Kubernetes Security & Best Practices

Hardening, Governance, and Secure Operations in Production Clusters

Preface

Kubernetes has fundamentally transformed how organizations build, deploy, and scale software. What began as an open-source container orchestration project from Google has become the de facto standard for running production workloads — from ambitious startups to the world's largest enterprises. But with this extraordinary power comes an equally extraordinary attack surface.

Kubernetes was designed for agility, not security by default. Its distributed architecture, declarative configuration model, and vast ecosystem of extensions create a landscape where a single misconfiguration — an overly permissive RBAC binding, an unrestricted pod security context, an exposed API server — can compromise an entire cluster. The breaches we've witnessed in recent years confirm a sobering truth: attackers understand Kubernetes, often better than the teams defending it.

This book exists because securing Kubernetes demands more than surface-level familiarity. It requires a deep, systematic understanding of how Kubernetes components interact, where trust boundaries lie, and how threats propagate across the layers of a cluster. *Kubernetes Security & Best Practices* was written to be the comprehensive, practitioner-focused guide I wished existed when I first confronted the challenge of hardening production Kubernetes environments.

What This Book Covers

The scope of this book spans the full lifecycle of Kubernetes security — from foundational concepts to operational maturity.

We begin by establishing **why Kubernetes security is fundamentally different** from traditional infrastructure security and how to threat model Kubernetes clusters effectively. From there, we move methodically through the critical layers of defense: **API server hardening and authentication**, **RBAC and access control**, **pod security**, **namespace isolation**, **network policies**, and **ingress protection**. Each chapter provides not just theory but actionable configurations, real-world scenarios, and battle-tested recommendations.

The middle sections tackle the challenges that keep platform engineers awake at night — **secrets management**, **policy enforcement at scale**, **runtime threat detection**, and **audit logging for compliance**. We then shift left into the **software supply chain**, covering container image security and **secure CI/CD pipelines**, areas where vulnerabilities are often introduced long before a workload reaches a cluster.

The final chapters address what many security books overlook: the **anti-patterns** that silently erode your Kubernetes security posture, and the cultural and organizational shift required to build a true **DevSecOps practice** around Kubernetes. Five appendices provide ready-to-use resources — including an RBAC role template library, secure pod YAML examples, an incident response playbook tailored to Kubernetes, and a security learning roadmap for continued growth.

Who This Book Is For

Whether you are a **platform engineer** hardening clusters, a **security professional** extending your expertise into cloud-native environments, a **DevOps practitioner** building secure pipelines, or an **engineering leader** establishing governance standards — this book meets you where you are and takes you further.

No prior Kubernetes security expertise is assumed, but a working familiarity with Kubernetes concepts (pods, deployments, services, namespaces) will help you extract maximum value from every chapter.

How to Read This Book

The chapters are designed to be read sequentially, as each builds upon concepts introduced earlier. However, experienced practitioners may choose to navigate directly to specific domains of concern. The appendices are intended as living references you'll return to repeatedly in your day-to-day work.

Acknowledgments

This book would not exist without the extraordinary Kubernetes security community — the researchers who disclose vulnerabilities responsibly, the open-source maintainers behind tools like Falco, OPA, Trivy, and Kyverno, and the countless engineers who share their hard-won lessons publicly. I am also deeply grateful to the technical reviewers whose sharp eyes and honest feedback strengthened every chapter, and to my family for their patience during the many late nights this work demanded.

Kubernetes clusters are only as secure as the intention and knowledge behind their configuration. My hope is that this book gives you both — the clarity to understand what must be secured, and the confidence to secure it well.

Let's begin.

Dorian Thorne

Table of Contents

Chapter 1: Why Kubernetes Security Is Different

Kubernetes has fundamentally transformed how organizations build, deploy, and manage applications at scale. It provides an extraordinary level of automation, self-healing capabilities, and declarative infrastructure management that was previously unimaginable. However, this power comes with a complexity that introduces an entirely new category of security challenges. If you have spent years securing traditional servers, virtual machines, or even basic containerized workloads, you will quickly discover that Kubernetes security operates on a different plane altogether. The attack surface is broader, the components are more numerous, the interactions are more dynamic, and the consequences of misconfiguration can be catastrophic. This chapter sets the foundation for everything that follows in this book by explaining precisely why Kubernetes security demands a fundamentally different mindset, a different set of tools, and a different operational discipline than what most teams are accustomed to.

