Kubernetes_Networking_Deep_Dive
- **Interacting with Kubernetes – Introduction to Ingress:** https://kube.academy/lessons/introduction-to-ingress
- **Kubernetes Security:** www.infosecinstitute.com/skills/courses/kubernetes-security
- **Service Mesh Fundamentals:** https://training.linuxfoundation.org/training/service-mesh-fundamentals-lfs243

The Linux Foundation offers a Certified Kubernetes Security Specialist (CKS) program to provide assurance that a CKS has the skills, knowledge, and competence on a broad range of best practices for security container-based applications and Kubernetes platforms during build, deployment, and runtime. Learn more

at https://training.linuxfoundation.org/certification/certified-kubernetes-security-specialist. Prep courses for the CKS certification include the following:

- **Kubernetes CKS 2021 Complete Course + Simulator:** www.udemy.com/course/certified-kubernetes-security-specialist
- **Abdennour T's References for CKS Exam Objectives – Certified Kubernetes Security Specialist:** https://github.com/abdennour/certified-kubernetes-security-specialist

Demos and Presentations

When you're ready to take a deeper dive into container network security, why not get a demo from the technical experts to help you understand what's going on "under the hood"? Take a look at the following demos:

- **Master Kubernetes with NSX:** https://nsx.techzone.vmware.com/kubernetes-nsx
- **Improving Container Security with VMware Tanzu Build Service and VMware Tanzu Application Catalog:** https://youtu.be/ZbtUfTsjaAs

View Tim Hockin's illustrated guide to Kubernetes Networking. Tim is a co-founder of the Kubernetes project and a principal software engineer at Google; he gives talks on Kubernetes, networking, storage, node, multi-cluster, resource isolation, and cluster sharing (https://speakerdeck.com/thockin/illustrated-guide-to-kubernetes-networking).

Documentation and Product Pages

When all else fails, read the manual! These links to official documentation and product pages will help you find the answers you need:

- **Kubernetes Service:** https://kubernetes.io/docs/concepts/services-networking/service
- **Kubernetes Networking:** https://kubernetes.io/docs/concepts/cluster-administration/networking
- **Antrea:** https://antrea.io/docs/v0.13.1
- **VMware Tanzu Service Mesh on Tanzu:** https://tanzu.vmware.com/service-mesh
- **VMware Tanzu Service Mesh on VMware.com:** www.vmware.com/products/tanzu-service-mesh.html
- **VMware Tanzu Service Mesh Documentation:** https://docs.vmware.com/en/VMware-Tanzu-Service-Mesh/index.html
- **VMware Carbon Black:** www.carbonblack.com/products/vmware-carbon-black-cloud-container
- **VMware NSX Distributed IDS/IPS:** www.vmware.com/products/nsx-distributed-ids-ips.html
- **VMware NSX Advanced Load Balancer:** www.vmware.com/products/nsx-advanced-load-balancer.html
- **VMware NSX Advanced Threat Protection:** www.vmware.com/products/nsx-advanced-threat-prevention.html
- **VMware Intrinsic Security:** www.vmware.com/security.html

Frameworks

The MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) framework is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations. Refer to the MITRE ATT&CK framework for common exploits to the Linux platform (https://attack.mitre.org/matrices/enterprise/linux).

Special Interest Groups

Join the Kubernetes Special Interest Groups to post questions and connect with your fellow container network security engineers in the industry:

- **Cloud Native Computing Foundation (CNCF) Special Interest Group on Security:** Secure access, policy control, privacy, auditing, and more (`https://github.com/cncf/sig-security`)
- **CNCF Special Interest Group on Networking:** Networking primitives, including load balancing, observability, authentication, authorization, policy, rate limiting, quality of service (QoS), mesh networks, legacy infrastructure bridging, and more (`https://github.com/cncf/sig-network`)

Videos

You can find a wealth of videos on container network security from practitioners and trainers:

