

Kubernetes Security A Visual Guide

Essential Fundamentals for Secure Clusters and Workloads

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with Kubernetes security, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

1. **Read the chapters in order.** The sequence builds the required background step by step: first the security model and trust boundaries, then identity and policy, then workload and data protection, and finally secure operations and recovery.
2. **Review “What You’ll Be Able to Do” before each chapter.** Use the learning goals to understand which capabilities the chapter is intended to build and which details deserve the most attention.
3. **Use diagrams to understand relationships, processes, and structures.** Read the nearby explanation first, then follow the visual path and confirm that you can explain every boundary, stage, and arrow in your own words.
4. **Try the code and hands-on examples whenever possible.** Use a safe learning cluster or approved non-production environment, confirm the stated prerequisites, and compare the actual result with the expected result described in the text.
5. **Use each Chapter Summary and Review Questions block to check retention.** Answer the questions without looking back first, then revisit the relevant section when an explanation is incomplete or uncertain.
6. **Treat advanced-topic overviews as signposts rather than memorization assignments.** At this stage, understand why the topic exists, where it fits, and when you may need it; return for deeper study when a project requires it.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Main Text](#)

[Part I — Security Mental Models and Trust Boundaries](#)

[Chapter 1 — Kubernetes Security Is Layered Risk Reduction](#)

[Start with what must be protected](#)

[Security protects three outcomes](#)

[Defense in depth assumes controls can fail](#)

[Least privilege limits blast radius](#)

[Design principles that reinforce one another](#)

[The four practical security layers](#)

[Managed Kubernetes still uses shared responsibility](#)

[A small threat model is better than an unexamined checklist](#)

[Example: a stolen workload token](#)

[High-frequency threats to recognize early](#)

[Misconceptions that lead to weak designs](#)

[A working security model](#)

[Chapter 2 — Read the Cluster as a Set of Trust Boundaries](#)

[Control plane and data plane](#)

[The API server is the main control entrance](#)

[etcd holds the cluster's persistent truth](#)

[Controllers are trusted automation](#)

[The scheduler controls placement, not authorization](#)

[Nodes concentrate workload risk](#)

[The kubelet is more than a process launcher](#)

[The runtime and host kernel enforce container boundaries](#)

[CNI plugins sit on critical network paths](#)

[CSI drivers control persistent-data access](#)

[Add-ons can become hidden cluster administrators](#)

[Namespaced and cluster-scoped resources](#)

[Four paths reveal most security relationships](#)

[1. The API request path](#)

[2. The workload creation path](#)

[3. The application network path](#)

[4. The data and Secret path](#)

[Walkthrough: from `kubectl apply` to a running container](#)

[Inventory privileged components before trusting them](#)

[Self-managed and managed responsibility boundaries](#)

[Misconceptions that hide trust relationships](#)

[The cluster as connected authority](#)

Main Text

Kubernetes security is the practice of reducing both the likelihood of compromise and the damage that follows when a control fails. It is not one product, one configuration switch, or one compliance checklist. A dependable design combines identity, authorization, admission policy, workload isolation, network restrictions, data protection, and operational evidence so that no single mistake decides the security of the whole cluster.

Chapter 1 — Kubernetes Security Is Layered Risk Reduction

The central conclusion is simple: secure Kubernetes by building several independent barriers around important assets. Credentials will sometimes leak, software will contain vulnerabilities, and people will occasionally make mistakes. A sound design assumes those failures are possible and prevents any one of them from becoming a cluster-wide incident.

What You'll Be Able to Do

After completing this chapter, you should be able to:

- Identify the assets, trust relationships, and failure consequences that a Kubernetes security review must consider.
- Explain defense in depth, least privilege, blast radius, and the confidentiality–integrity–availability model.
- Build a small threat model for a workload without turning the exercise into an unfocused checklist.

Start with what must be protected

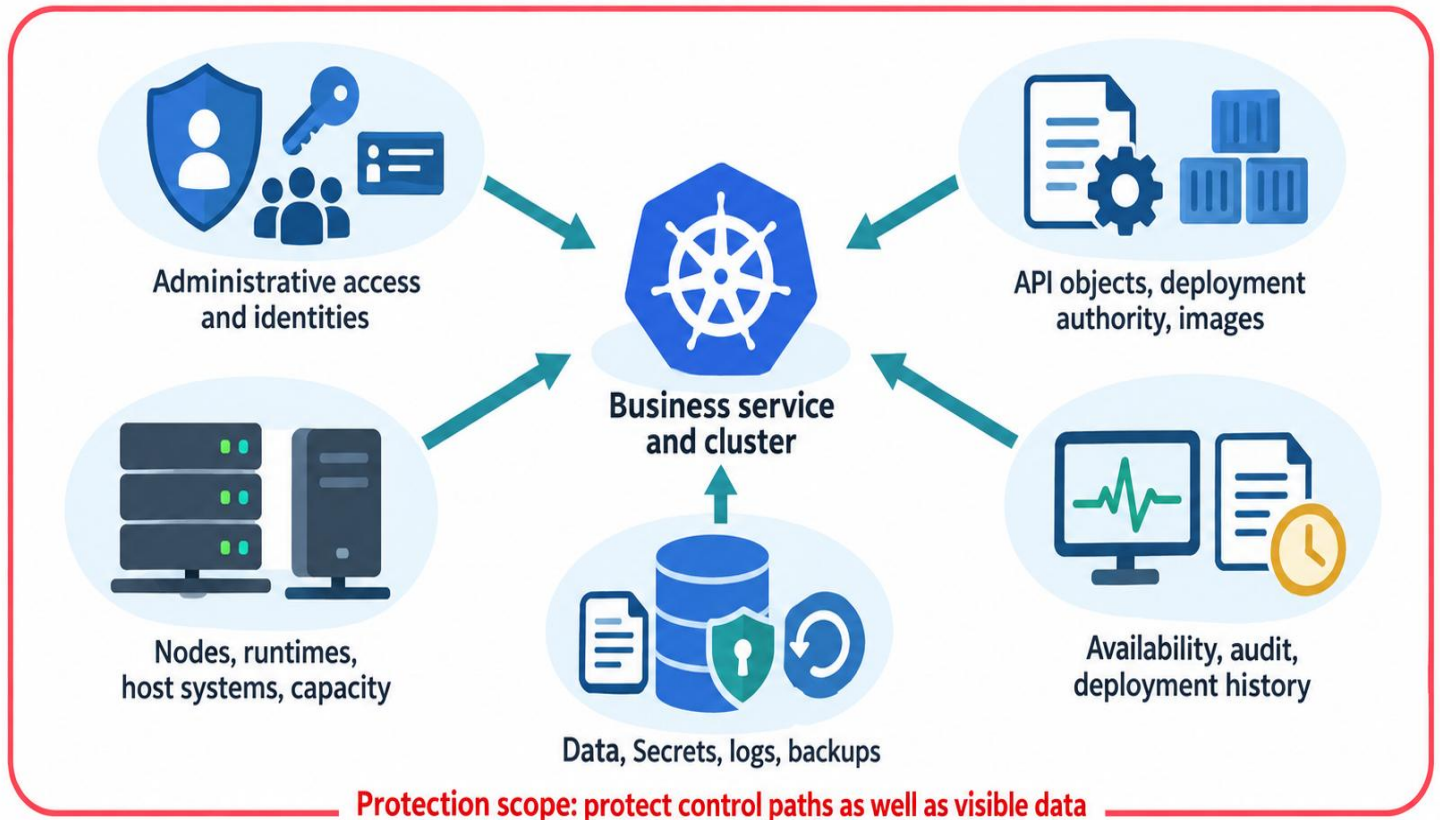
An **asset** is anything whose loss, disclosure, or unauthorized change would harm the organization. In Kubernetes, the most important assets are not limited to application data. They include:

- Administrative access to the cluster.
- Human and workload credentials.
- Kubernetes API objects, including Deployments, Roles, and Secrets.
- Application data in memory, storage, logs, and backups.
- Container images and the systems that build them.
- Nodes, container runtimes, and host operating systems.
- Service availability and the capacity needed to run critical workloads.
- Security evidence such as audit records and deployment history.

A useful first question is therefore not “Which security tool should we install?” It is “**Which assets matter, who should be able to reach them, and what happens if that access is abused?**”

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-1 Start with what must be protected



001

Figure 1-1: Start with what must be protected

Consider an internal application with a frontend, an API, and a database. The database password is clearly sensitive, but so are the following items:

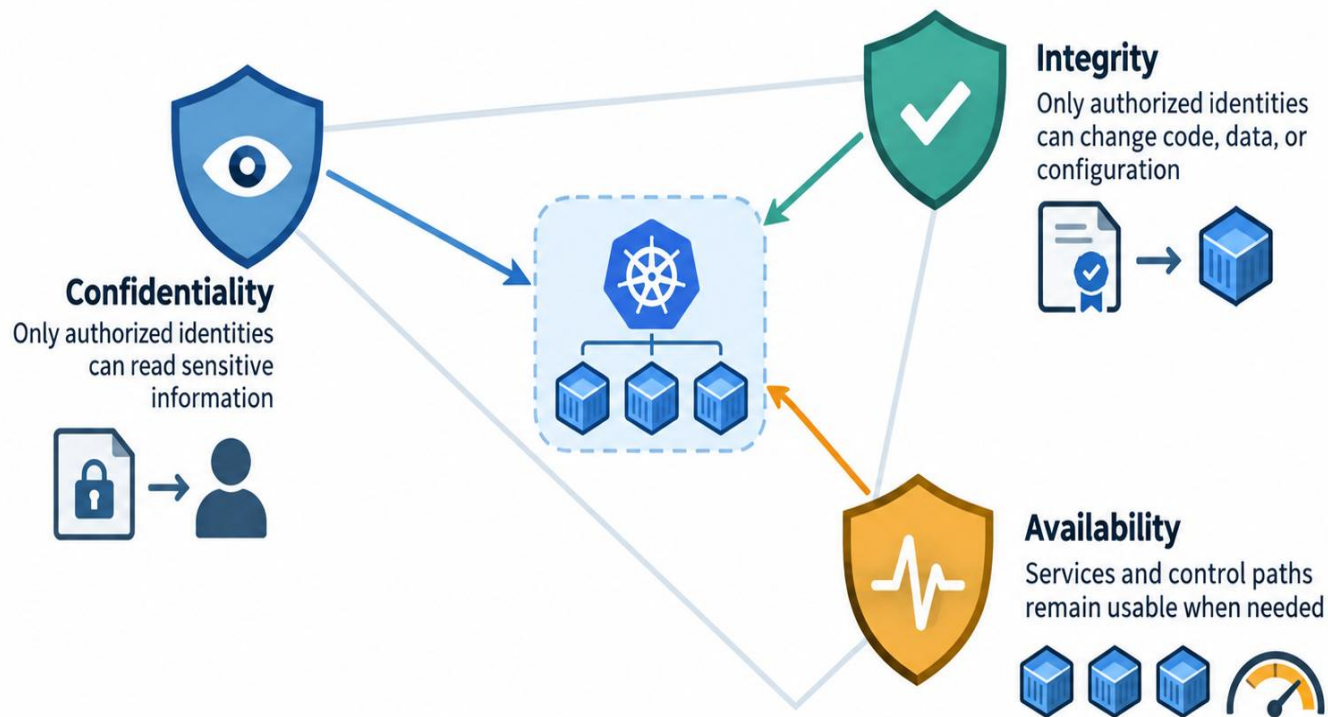
- The API's ServiceAccount token, because it may authorize Kubernetes API requests.
- The Deployment specification, because changing the image could run an attacker's code.
- The image registry credentials, because they may permit image replacement.
- The node that hosts the database Pod, because node-level control can bypass many Pod-level restrictions.
- The backup, because it may contain the same data as the live database.

This broader view prevents a common failure: protecting the visible application endpoint while leaving the control paths around it weak.

Security protects three outcomes

The traditional security goals of **confidentiality**, **integrity**, and **availability** are directly useful in Kubernetes.