The Shift from Traditional Infrastructure Security

In a traditional infrastructure model, security teams operated with a relatively stable and well-understood environment. You had physical or virtual servers, each with a known IP address, a known operating system, and a known set of running processes. Firewalls sat at network boundaries. Intrusion detection systems monitored traffic. Configuration management tools ensured that servers remained in a desired state. The security perimeter was clearly defined, and the number of entities to protect was manageable and relatively static.

Kubernetes dismantles nearly every one of these assumptions. Instead of long-lived servers with stable identities, you have ephemeral Pods that can be created, destroyed, and rescheduled across different nodes in seconds. Instead of a small number of well-known network endpoints, you have potentially thousands of services communicating with each other through a flat network where, by default, every Pod can reach every other Pod. Instead of a single operating system to harden, you have container images pulled from registries that may or may not be trustworthy, running on a shared kernel with varying levels of isolation.

Consider the following comparison to understand the magnitude of this shift:

Security Aspect	**Traditional Infrastructure**	**Kubernetes Environment**
Compute Identity	Static servers with fixed IPs and hostnames	Ephemeral Pods with dynamic IPs, scheduled across nodes
Network Perimeter	Well-defined firewall boundaries between zones	Flat network by default; all Pods can communicate freely
Access Control	SSH keys, user accounts on individual machines	RBAC policies, ServiceAccounts, API server authentication
Configuration Management	Chef, Puppet, Ansible managing server state	Declarative YAML manifests, Admission Controllers, GitOps
Attack Surface	Operating system, installed packages, open ports	API server, etcd, kubelet, container runtime, images, secrets, RBAC, network policies, and more
Lifecycle	Servers live for months or years	Pods may live for seconds or minutes

Secrets Management	Files on disk, environment variables, vault integrations	Kubernetes Secrets (base64 encoded by default, not encrypted)
Audit Trail	System logs, SSH session recordings	API server audit logs, container stdout/stderr, event streams

This table is not merely academic. Each row represents a domain where security teams must completely rethink their approach. The tools, techniques, and mental models that worked for traditional infrastructure are insufficient for Kubernetes. They are not wrong, but they are incomplete.

Understanding the Kubernetes Attack Surface

To appreciate why Kubernetes security is different, you must first understand the sheer breadth of the Kubernetes attack surface. A Kubernetes cluster is not a single system. It is a distributed system composed of multiple interacting components, each of which can be targeted, misconfigured, or exploited.

The control plane is the brain of the cluster. It consists of the API server, etcd, the scheduler, and the controller manager. The API server is the single point of entry for all cluster operations. Every kubectl command, every controller reconciliation loop, every kubelet heartbeat passes through the API server. If an attacker gains unauthorized access to the API server, they effectively own the entire cluster. They can create Pods, read Secrets, modify RBAC policies, and exfiltrate data. The API server must be protected with strong authentication, robust authorization policies, and comprehensive audit logging.

The etcd datastore is arguably the most sensitive component in the entire cluster. It stores all cluster state, including Secrets, ConfigMaps, RBAC policies, and workload definitions. If etcd is compromised, the attacker has access to everything. In many default installations, etcd communication is not encrypted, and access controls are minimal. This is one of the most critical hardening priorities for any production cluster.

The kubelet runs on every worker node and is responsible for managing Pods on that node. It exposes an API that, if left unsecured, allows an attacker to execute commands inside containers, read logs, and even escalate privileges to the host. Historically, the kubelet API has been a frequent target in real-world Kubernetes attacks.

The container runtime, whether it is containerd, CRI-O, or another implementation, is responsible for actually running containers. Vulnerabilities in the container runtime can allow container escapes, where an attacker breaks out of the container sandbox and gains access to the underlying host operating system.