- **Advanced Persistence Threats: The Future of Kubernetes Attacks,** by Ian Coldwater and Brad Geesaman (`https://youtu.be/auUgVu1lAWM`)
- **The Devil in the Details: Kubernetes' First Security Assessment,** by Aaron Small, Google, and Jay Beale, InGuardians (`https://youtu.be/vknE5XEa_Do`)
- **Keynote: SIG-Honk AMA Panel: Hacking and Hardening in the Cloud Native Garden** (`https://youtu.be/CAZ5s0zli6g`)
- **Keynote: Hello From the Other Side: Dispatches From a Kubernetes Attacker,** by Ian Coldwater (`https://youtu.be/3jGNjan6I3Y`)
- **Kubernetes/Container Security,** by Ian Coldwater, PSW #640 (`https://youtu.be/vuaoVUD-TJU`)

- **Kubernetes Security Best Practices,** by Ian Lewis, Google (https://youtu.be/wqsUfvRyYpw)
- **Kubernetes: Vulnerabilities, Efficiency, and Cloud Security | Cyber Work Podcast** (https://youtu.be/KKGePS4hj9M)
- **New Security Features In Kubernetes 1.18,** by Haim Helman, VMware (https://youtu.be/oRJtixU8Coc)
- **Seccomp Security Profiles and You: A Practical Guide,** by Duffie Cooley, VMware (https://youtu.be/0Puu8wsu2Zc)
- **Service Mesh Security in a Nutshell,** by Venil Noronha and Manish Chugtu, VMware (https://youtu.be/liu51fCC3N4)
- **The ABCs of Kubernetes Security,** by Roger Klorese and Danny Sauer, SUSE (https://youtu.be/tGg_IjPLB20)
- **The Path Less Traveled: Abusing Kubernetes Defaults**, by Ian Coldwater and Duffie Cooley (https://youtu.be/HmoVSmTIOxM)

Webinars

Tune into webinars to get caught up on the latest trends on container network security from industry experts:

- **Securing Containers and Kubernetes-Orchestrated Environments:** www.carbonblack.com/resources/securing-containers-and-kubernetes-orchestrated-environments
- **Achieve Application Scalability with Tanzu Service Mesh:** https://tanzu.vmware.com/content/webinars/mar-11-achieve-application-scalability-with-tanzu-service-mesh
- **Deploy Secure, Scalable Kubernetes Apps with Tanzu Service Mesh and Ingress Services:** https://info.avinetworks.com/webinars/kubernetes-apps-service-mesh-ingress-services

- **Securing and Accelerating the Kubernetes CNI Data Plane with Project Antrea and NVIDIA Mellanox ConnectX SmartNICs:** www.cncf.io/webinars/securing-and-accelerating-the-kubernetes-cni-data-plane-with-project-antrea-and-nvidia-mellanox-connectx-smartnics
- **Zero Trust Security for Cloud Native Apps:** www.cncf.io/webinars/zero-trust-security-for-cloud-native-apps

Secure the software supply chain

Recent events have put the security of software supply chains squarely in the sights of corporate boardrooms while developers remain under constant pressure to deliver new software to market ever faster. To accelerate innovation, developers often leverage open-source software (OSS) components. But the unfortunate reality is that threat actors no longer respect the unwritten code of honor in the open-source community and are now actively targeting OSS software components and public repositories. Clearly, development teams must now take proactive steps to ensure container network security.

Inside...

- Recognize attack vectors in dev environments
- Create policies with the NetworkPolicy API
- Leverage CNI providers
- Run end-to-end validation tests
- Secure Kubernetes resources
- Control ingress traffic
- Secure east–west communications

vmware®

Haim Helman (@haimhelman) is the CTO of Carbon Black App Security. **Manish Chugtu** (@chugtuM) is the Enterprise Technologist at the CTO Office in the VMware Network and Security Business Unit. **Jay Vyas** (@jayunit100) is a Member of Technical Staff, Kubernetes SIG-Windows Lead. **Susan Wu** (@susanwu88) is the Senior Product Marketing Manager at VMware.

ISBN: 978-1-119-81121-3
Not For Resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.