Chapter 1: Kubernetes Security Is Layered Risk Reduction
1-2 Security protects three outcomes



Security requires all three outcomes—not confidentiality alone

002

Figure 1-2: Security protects three outcomes

Goal	Meaning	Kubernetes example
Confidentiality	Information is visible only to authorized identities.	A support engineer can read Pod logs but cannot read database Secrets.
Integrity	Data and configuration cannot be changed without authorization.	Only the deployment pipeline can change the production image reference.
Availability	Systems remain usable when needed.	One workload cannot exhaust all node memory or block every API request.

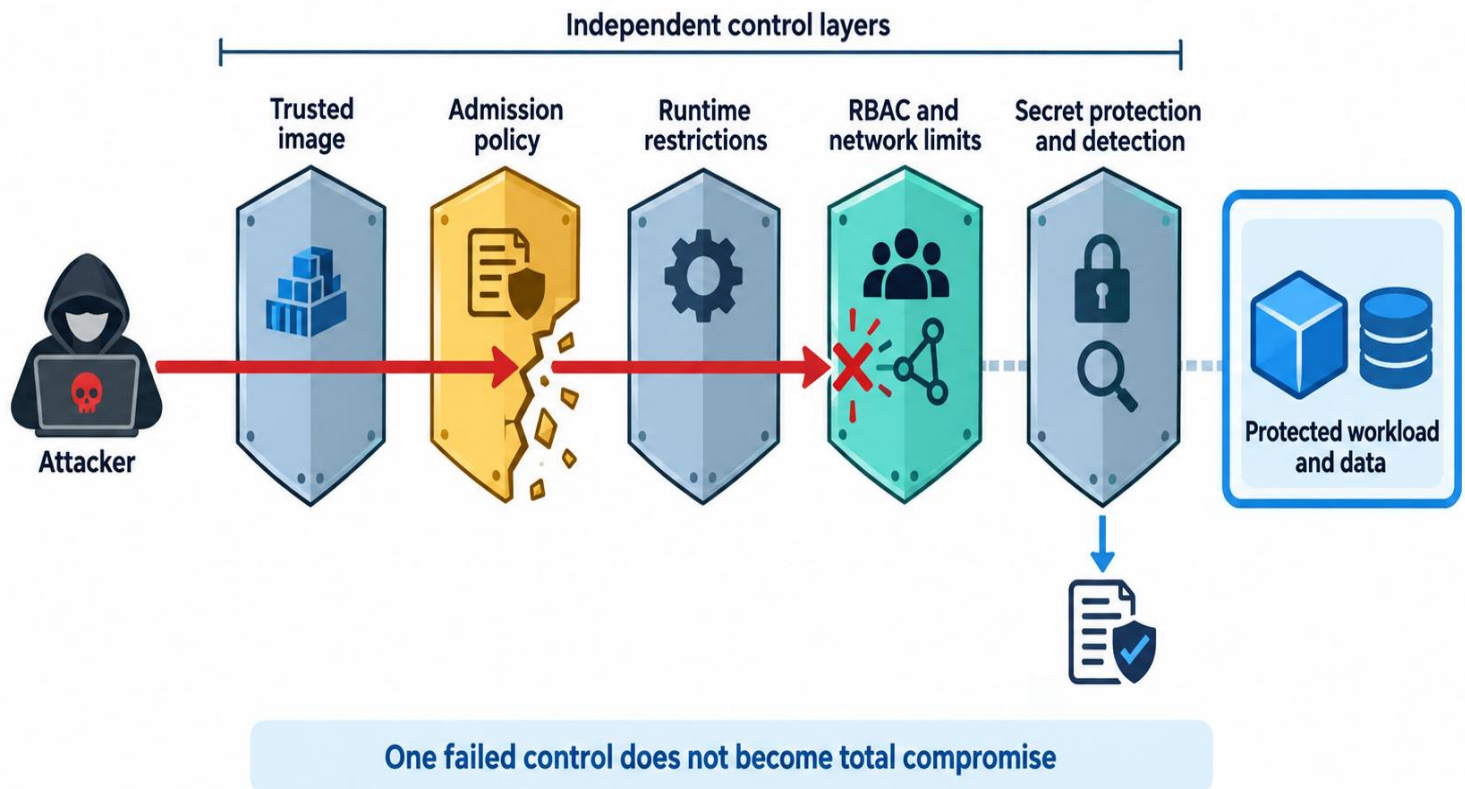
These goals can conflict. A policy engine that rejects unsafe Pods improves integrity, but if every API request depends on an unavailable external webhook, that same control can damage availability. Security engineering therefore requires explicit tradeoffs rather than a search for the strictest possible setting.

Defense in depth assumes controls can fail

Defense in depth means placing multiple security controls in sequence so that another control still limits damage when one fails. The idea is similar to protecting a building with exterior doors, room locks, access badges, cameras, and a safe. None is perfect, but an intruder must cross several independent barriers.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-3 Defense in depth assumes controls can fail



003

Figure 1-3: Defense in depth assumes controls can fail

For a Kubernetes workload, the layers might include:

1. The image is built from reviewed source and stored in a controlled registry.
2. Admission policy rejects an image or Pod configuration that violates the cluster baseline.
3. The Pod runs as a non-root user with unnecessary Linux capabilities removed.
4. RBAC gives its ServiceAccount only the API permissions it needs.
5. NetworkPolicy limits which peers the Pod can contact.
6. Secrets are delivered only to the container that needs them.
7. Audit and runtime signals make suspicious activity visible.

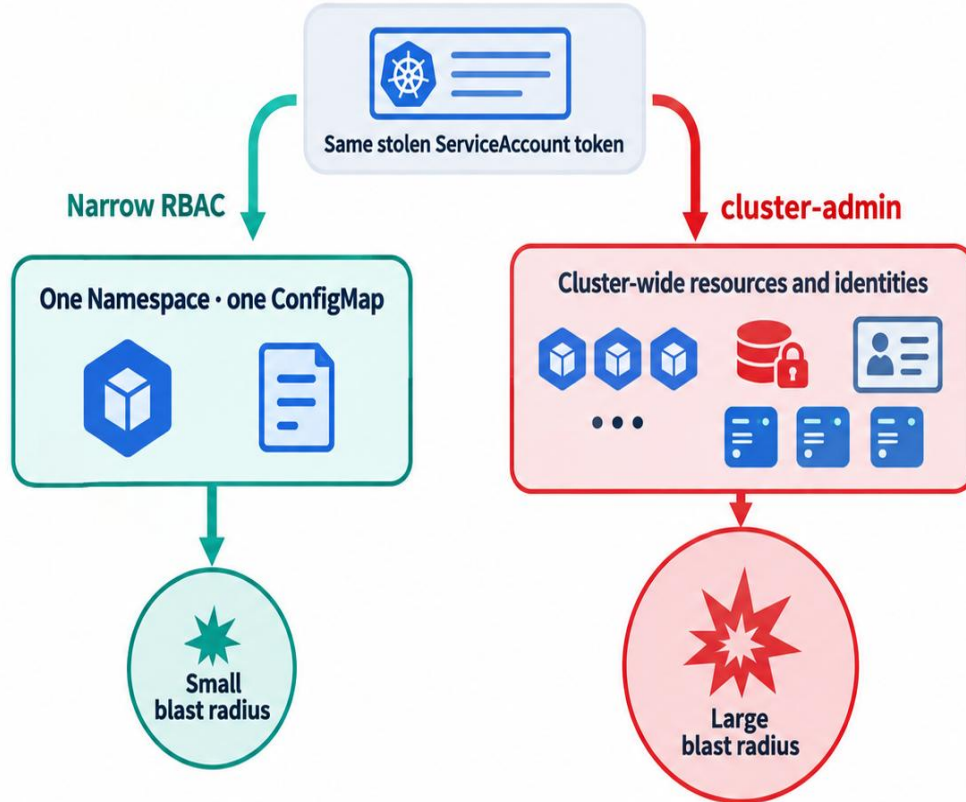
The value of the design is not that compromise becomes impossible. Its value is that **an attacker who defeats one layer still encounters other barriers, and defenders have more opportunities to detect and contain the incident.**

Least privilege limits blast radius

Least privilege means giving a person, workload, or component only the permissions and access needed for its current task. The security benefit is best understood through **blast radius**, which is the maximum damage a failure or compromise can cause.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-4 Least privilege limits blast radius



Least privilege does not improve function; it limits damage

004

Figure 1-4: Least privilege limits blast radius

Suppose an application only needs to read one ConfigMap in its own Namespace. Giving its ServiceAccount `cluster-admin` permission does not make the application work better. It changes the consequence of a stolen token from “one configuration value may be exposed” to “the entire cluster may be controlled.”

Least privilege applies to more than RBAC:

- A Pod should receive only the Linux capabilities it needs.
- A workload should reach only required network destinations.
- A container should mount only required Secrets and volumes.
- A human should administer only the relevant environments and Namespaces.
- A CI/CD system should deploy only the workloads for which it is responsible.
- A controller should watch and modify only the resources in its intended scope.

The practical test is: **If this identity or component were compromised today, what could it read, change, create, delete, impersonate, or disrupt?**

Design principles that reinforce one another

Several principles repeatedly appear throughout Kubernetes security:

- **Secure defaults:** New workloads begin with restrictive settings rather than depending on every author to remember them.
- **Explicit trust:** Access is granted because a documented relationship requires it, not merely because two workloads share a cluster.
- **Short-lived credentials:** A stolen credential becomes useless quickly and is easier to rotate than a certificate or token intended to last for years.
- **Separation of duties:** No single routine identity can build, approve, deploy, and conceal a high-risk change.
- **Known-good rebuilding:** When a node or image loses trust, replace it from a controlled source instead of trying to clean it in place.
- **Verifiable controls:** Permissions and policies are tested, and security-relevant actions produce evidence.

These principles are stronger together. For example, a short-lived token has limited value if the identity behind it is still a cluster administrator. Narrow RBAC has limited value if the same workload can mount the host filesystem and steal node credentials. Security decisions must therefore be evaluated as a system.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-5 Design principles that reinforce one another



005

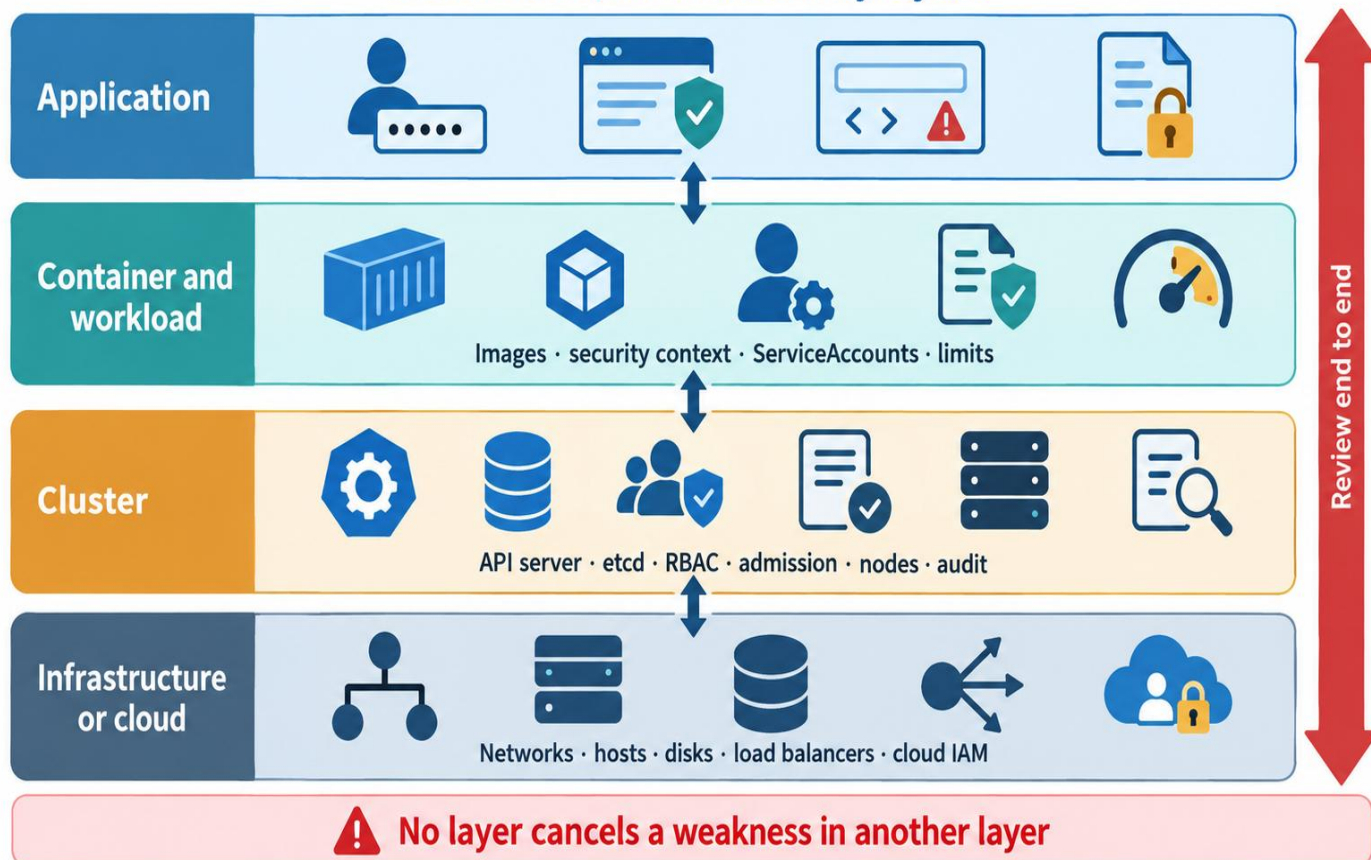
Figure 1-5: Design principles that reinforce one another