Beyond these core components, the attack surface extends to:

Component	**Security Risk**	**Example Threat**
API Server	Unauthorized access, privilege escalation	Anonymous authentication enabled, overly permissive RBAC
etcd	Data exfiltration, cluster takeover	Unencrypted etcd communication, no client certificate auth
Kubelet	Remote code execution, container escape	Unauthenticated kubelet API, host path mounts
Container Images	Malware, vulnerabilities, supply chain attacks	Pulling unverified images from public registries
Kubernetes Secrets	Credential theft	Secrets stored as base64 (not encrypted at rest)
Network	Lateral movement, data interception	No NetworkPolicies, unencrypted Pod-to-Pod traffic

RBAC	Privilege escalation	Wildcard permissions, cluster-admin bound to default SA
Admission Controllers	Policy bypass	No Pod Security enforcement, no image validation
Service Accounts	Token theft, API abuse	Auto-mounted SA tokens in Pods that do not need API access
Persistent Volumes	Data leakage	Sensitive data on volumes accessible to multiple Pods

This is not an exhaustive list, but it illustrates a critical point: Kubernetes security is not about securing one thing. It is about securing an entire ecosystem of interacting components, each with its own threat model.

The Default Configuration Problem

One of the most dangerous aspects of Kubernetes security is that the default configuration of most Kubernetes distributions is optimized for ease of use, not for security. This design philosophy makes sense from an adoption perspective. Kubernetes is already complex enough without requiring security expertise to run a simple workload. However, it means that a freshly installed cluster is almost certainly insecure for production use.

Let us examine some of the most significant default behaviors that create security risks:

By default, Kubernetes does not enforce any NetworkPolicies. This means that every Pod in the cluster can communicate with every other Pod, regardless of namespace. An attacker who compromises a single Pod can potentially reach the database, the payment service, the internal API, and any other workload running in the cluster. This flat network model is the Kubernetes equivalent of having no firewall at all.

By default, Kubernetes automatically mounts a ServiceAccount token into every Pod. This token grants the Pod the ability to authenticate to the Kubernetes API server. If the ServiceAccount has been granted excessive permissions through RBAC, a compromised Pod can use this token to query the API, read Secrets, or even create new workloads. Many organizations do not realize that their Pods have this capability until an incident occurs.

By default, Kubernetes Secrets are stored in etcd as base64-encoded values, not encrypted values. Base64 is an encoding scheme, not an encryption scheme. Anyone with read access to etcd, or with the appropriate RBAC permissions to read Secrets, can decode them trivially. Encryption at rest must be explicitly configured.

By default, Pods can run as root, can mount host paths, can use host networking, and can access the host PID namespace. These capabilities are enormously powerful and enormously dangerous. A Pod running as root with a host path mount can read and write any file on the node. A Pod with host networking can sniff traffic from other Pods on the same node. A Pod with access to the host PID namespace can see and potentially interact with processes running on the host.

The following example demonstrates how a seemingly innocent Pod specification can create a severe security risk:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dangerous-pod
  namespace: default
spec:
  hostNetwork: true
  hostPID: true
  containers:
```