The four practical security layers

A cloud-native system is often divided into four connected layers. The exact ownership depends on the environment, but the model helps locate responsibility.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-6 The four practical security layers



006

Figure 1-6: The four practical security layers

Layer	Typical concerns	Examples of controls
Infrastructure or cloud	Physical systems, virtual networks, host identity, disks, load balancers, managed services	Network segmentation, host patching, disk encryption, cloud IAM
Cluster	API server, etcd, nodes, add-ons, authorization, admission	TLS, RBAC, admission policy, node hardening, audit logging
Container and workload	Image contents, process privileges, Pod configuration, workload identity	Image review, security context, ServiceAccounts, resource limits
Application	User authentication, input handling, business authorization, application secrets	Secure coding, application access control, dependency maintenance

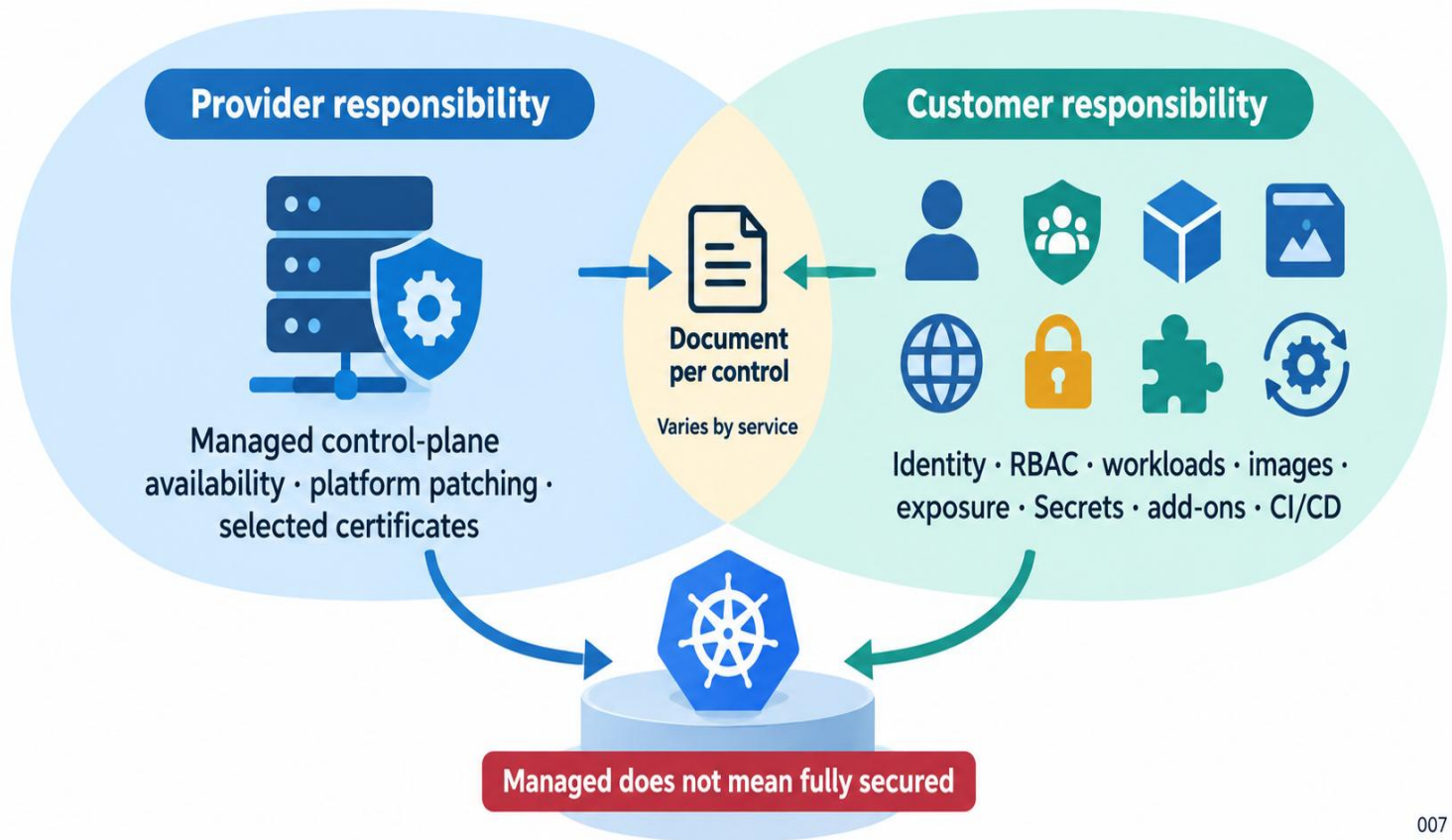
A strong control at one layer does not erase weaknesses in another. Kubernetes RBAC cannot correct an SQL injection flaw. Application authentication cannot protect a node whose runtime socket is exposed to an untrusted Pod.

Managed Kubernetes still uses shared responsibility

A **managed Kubernetes service** is a service in which a provider operates some cluster components, commonly parts of the control plane. This can remove difficult operational work, but it does not transfer every security decision to the provider.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-7 Managed Kubernetes still uses shared responsibility



007

Figure 1-7: Managed Kubernetes still uses shared responsibility

Depending on the service, the provider may manage control-plane availability, patching, or certificate rotation. The cluster owner usually still controls many high-impact areas, including:

- Human access and identity-provider integration.
- RBAC bindings and workload ServiceAccounts.
- Pod specifications and admission policies.
- Public Services, ingress routes, and many network rules.
- Container images and application dependencies.
- Secrets, data access, and backup handling.
- Add-ons, operators, and CI/CD identities installed by the organization.

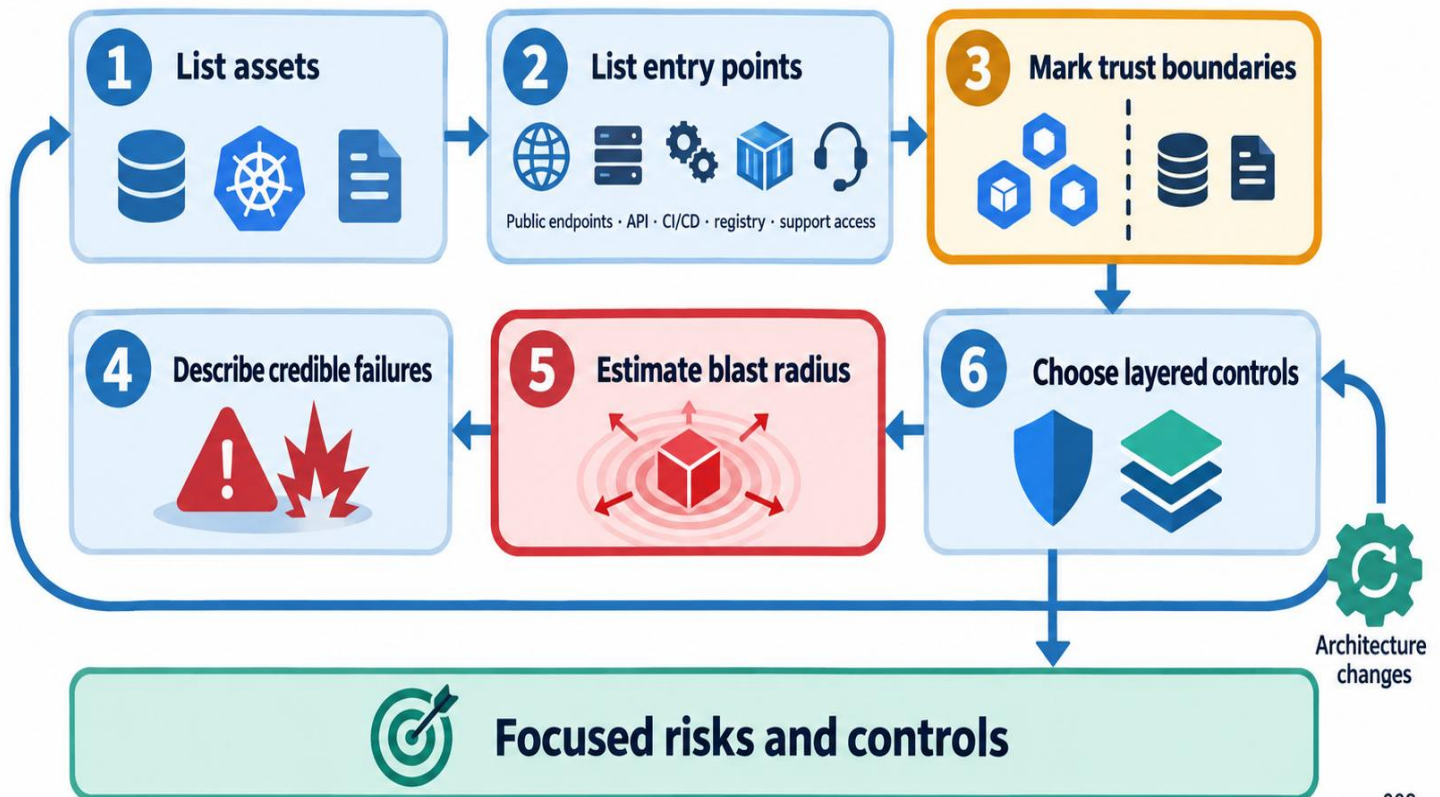
The precise boundary is environment-specific. **The safe approach is to document responsibility control by control rather than assuming that “managed” means “secured.”**

A small threat model is better than an unexamined checklist

A **threat model** is a structured description of what must be protected, how it can be reached, where trust changes, and how failure could spread. It does not need to be a large document. For a new workload, a one-page model can be enough to reveal the most important risks.

Chapter 1: Kubernetes Security Is Layered Risk Reduction

1-8 A small threat model is better than an unexamined checklist



008

Figure 1-8: A small threat model is better than an unexamined checklist

Use this sequence:

1. **List assets.** Include data, credentials, identities, deployment authority, and availability.
2. **List entry points.** Include public endpoints, the Kubernetes API, CI/CD, image registries, support access, and third-party integrations.
3. **Mark trust boundaries.** A trust boundary is a point where data or control passes between parties with different privileges or levels of trust.
4. **Describe likely failures.** Focus on realistic events such as a stolen kubeconfig, vulnerable image, excessive RoleBinding, exposed Service, or compromised controller.
5. **Estimate blast radius.** Record what each failure could read, change, or interrupt.
6. **Choose layered controls.** Prefer controls that prevent the event, limit its impact, and make it detectable.

For the internal three-tier application, a compact model might look like this:

Asset or path	Credible failure	Potential impact	Important controls
Developer access	Kubeconfig copied from a workstation	Unauthorized workload changes	Short-lived login, narrow group RBAC, context checks, audit
API ServiceAccount	Token disclosed in an error report	Kubernetes API access as the application	Dedicated ServiceAccount, narrow RBAC, short-lived projected token

Asset or path	Credible failure	Potential impact	Important controls
Frontend-to-API path	Frontend container compromised	Lateral movement to database or control endpoints	NetworkPolicy, separate identities, application authorization
Deployment image	Registry tag replaced	Attacker code runs in the cluster	Immutable image reference, admission verification, reviewed pipeline
Database credential	Secret read by unrelated workload	Data disclosure or modification	Secret-specific RBAC, restricted Pod creation, narrow mounts, rotation

This model should change when the architecture changes. Adding an operator, service mesh, storage driver, or GitOps controller creates new identities and trust paths that deserve review.

Example: a stolen workload token

Assume an attacker obtains the ServiceAccount token used by the API component. No single control can make the disclosure harmless, but several controls can sharply reduce the outcome.

Chapter 1: Kubernetes Security Is Layered Risk Reduction
1-9 Example: a stolen workload token

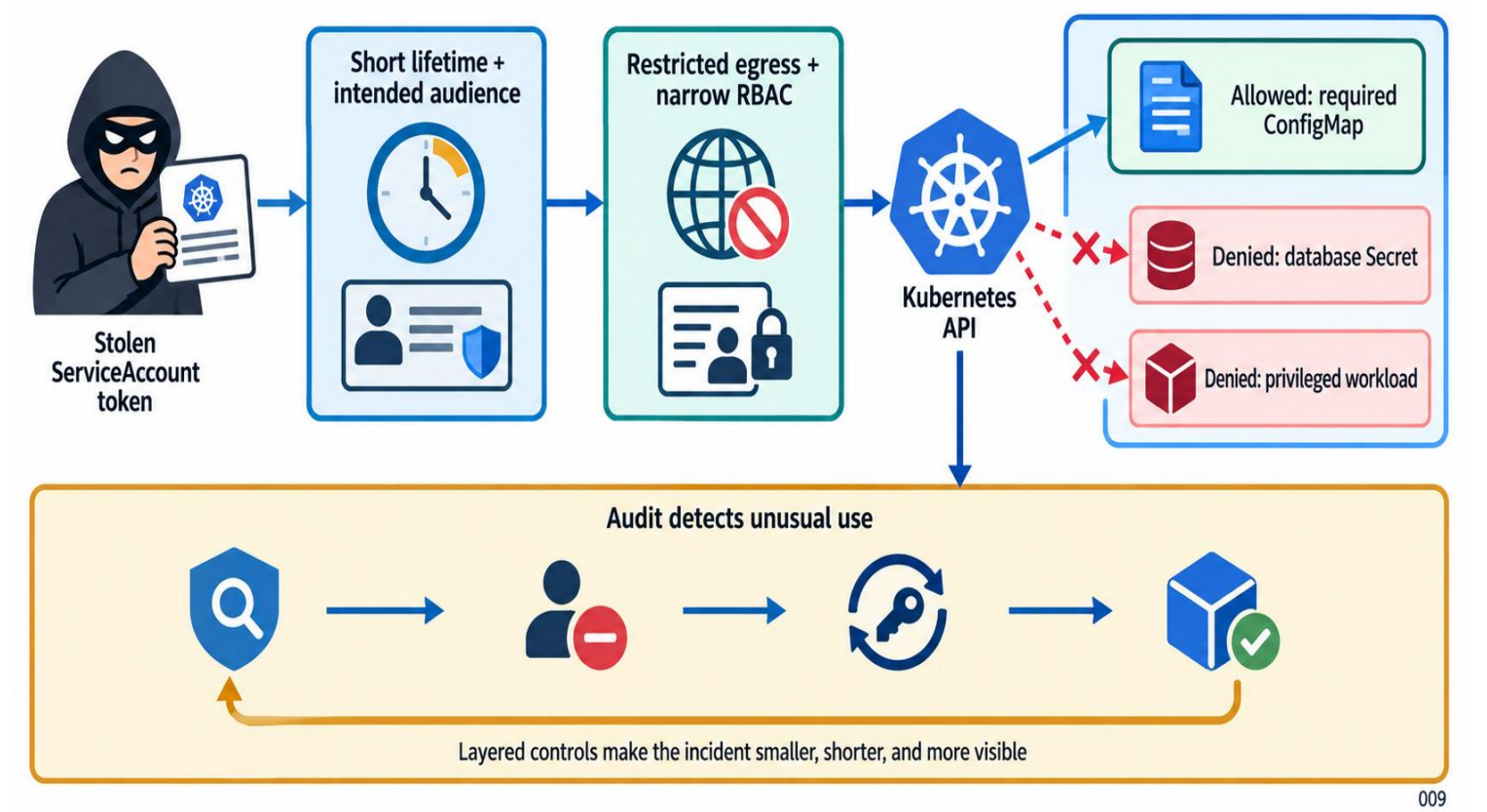


Figure 1-9: Example: a stolen workload token

1. **The token has a short lifetime.** The window for use is limited.
2. **The token has a specific audience.** A service that checks token audiences will reject it outside its intended use.
3. **RBAC is narrow.** The identity can read only the one ConfigMap required by the API.
4. **Network egress is restricted.** The compromised Pod cannot freely contact the API server or external collection systems unless those paths are required.
5. **Secrets are separated.** The database Secret is not readable by the ServiceAccount and is not mounted into unrelated containers.
6. **Admission policy blocks dangerous Pods.** The attacker cannot use permitted workload changes to request host access or privileged execution.

7. **Audit evidence exists.** Unusual requests by the ServiceAccount can be investigated.
8. **Rotation is prepared.** The team can stop the workload, revoke authorization, replace dependent credentials, and deploy a known-good version.

The token was still stolen, but the incident is smaller, shorter, and more visible. **That is the practical purpose of layered security.**

High-frequency threats to recognize early

Beginners should learn to identify a small set of recurring, high-impact conditions:

- A shared or stolen administrative kubeconfig.
- A ServiceAccount with permissions far beyond its workload's purpose.
- Permission to create Pods combined with access to sensitive Secrets or host resources.
- A vulnerable or untrusted image entering through an uncontrolled build path.
- A Service unintentionally exposed outside the intended network.
- A privileged add-on, operator, or admission webhook with broad cluster access.
- A compromised node that can observe or alter multiple workloads.

These risks appear in different forms across environments, but they all involve excessive trust, weak boundaries, or insufficient containment.

Misconceptions that lead to weak designs

“A managed cluster is secure by default.” A provider can secure only the components and settings it owns. Customer RBAC, workloads, data, and exposure remain customer decisions.

“A container is a lightweight virtual machine.” Standard containers share the node kernel. Their isolation is valuable, but a kernel or node compromise can cross workload boundaries.

“Passing a compliance check proves the cluster is secure.” Compliance can confirm that selected controls exist at a point in time. It does not prove resistance to every current threat or verify that permissions are used safely.

“A security control matters only if it blocks deployment.” Controls can also reduce blast radius, improve detection, protect evidence, or make recovery reliable. Prevention is one part of the security outcome.

A working security model

Kubernetes security begins with assets, trust, and consequences—not with individual YAML fields. Use defense in depth so that controls back one another up, apply least privilege to reduce blast radius, and maintain a small threat model that follows real identities and communication paths. The remaining chapters turn this mental model into concrete cluster decisions.

Chapter Summary

Kubernetes security is layered risk reduction. Protect the control paths around workloads as well as visible data, grant only necessary authority, and assume that individual controls can fail.

Review Questions

1. Which Kubernetes assets matter besides application data?
2. How does defense in depth change the result when one control fails?
3. What would a one-page threat model record for a newly assigned workload?

Chapter 2 — Read the Cluster as a Set of Trust Boundaries

The safest way to understand a Kubernetes cluster is to see it as a collection of privileged components connected by trusted paths. A cluster is not one security boundary. It contains several components that can create workloads, distribute credentials, route traffic, attach storage, or control the host. Security decisions become clearer when each component is examined according to the authority it holds and the consequences of its compromise.

What You'll Be Able to Do

After completing this chapter, you should be able to:

- Describe the authority held by the API server, etcd, controllers, scheduler, nodes, kubelet, runtime, CNI, CSI, and add-ons.
- Trace API, workload-creation, network, and data paths across cluster trust boundaries.
- Distinguish managed-provider responsibilities from responsibilities that remain with the cluster owner.

Control plane and data plane

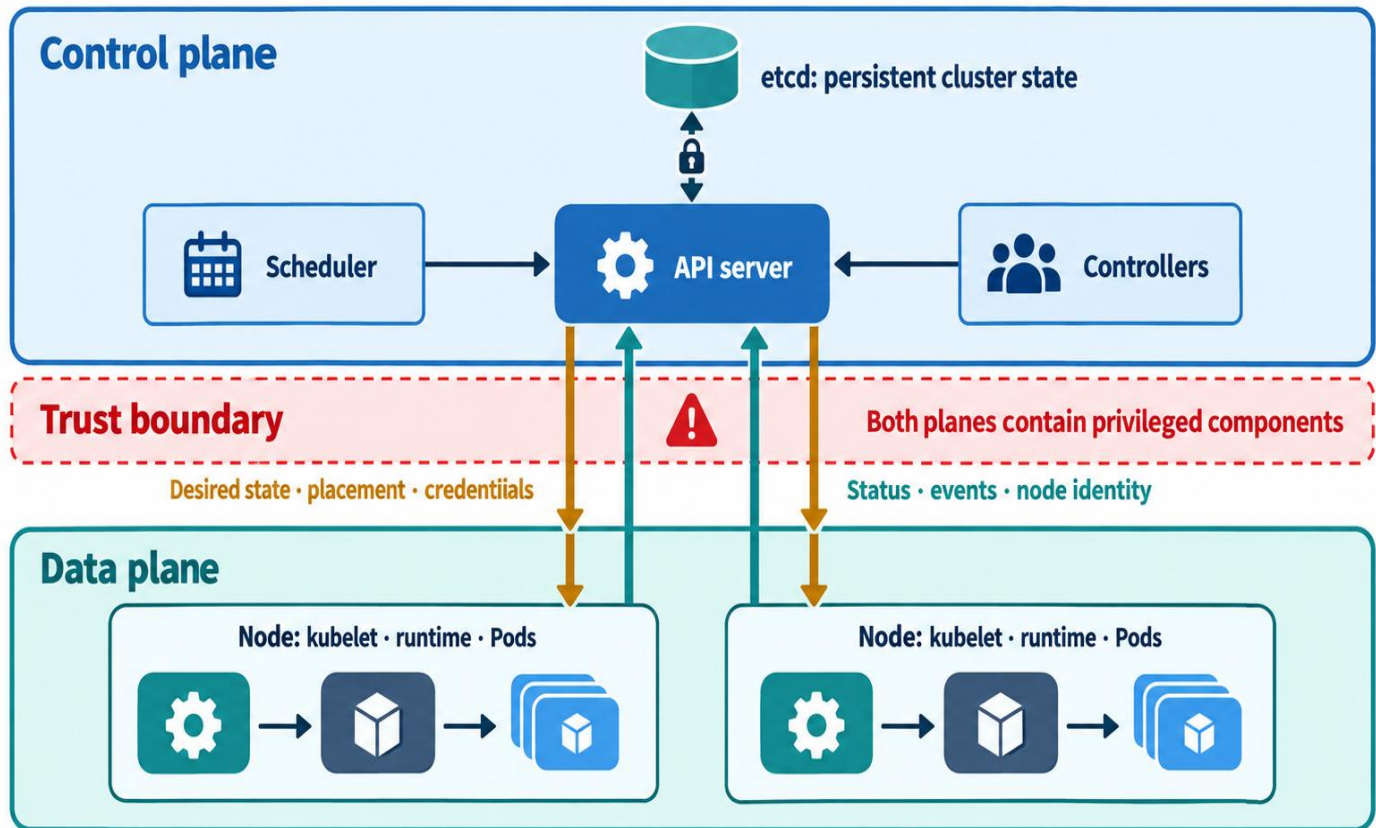
Kubernetes architecture is commonly divided into two broad areas:

- The **control plane** stores desired state and makes cluster-wide decisions about how that state should be realized.
- The **data plane** consists mainly of worker nodes and the workloads running on them.