```yaml
  - name: attacker
    image: ubuntu:latest
    securityContext:
      privileged: true
    volumeMounts:
    - name: host-root
      mountPath: /host
  volumes:
  - name: host-root
    hostPath:
      path: /
```

This Pod specification requests privileged mode, host networking, host PID namespace access, and mounts the entire host filesystem. If deployed, this Pod effectively has root access to the underlying node. In a default Kubernetes installation, nothing prevents this Pod from being created. There is no admission controller blocking it, no policy engine rejecting it, and no warning alerting the operator.

This is why understanding the defaults is the first step in Kubernetes security. You cannot secure what you do not understand, and you cannot protect against threats you do not know exist.

Note: Starting with Kubernetes 1.25, the Pod Security Admission controller replaced the deprecated PodSecurityPolicy. This built-in admission controller can enforce security standards at the namespace level, but it must be explicitly configured. It does not block dangerous Pods by default in all configurations.

The Shared Responsibility Model

Kubernetes security is not solely the responsibility of any single team or any single tool. It follows a shared responsibility model that spans multiple layers and multiple stakeholders.

If you are using a managed Kubernetes service such as Amazon EKS, Google GKE, or Azure AKS, the cloud provider is responsible for securing the control plane components. They manage the API server, etcd, the scheduler, and the controller manager. They handle patching, high availability, and network security for these components. However, everything else is your responsibility. You are responsible for securing your workloads, your container images, your RBAC policies, your network policies, your secrets management, and your node configuration.

If you are running a self-managed Kubernetes cluster using tools like kubeadm, kops, or Kubespray, you are responsible for everything. You must secure the control plane, the worker nodes, the network, the storage, and the workloads. This is a significantly larger burden and requires deep expertise.

The following table clarifies this division:

Responsibility	**Managed Kubernetes (EKS, GKE, AKS)**	**Self-Managed Kubernetes**
Control Plane Security	Cloud Provider	Your Team
etcd Encryption and Backup	Cloud Provider	Your Team
API Server Authentication	Shared (provider configures, you manage identities)	Your Team
Worker Node OS Patching	Your Team	Your Team
Container Image Security	Your Team	Your Team
RBAC Configuration	Your Team	Your Team
Network Policies	Your Team	Your Team
Secrets Management	Your Team	Your Team
Pod Security Standards	Your Team	Your Team

Audit Logging Configuration	Shared (provider provides, you configure and analyze)	Your Team
Ingress and Egress Security	Your Team	Your Team

Regardless of the deployment model, the majority of security responsibilities fall on the team operating the cluster. This is a sobering reality that many organizations underestimate.

Thinking in Layers: Defense in Depth for Kubernetes

Effective Kubernetes security requires a defense-in-depth strategy. No single control is sufficient. You must implement security measures at every layer of the stack, so that if one layer is breached, the next layer provides protection.

The layers of Kubernetes security can be conceptualized as follows:

The Cluster Layer encompasses the security of the control plane and the infrastructure on which the cluster runs. This includes securing the API server with strong authentication and authorization, encrypting etcd at rest and in transit, restricting access to the kubelet API, and ensuring that the underlying nodes are hardened and patched.

The Namespace Layer provides logical isolation within the cluster. Namespaces should be used to separate workloads by team, environment, or sensitivity level. RBAC policies should be scoped to namespaces to limit what users and ServiceAccounts can do. Resource quotas and limit ranges should be applied to prevent resource exhaustion.

The Network Layer controls communication between Pods, between namespaces, and between the cluster and the outside world. NetworkPolicies should be used to implement a default-deny posture, where Pods can only communicate with explicitly allowed destinations. Service meshes can provide mutual TLS for encrypting Pod-to-Pod traffic.

The Workload Layer addresses the security of the Pods themselves. This includes enforcing Pod Security Standards to prevent privileged containers, scanning container images for vulnerabilities, using read-only root filesystems, dropping unnecessary Linux capabilities, and running containers as non-root users.

The Application Layer is the security of the code running inside the containers. This includes secure coding practices, dependency management, input validation, and proper handling of credentials and sensitive data.

Each layer reinforces the others. A vulnerability at one layer may be mitigated by controls at another layer. This is the essence of defense in depth, and it is particularly important in Kubernetes because the system is so dynamic and complex.

Practical Exercise: Assessing Your Cluster's Default Security Posture

To ground these concepts in practice, perform the following exercise on a test cluster. Do not perform this on a production cluster without proper authorization.

First, check whether anonymous authentication is enabled on the API server:

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

If this command returns a list of permissions rather than an error, anonymous authentication is enabled and may allow unauthorized access.

Next, check whether any ClusterRoleBindings grant excessive permissions:

```
kubectl get clusterrolebindings -o json | jq '.items[] |
select(.roleRef.name == "cluster-admin") | .metadata.name'
```

This command lists all ClusterRoleBindings that reference the cluster-admin role. Each of these bindings grants full administrative access to the cluster. Review each one carefully and remove any that are unnecessary.

Check whether Pods in the default namespace have ServiceAccount tokens automatically mounted:

```
kubectl get pods -n default -o json | jq '.items[] |
{name: .metadata.name,
automountServiceAccountToken: .spec.automountServiceAccountToken}
'
```

If the automountServiceAccountToken field is null or true, the Pod has a Kubernetes API token mounted that it may not need.

Finally, check whether any NetworkPolicies exist in your namespaces:

```
kubectl get networkpolicies --all-namespaces
```

If this command returns no results, your cluster has no network segmentation whatsoever. Every Pod can communicate with every other Pod.

Note: These checks represent only the beginning of a comprehensive security assessment. Tools such as kube-bench (which checks against the CIS Kubernetes Benchmark), kubeaudit, and Trivy can automate much of this assessment process. We will explore these tools in detail in later chapters.

Setting the Stage for What Follows

This chapter has established a critical foundation. Kubernetes security is different because the platform itself is different. It is more dynamic, more distributed, more complex, and more powerful than traditional infrastructure. The default configurations are permissive. The attack surface is vast. The shared responsibility model places the majority of the burden on the team operating the cluster. And effective security requires a layered, defense-in-depth approach that addresses every component from the control plane to the application code.

In the chapters that follow, we will systematically address each layer of this security model. We will begin with hardening the control plane and securing cluster infrastructure. We will then move to RBAC design, network policy implementation, secrets management, image security, runtime protection, audit logging, and governance frameworks. Each chapter will build on the concepts introduced here, providing practical, actionable guidance for securing Kubernetes clusters in production.

The journey toward a secure Kubernetes environment is not a single action but a continuous process. It requires vigilance, discipline, and a willingness to understand the platform at a deep level. This book is designed to guide you through that process, one layer at a time.

Chapter 2: Threat Modeling Kubernetes Clusters

Understanding the security posture of a Kubernetes cluster begins long before the first pod is deployed. It starts with a disciplined, methodical process of identifying what could go wrong, who might exploit weaknesses, and where the most critical vulnerabilities reside. This process is known as threat modeling, and when applied to Kubernetes, it becomes one of the most powerful exercises a platform team can undertake. Kubernetes, by its very nature, is a complex distributed system with a vast attack surface. The API server, etcd, kubelet, container runtimes, networking layers, and the supply chain of container images all present distinct categories of risk. In this chapter, we will walk through the full practice of threat modeling a Kubernetes cluster, from understanding the attack surface to building a structured threat model that can guide hardening decisions for months and years to come.

Why Threat Modeling Matters for Kubernetes

Every organization running Kubernetes in production faces a fundamental question: what are we protecting, and from whom? Without answering this question explicitly, security efforts become reactive and fragmented. Teams patch vulnerabilities as they appear, apply security policies inconsistently, and fail to prioritize the controls that would have the greatest impact. Threat modeling reverses this dynamic. It forces teams to think like an attacker, to trace the paths an adversary might follow from initial access to full cluster compromise, and to place controls at the points where they matter most.

Kubernetes is particularly well suited to threat modeling because its architecture is well documented and its components interact through clearly defined interfaces. The Kubernetes API server, for example, is the central control plane component through which nearly all operations flow. An attacker who gains access to the API server with sufficient privileges can create pods, read secrets, modify RBAC policies, and effectively own the entire cluster. Understanding this single fact shapes dozens of security decisions, from how authentication is configured to how network policies restrict access to the API server endpoint.

Threat modeling is not a one-time activity. As clusters evolve, as new workloads are deployed, and as the broader threat landscape shifts, the model must be revisited and updated. The goal is not perfection but rather a living document that captures the team's best understanding of risk at any given point in time.

Understanding the Kubernetes Attack Surface

Before building a threat model, it is essential to have a thorough understanding of the Kubernetes attack surface. The attack surface is the sum of all points where an unauthorized user or process could attempt to interact with the system. In Kubernetes, this surface is broad and multi-layered.

The following table describes the primary components of a Kubernetes cluster and the security concerns associated with each.

Component	Description	Security Concerns
API Server	The central management endpoint for all cluster operations. All kubectl commands, controller actions, and scheduler decisions pass through the API server.	Unauthorized access, privilege escalation through RBAC misconfiguration, unauthenticated endpoints, token theft
etcd	The distributed key-value store that holds all cluster state, including secrets, configuration, and RBAC policies.	Direct access to etcd bypasses all Kubernetes authorization. Data is stored in base64 encoding by default, not encrypted.
Kubelet	The agent running on each node that manages pod lifecycle and communicates with the API server.	The kubelet API can be exploited if anonymous authentication is enabled. Node-level compromise gives access to all pods on that node.
Container Runtime	The software responsible for running containers, such as containerd or CRI-O.	Container escape vulnerabilities allow attackers to break out of the container and access the host operating system.
Networking (CNI)	The Container Network Interface plugin that provides pod-to-pod communication.	By default, all pods can communicate with all other pods. Lack of network policies allows lateral movement.