The names can suggest that the control plane only gives instructions while the data plane only executes them. In practice, both contain powerful identities and interfaces. A controller can create Pods across many Namespaces. A compromised node can observe or manipulate the workloads assigned to it. The boundary is therefore useful for architecture, but it is not a complete statement of trust.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-1 Control plane and data plane



010

Figure 2-1: Control plane and data plane

A good security review asks two questions about every component:

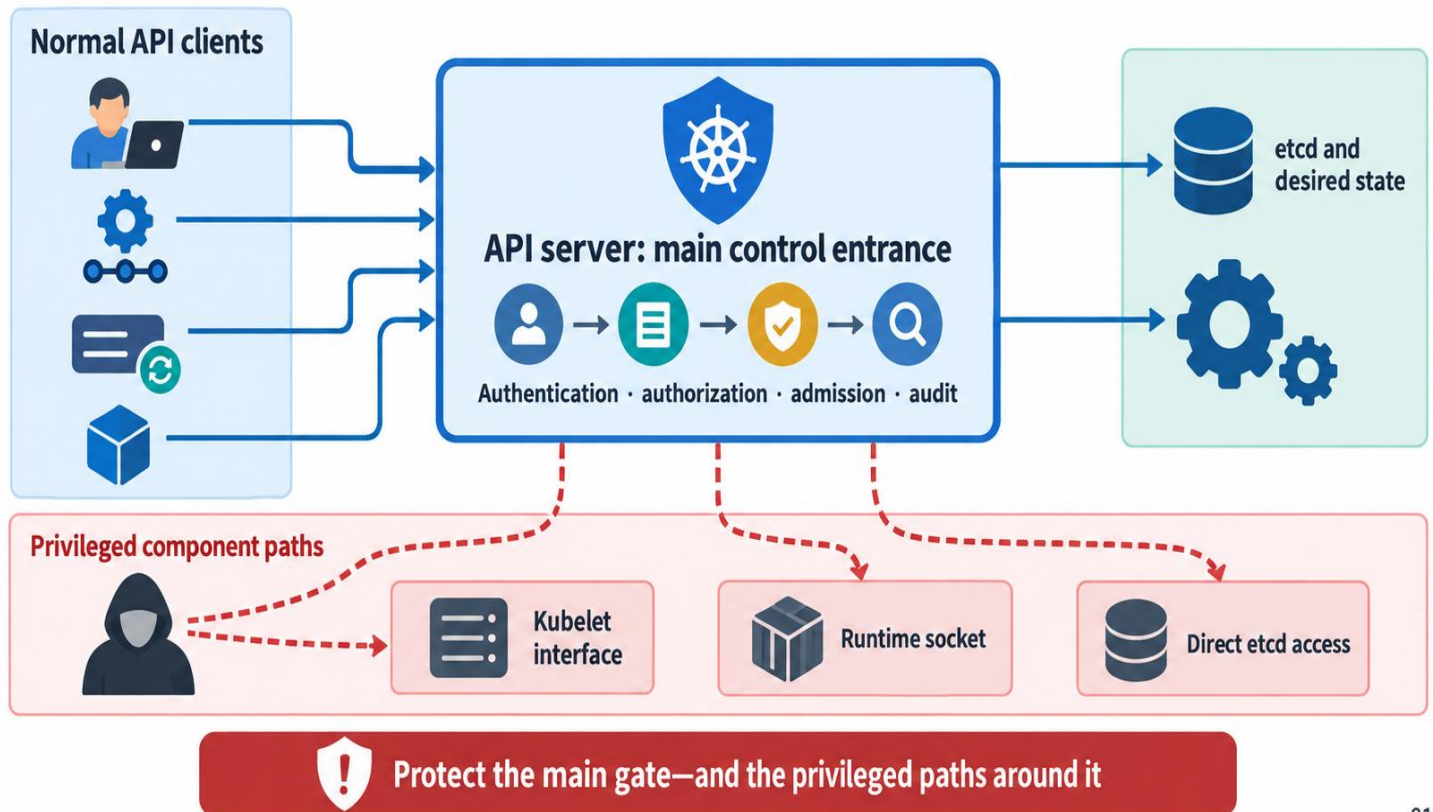
1. **What can this component cause the cluster to do?**
2. **Which credentials, data, or host interfaces can this component access?**

The API server is the main control entrance

The **Kubernetes API server** is the component that exposes the Kubernetes API and processes requests to read or change cluster objects. Commands such as `kubectl apply`, controller updates, scheduler decisions, and many add-on operations pass through it.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-2 The API server is the main control entrance



011

Figure 2-2: The API server is the main control entrance

The API server is comparable to the front desk of a highly controlled facility. It verifies identity, checks permission, applies policy, records the accepted desired state, and returns a result. Because it is the main decision point, it is also a high-value target.

Important security properties include:

- Clients connect using TLS so the server identity and traffic are protected in transit.
- Requests are authenticated and authorized.
- Admission controls examine many write requests before objects are stored.
- API access is restricted at the network layer where practical.
- Security-relevant requests produce audit evidence.

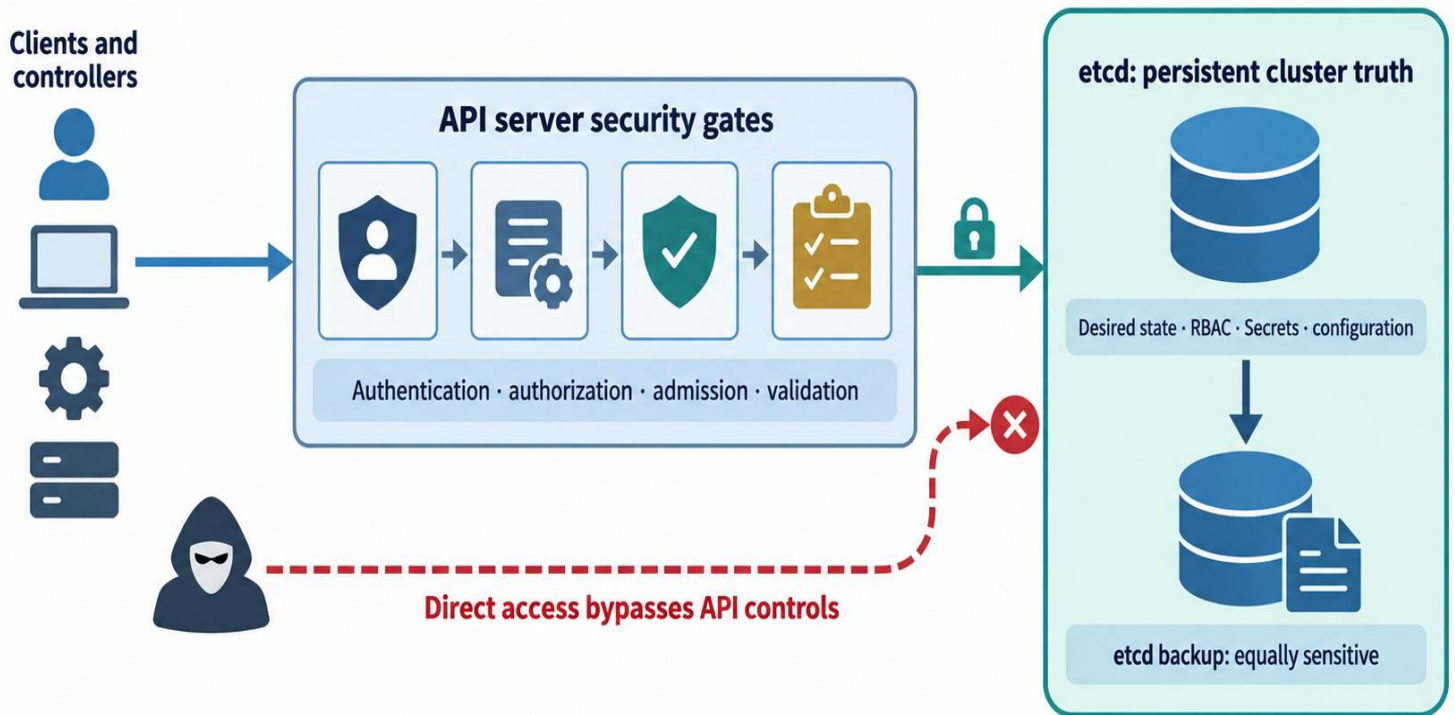
The API server is the main entrance, but it is not the only possible path to privileged behavior. Direct access to etcd, the kubelet, a container runtime socket, or a powerful controller may bypass some API protections. This is why component interfaces must be secured independently.

etcd holds the cluster's persistent truth

etcd is a distributed key-value store used by Kubernetes to persist API data. Deployments, RBAC objects, Secret objects, and much of the cluster's desired state ultimately reside there.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-3 etcd holds the cluster's persistent truth



012

Figure 2-3: etcd holds the cluster's persistent truth

Think of the API server as the guarded service counter and etcd as the secure records room behind it. Strong rules at the counter do not help if an attacker can walk directly into the records room.

A party with unauthorized etcd access may be able to:

- Read sensitive API data, including Secret objects if they are not adequately protected at rest.
- Change stored state without passing through normal authorization or admission controls.
- Corrupt or delete data and damage cluster availability.
- Copy snapshots that contain highly sensitive historical state.

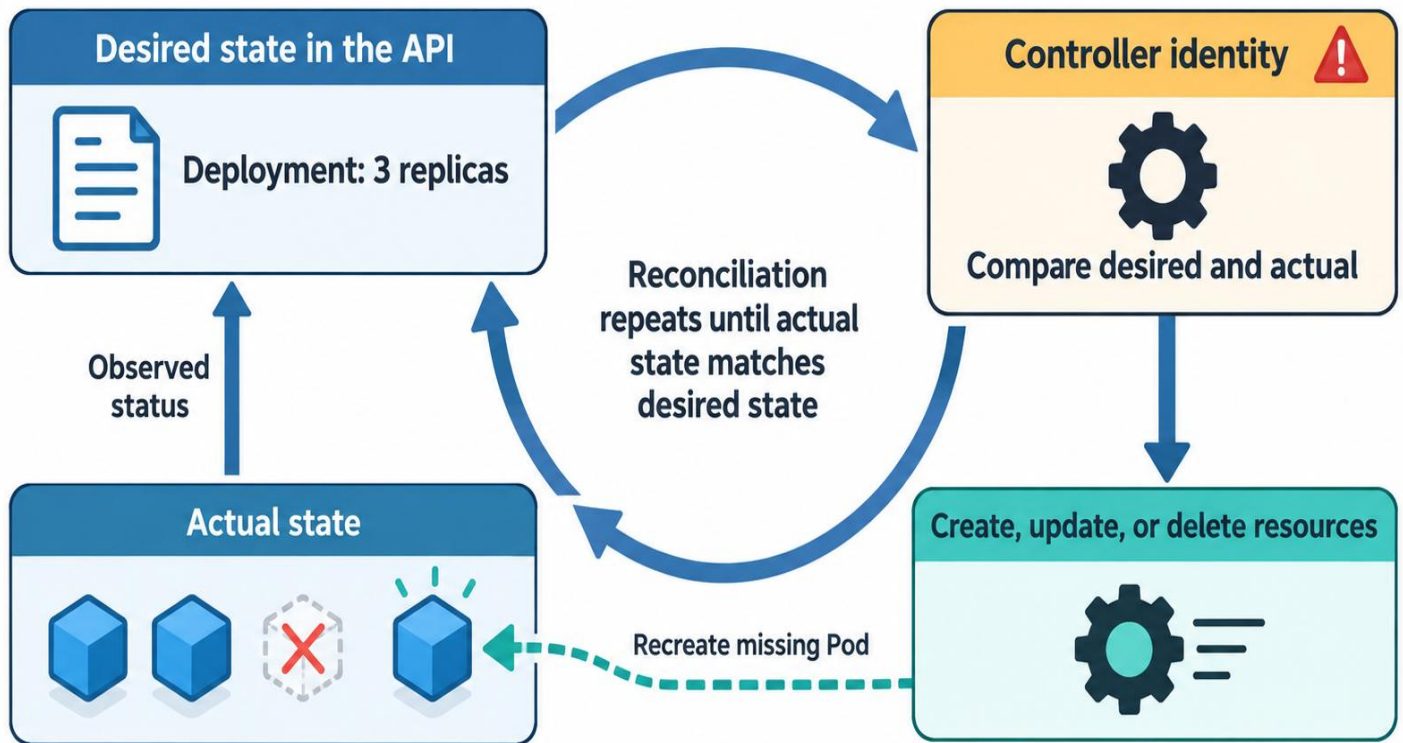
For that reason, etcd network access, client identity, transport encryption, encryption at rest, backups, and restore procedures are control-plane security concerns. Their detailed operation belongs later in the learning path, but the trust principle is needed now: **direct access to the cluster's backing store is at least as sensitive as high-level API access.**

Controllers are trusted automation

A **controller** is a control loop that observes current state and works to make it match desired state. The Deployment controller, for example, creates or updates ReplicaSets so that the requested number of Pods can exist.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-4 Controllers are trusted automation



013

Figure 2-4: Controllers are trusted automation

Controllers are powerful because they act repeatedly and automatically. Their permissions may cover many objects, Namespaces, or even the whole cluster. A compromised or misconfigured controller can therefore reproduce a harmful change after an operator tries to correct it.

For example, deleting a suspicious Pod is not a complete response if the Deployment still declares that Pod template. The controller will create a replacement with the same unsafe configuration. Security review must follow ownership upward from a Pod to its ReplicaSet, Deployment, operator, or GitOps source.

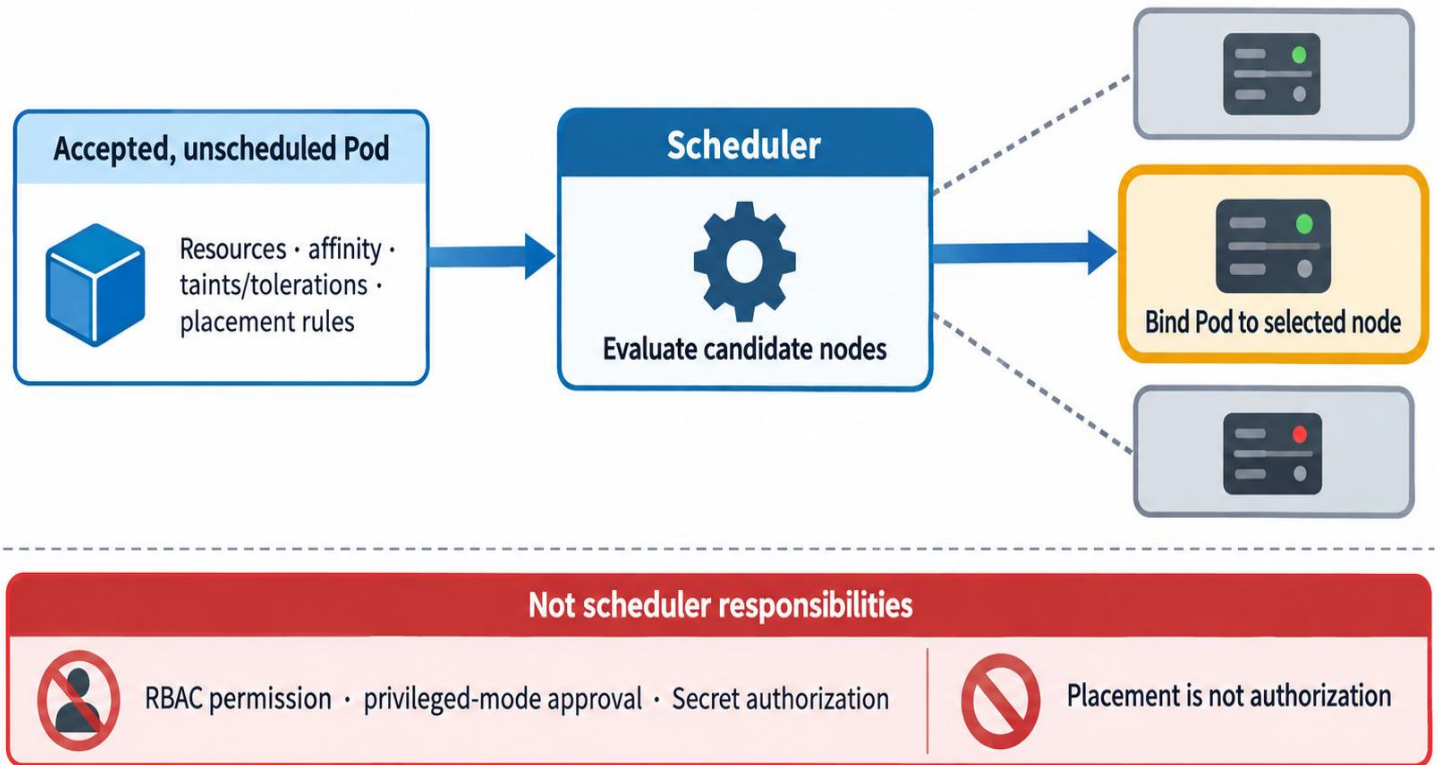
Custom controllers and **operators** deserve the same scrutiny as built-in components. An operator is a controller that manages an application or platform through Kubernetes APIs, often using custom resources. A database operator that appears to manage only database configuration may also be allowed to create Pods, mount Secrets, provision storage, and update cluster-scoped resources.

The scheduler controls placement, not authorization

The **scheduler** selects a suitable node for a Pod that has not yet been assigned. It considers resource requests, placement rules, taints, tolerations, affinity, and other constraints.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-5 The scheduler controls placement, not authorization



014

Figure 2-5: The scheduler controls placement, not authorization

Placement influences risk. A sensitive Pod placed on a shared node inherits exposure to the node's kernel, privileged agents, and other node-level conditions. However, scheduling rules such as node affinity or taints are not complete security boundaries. They guide placement; they do not by themselves stop an authorized party from changing the Pod or gaining control of the node.

Use scheduling controls as part of an isolation design, not as a substitute for identity, admission, node hardening, or stronger runtime isolation.

Nodes concentrate workload risk

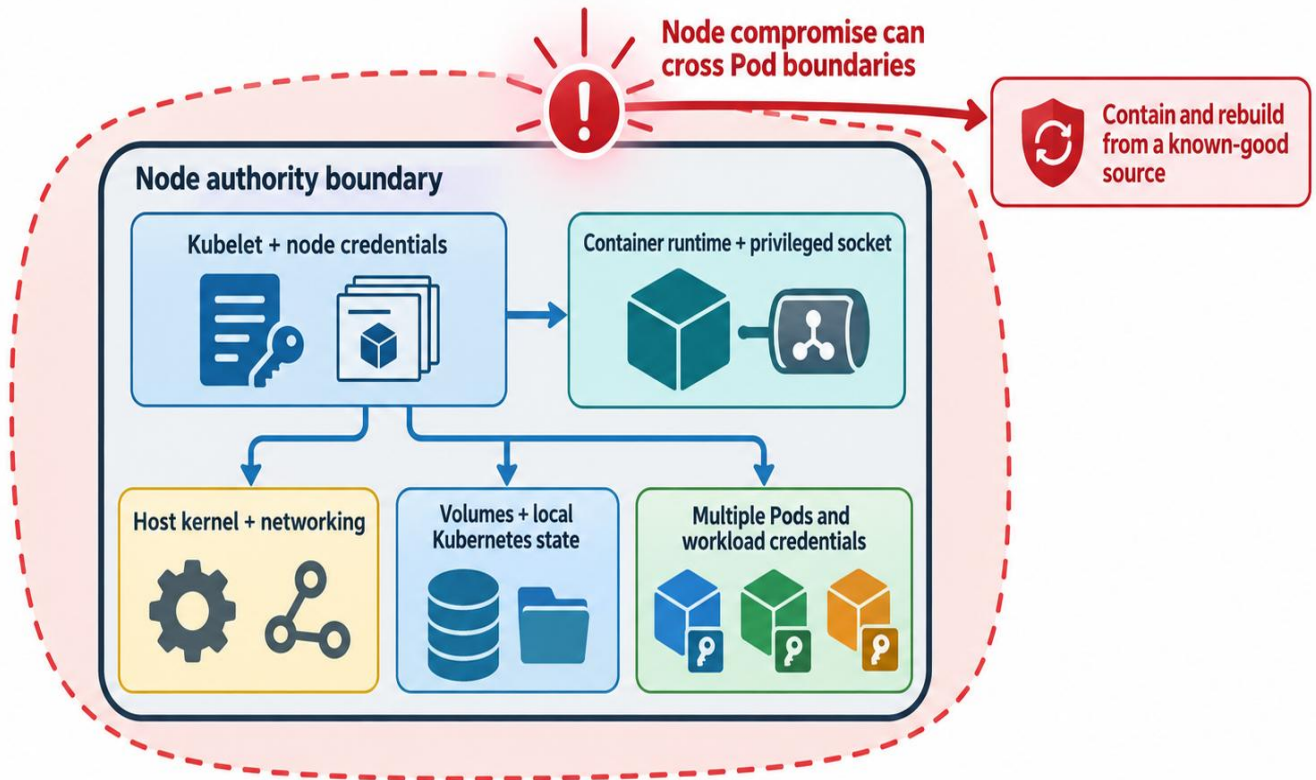
A **node** is a machine—physical or virtual—that runs Pods. Several important components and interfaces meet there:

- The **kubelet** is the node agent that receives Pod assignments and manages their local lifecycle.
- The **container runtime** creates and runs containers through the Container Runtime Interface.
- The host **kernel** provides process, network, filesystem, and device isolation for standard Linux containers.
- Network and storage plugins configure Pod connectivity and volume attachment.
- Privileged DaemonSets may monitor or manage the host.

The node is a high-impact boundary because ordinary containers share its kernel. An attacker with node administrator access, runtime-socket access, or a successful kernel exploit may be able to inspect other containers, read mounted credentials, alter processes, or impersonate node components.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-6 Nodes concentrate workload risk



015

Figure 2-6: Nodes concentrate workload risk

This leads to a foundational rule: **Pod-level controls reduce risk, but they do not make a fully compromised node trustworthy.** A suspected node compromise normally requires containment and rebuilding from a known-good source, not only restarting the affected container.

The kubelet is more than a process launcher

The kubelet communicates with the API server, manages local Pod state, requests or uses credentials, mounts volumes, and exposes interfaces used for node and Pod operations. It is therefore a privileged identity and a sensitive network service.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-7 The kubelet is more than a process launcher