Scheduler	The component that assigns pods to nodes based on resource requirements and constraints.	Manipulating scheduling decisions can place malicious pods on sensitive nodes.
Controller Manager	Runs controllers that regulate the state of the cluster, such as the ReplicaSet and Namespace controllers.	Compromise of the controller manager allows manipulation of cluster state at a fundamental level.

Cloud Provider Integration	Integrations with cloud APIs for load balancers, storage, and identity.	Overly permissive cloud IAM roles attached to nodes or pods can allow cloud account compromise.
Container Images	The software artifacts deployed as containers within the cluster.	Vulnerable or malicious base images, unpatched dependencies, embedded secrets in image layers.
Secrets Management	Kubernetes native secrets stored in etcd and mounted into pods.	Secrets are base64 encoded, not encrypted by default. Any pod with the correct RBAC or service account can read them.

This table is not exhaustive, but it captures the most critical areas where threats emerge. Each component represents a potential entry point or escalation path for an attacker.

Identifying Threat Actors and Their Motivations

A threat model is incomplete without a clear understanding of who the adversaries are. In the context of Kubernetes, threat actors can be categorized into several groups, each with different capabilities, motivations, and levels of access.

External Attackers are individuals or groups with no initial access to the cluster. Their goal is typically to gain a foothold through exposed services, vulnerable applications, or misconfigured ingress points. An external attacker might scan for publicly exposed Kubernetes dashboards, unprotected API servers, or applications with known vulnerabilities running inside the cluster.

Malicious Insiders are individuals who already have some level of legitimate access to the cluster. This could be a developer with namespace-scoped permissions who attempts to escalate privileges, or an operations engineer who abuses their broad access for unauthorized purposes. Insider threats are particularly dangerous in Kubernetes because the RBAC system, while powerful, is frequently misconfigured to grant overly broad permissions.

Compromised Workloads represent a scenario where a legitimate application running in the cluster is exploited by an external attacker. Once inside the pod, the attacker can attempt to access the Kubernetes API using the pod's service account token, read mounted secrets, communicate with other pods, or attempt a container escape to reach the underlying node.

Supply Chain Attackers target the software supply chain rather than the cluster directly. They might inject malicious code into a base image, compromise a CI/CD pipeline, or tamper with Helm charts or Kubernetes manifests before they reach the cluster.

Understanding these threat actors helps prioritize controls. For example, if the primary concern is compromised workloads, then runtime security, network policies, and restrictive pod security standards become the highest priority. If supply chain attacks are the primary concern, then image signing, admission control, and pipeline security take precedence.

Applying the STRIDE Framework to Kubernetes

STRIDE is a well-established threat modeling framework developed by Microsoft that categorizes threats into six types: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Applying STRIDE to Kubernetes provides a structured way to identify threats across the entire cluster.

STRIDE Category	**Kubernetes Example**	**Mitigation Strategy**
Spoofing	An attacker uses a stolen service account token to authenticate to the API server as a legitimate workload.	Enable short-lived, projected service account tokens. Rotate tokens regularly. Use OIDC for human authentication.
Tampering	An attacker modifies a ConfigMap or Secret to change application behavior or inject malicious configuration.	Use RBAC to restrict write access. Enable admission controllers to validate changes. Use GitOps to detect drift.
Repudiation	A user deletes critical resources and there is no audit trail to identify who performed the action.	Enable Kubernetes audit logging. Forward audit logs to a centralized, immutable logging system.
Information Disclosure	Secrets mounted into pods are readable by any process in the container. Environment variables expose sensitive data in process listings.	Use volume-mounted secrets instead of environment variables. Enable etcd encryption at rest. Use external secret managers.
Denial of Service	A pod without resource limits consumes all CPU and memory on a node, causing other pods to be evicted.	Enforce resource requests and limits through LimitRange and ResourceQuota objects. Use Pod Priority and Preemption.
Elevation of Privilege	A pod running as root with a privileged security context escapes the container and gains access to the host.	Enforce Pod Security Standards. Use seccomp and AppArmor profiles. Disable privileged containers through admission control.

This framework provides a repeatable, comprehensive approach to identifying threats. Each category prompts specific questions about the cluster's configuration and the controls in place.

Building a Kubernetes Threat Model Step by Step

Building a threat model for a Kubernetes cluster is a collaborative exercise that should involve platform engineers, security teams, application developers, and operations staff. The following process provides a practical, repeatable approach.

Step 1: Define the Scope. Begin by defining what is being modeled. This might be the entire cluster, a specific namespace, or a particular workload. For a first threat model, it is often most productive to focus on the control plane and a single representative workload.

Step 2: Create a Data Flow Diagram. Map how data flows through the system. In Kubernetes, this includes the flow of API requests from users and controllers to the API server, the flow of secrets from etcd to pods, the flow of container images from registries to nodes, and the flow of network traffic between pods. A clear data flow diagram reveals the trust boundaries in the system, which are the points where data crosses from one security domain to another.

Consider the following simplified data flow for a typical Kubernetes deployment:

```
Developer --> kubectl --> API Server --> etcd
                                    --> Scheduler --> Kubelet -->
Container Runtime --> Pod
                                    --> Controller Manager

Pod --> Kubernetes API (via Service Account Token)
Pod --> Other Pods (via CNI Network)
Pod --> External Services (via Egress)

CI/CD Pipeline --> Container Registry --> Kubelet (Image Pull)
```

Each arrow in this diagram represents a potential attack vector. The connection between the developer and the API server must be authenticated and encrypted. The connection between the pod and the Kubernetes API must be restricted by RBAC. The connection between the CI/CD pipeline and the container registry must be protected against tampering.

Step 3: Identify Threats. Using the STRIDE framework and the data flow diagram, systematically identify threats at each trust boundary. For each arrow in the diagram, ask: Can this communication be spoofed? Can the data be tampered with? Is there an audit trail? Could sensitive data be disclosed? Could this be used for denial of service? Could this lead to privilege escalation?

Step 4: Assess Risk. Not all threats are equally likely or equally impactful. Assess each threat based on its likelihood and potential impact. A useful approach is to use a simple risk matrix:

Likelihood / Impact	Low Impact	Medium Impact	High Impact
High Likelihood	Medium Risk	High Risk	Critical Risk
Medium Likelihood	Low Risk	Medium Risk	High Risk
Low Likelihood	Low Risk	Low Risk	Medium Risk

For example, a container escape vulnerability in an unpatched runtime has high impact (full node compromise) and medium likelihood (requires specific conditions), making it a high risk item. An unauthenticated kubelet API, on the other hand, has high impact and high likelihood if exposed, making it a critical risk.

Step 5: Define Mitigations. For each identified threat, define specific, actionable mitigations. These should be tied to Kubernetes configuration, policy, or architecture decisions. For example:

```yaml
# Example: Enforcing a restricted Pod Security Standard at the
namespace level
apiVersion: v1
kind: Namespace
metadata:
  name: production
```