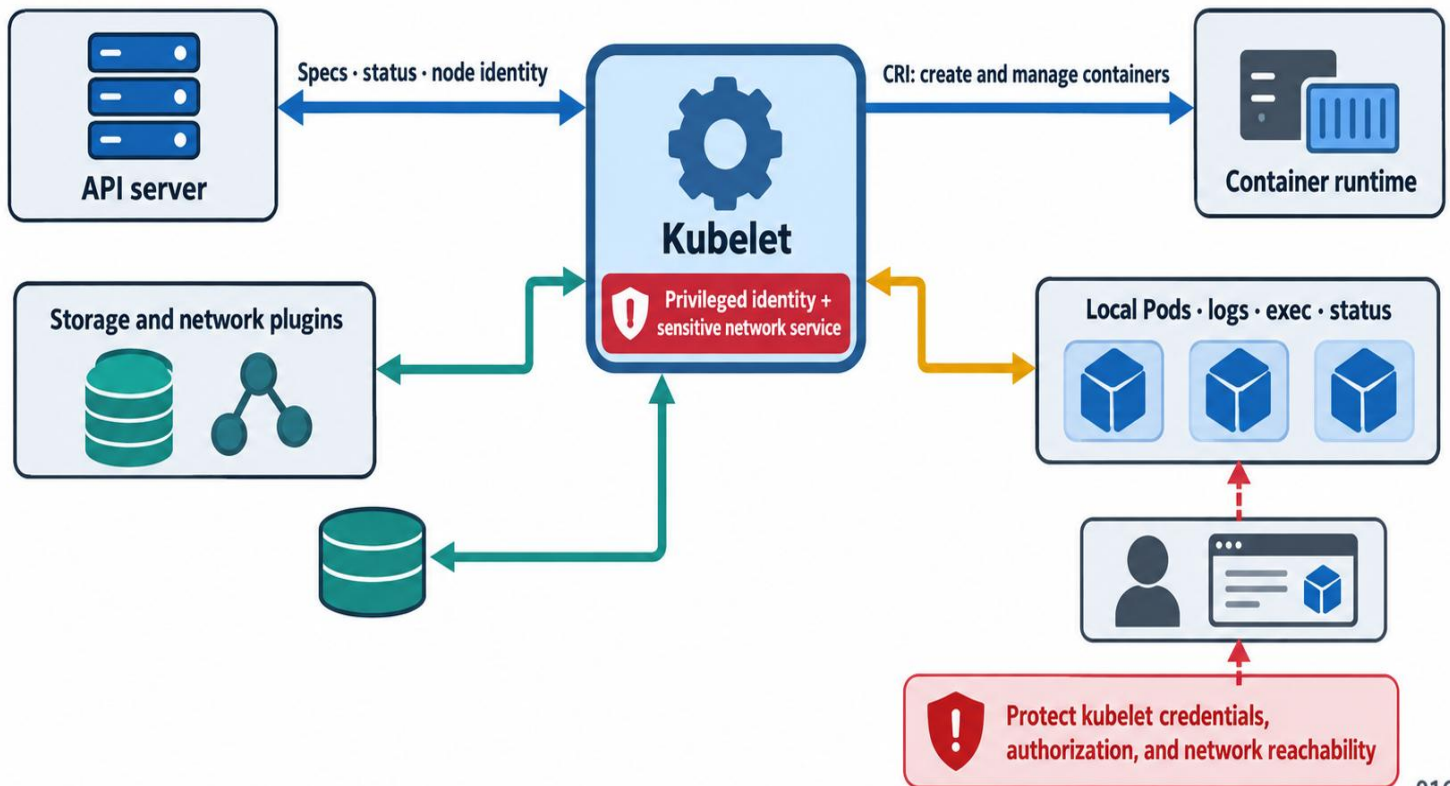


Figure 2-7: The kubelet is more than a process launcher

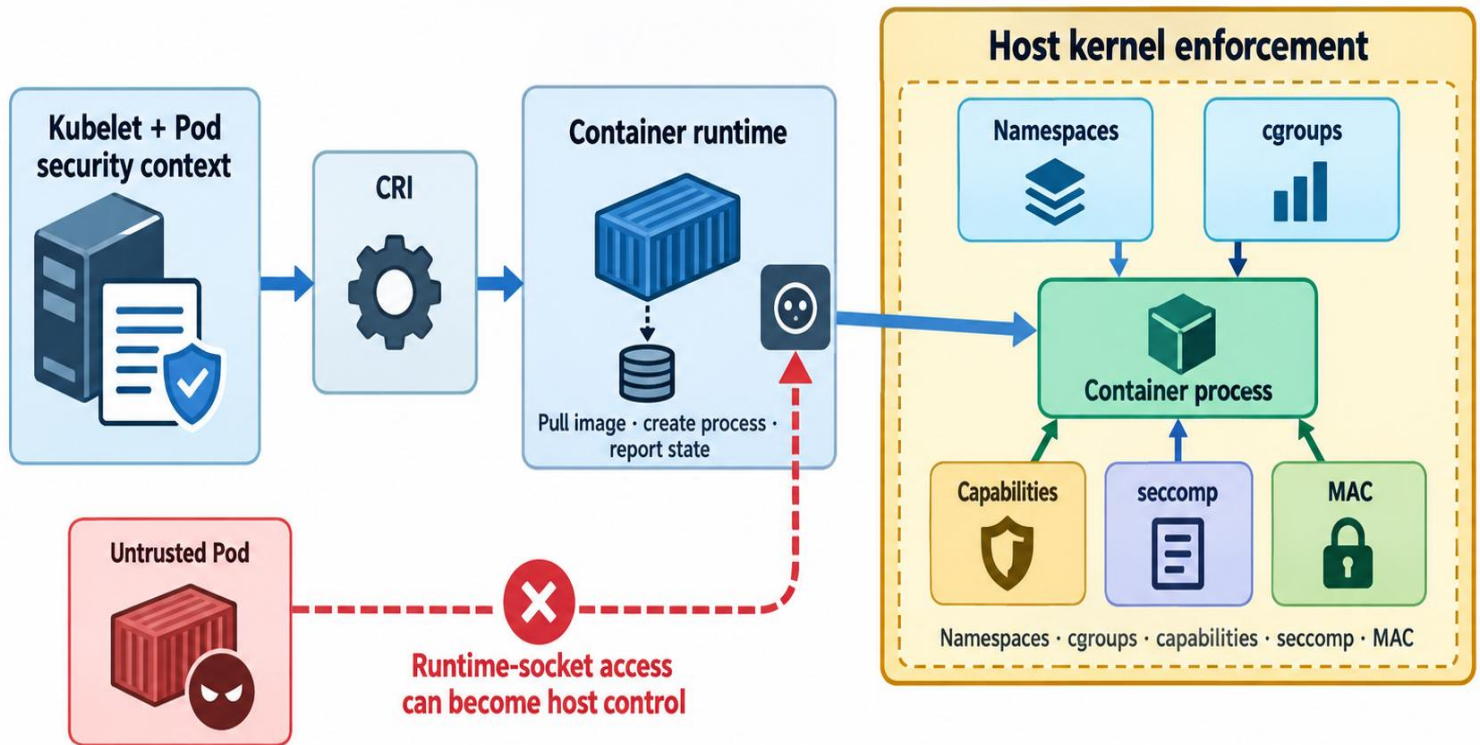
Permissions involving node proxying, kubelet APIs, or node credentials can be more powerful than they first appear. For example, an apparently convenient debugging path may allow command execution in Pods on a node. The details vary by configuration and distribution, but the review question remains consistent: **Can this path cross from cluster API access into direct workload or host control?**

The runtime and host kernel enforce container boundaries

The **container runtime** is the software that pulls images, creates container processes, configures isolation, and reports container state to the kubelet. Kubernetes communicates with it through the **Container Runtime Interface (CRI)**.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-8 The runtime and host kernel enforce container boundaries



017

Figure 2-8: The runtime and host kernel enforce container boundaries

Access to the runtime's privileged socket is commonly equivalent to powerful host access. A user or container that can instruct the runtime may be able to start a highly privileged container, mount host paths, or inspect other containers. The socket is not an ordinary application API and should not be exposed for routine convenience.

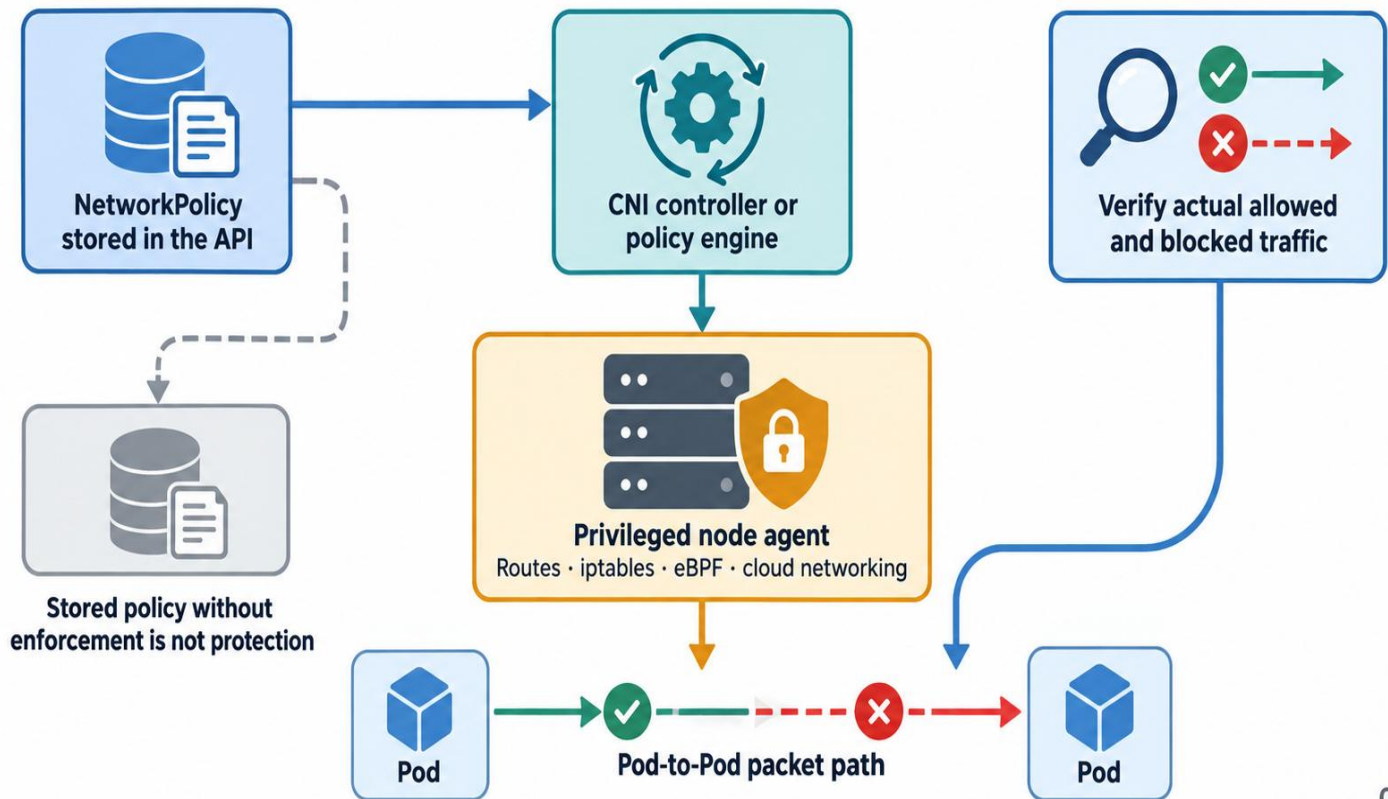
The kernel enforces namespaces, control groups, capabilities, seccomp, mandatory access controls, and other isolation mechanisms. Kubernetes configures many of these through a Pod's security context, but the final enforcement occurs on the node. This is why Pod security and node security are connected rather than independent subjects.

CNI plugins sit on critical network paths

The **Container Network Interface (CNI)** is a specification used by runtimes and network plugins to configure container networking. A Kubernetes CNI implementation may assign Pod addresses, install routes, enforce NetworkPolicy, encrypt traffic, or integrate with cloud networking.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-9 CNI plugins sit on critical network paths



018

Figure 2-9: CNI plugins sit on critical network paths

Network plugins often run privileged components on every node. Depending on their design, they may modify host network rules, load eBPF programs, access network namespaces, or call cloud APIs. A network plugin is therefore part of the cluster's trusted computing base—the set of components whose correct and secure behavior the system depends on.

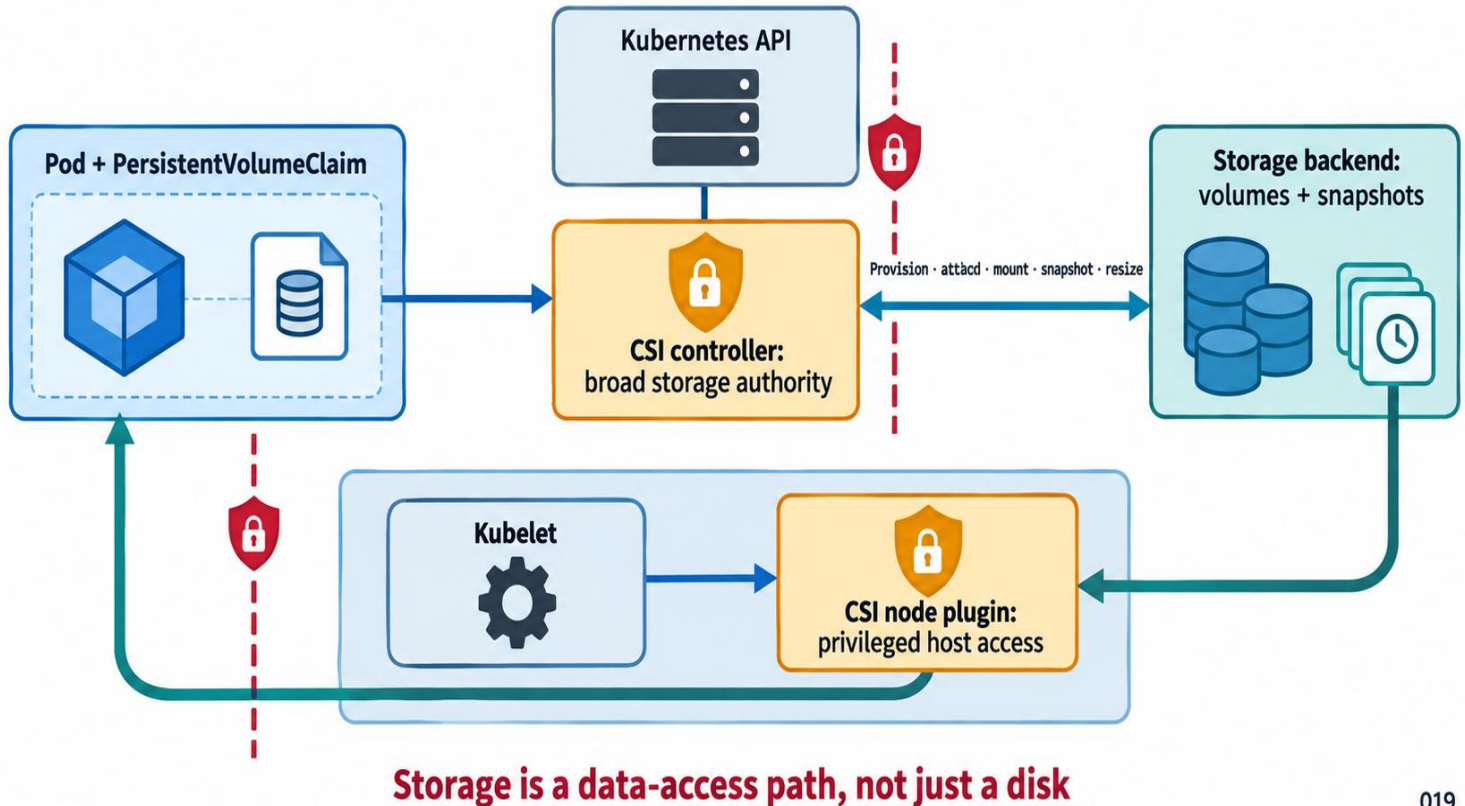
A NetworkPolicy object also provides protection only when the installed network implementation enforces it. The API can accept a valid policy even if the data-plane plugin does not implement the expected behavior. Verification must include the actual traffic result, not only the existence of YAML.

CSI drivers control persistent-data access

The **Container Storage Interface (CSI)** is a standard through which storage systems expose operations such as provisioning, attaching, mounting, snapshotting, and resizing volumes.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-10 CSI drivers control persistent-data access



019

Figure 2-10: CSI drivers control persistent-data access

A CSI driver may include a controller component with broad Kubernetes permissions and a node plugin with privileged host access. That is reasonable for its function, but it makes the driver highly trusted. A compromised storage component could expose data across workloads, mount the wrong volume, access node filesystems, or disrupt many applications.

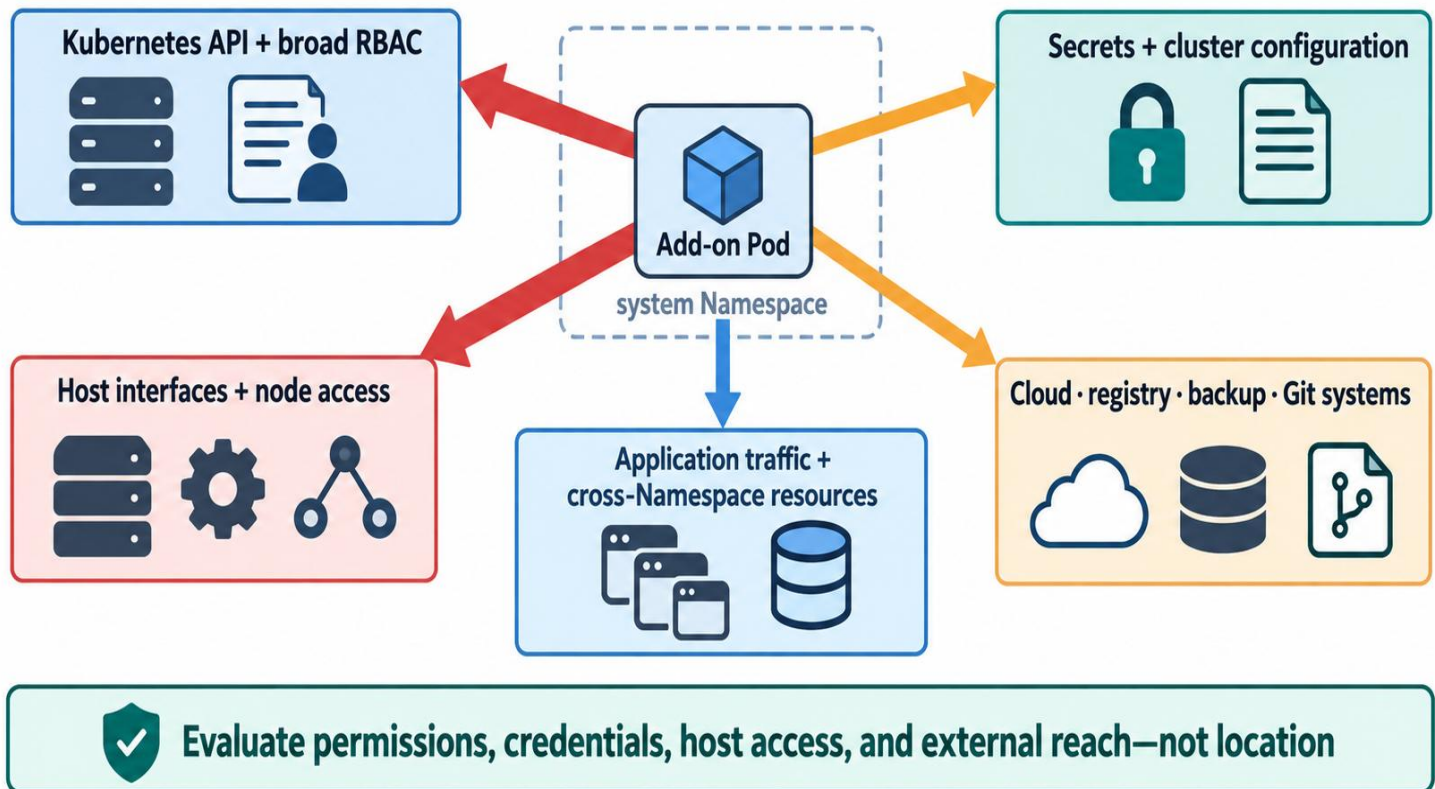
The important beginner lesson is that storage is not “just a disk.” The identities and agents that attach storage can cross important data boundaries.

Add-ons can become hidden cluster administrators

Clusters commonly include DNS, ingress or gateway controllers, metrics agents, certificate managers, policy engines, service-mesh components, backup agents, operators, and GitOps controllers. An **add-on** is an additional component installed to provide cluster or application capabilities beyond the core control plane.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-11 Add-ons can become hidden cluster administrators



020

Figure 2-11: Add-ons can become hidden cluster administrators

The Namespace in which an add-on runs does not prove its trustworthiness. A Pod in a system Namespace may still use an unsafe image, broad RBAC, host mounts, privileged mode, or an exposed management endpoint.

Several add-ons deserve particular attention:

- An ingress controller receives external traffic and often reads routing configuration across Namespaces.
- A certificate controller may create or update TLS Secrets.
- A GitOps controller may modify most application resources continuously.
- A policy engine may determine whether other workloads can be admitted.
- A monitoring or security agent may run on every node with host access.
- A backup controller may read large amounts of application and cluster data.

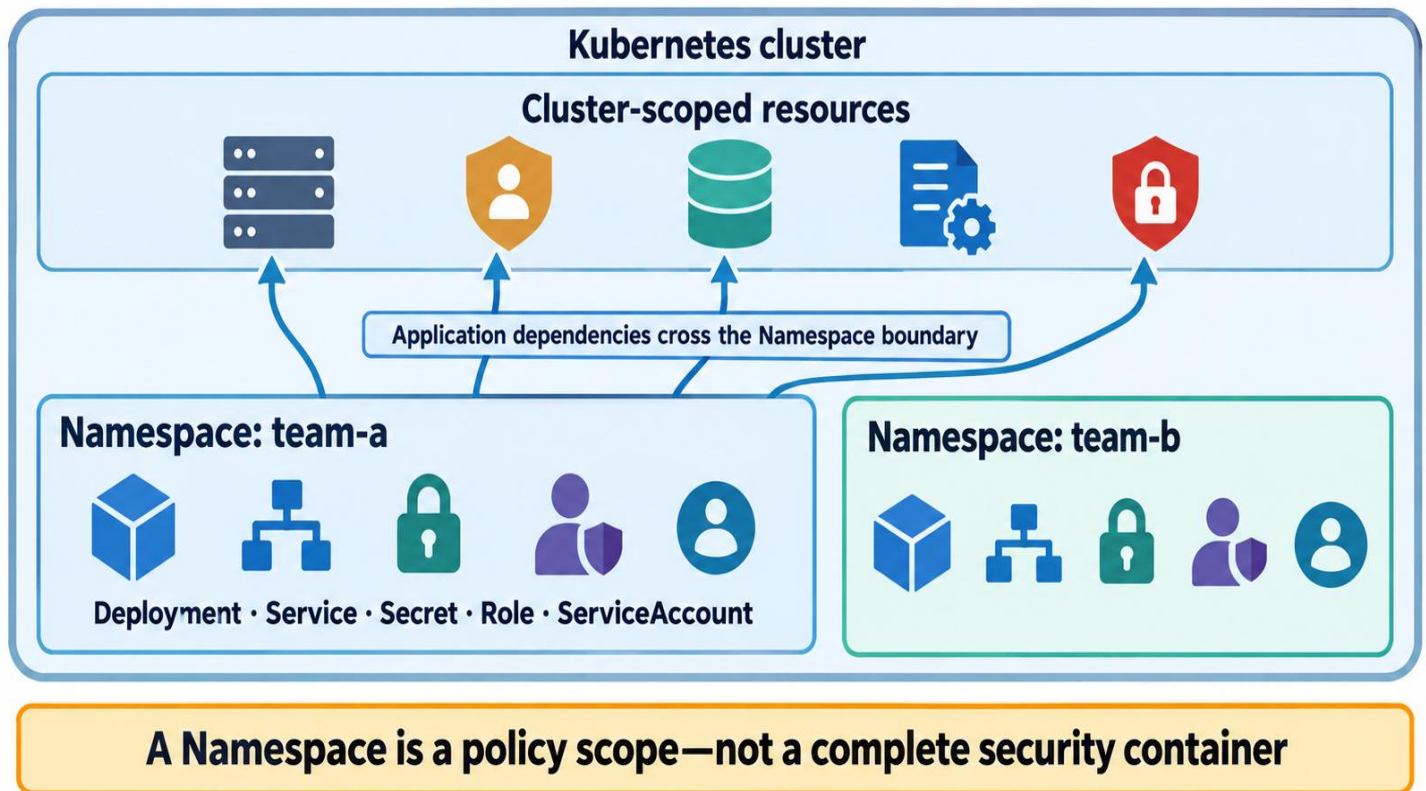
The correct question is not “Is it installed in `kube-system`?” It is “**Which permissions, credentials, host interfaces, and external systems can it reach?**”

Namespaced and cluster-scoped resources

A **Namespace** is a logical scope used to organize many Kubernetes resources and to apply namespaced policies and permissions. Deployments, Services, Secrets, Roles, and ServiceAccounts are commonly namespaced.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-12 Namespaced and cluster-scoped resources



021

Figure 2-12: Namespaced and cluster-scoped resources

Other resources are **cluster-scoped**, meaning that they are not contained by one Namespace. Examples include Nodes, Namespaces themselves, ClusterRoles, ClusterRoleBindings, StorageClasses, CustomResourceDefinitions, and many admission configurations.

This distinction matters because a workload's architecture may extend beyond its Namespace. A team that owns a namespaced application may depend on:

- A cluster-scoped StorageClass.
- A cluster-wide ingress or gateway controller.
- A CustomResourceDefinition installed by an operator.
- ClusterRole permissions bound into the team's Namespace.
- Node labels that influence placement.
- Admission policies that affect every Namespace.

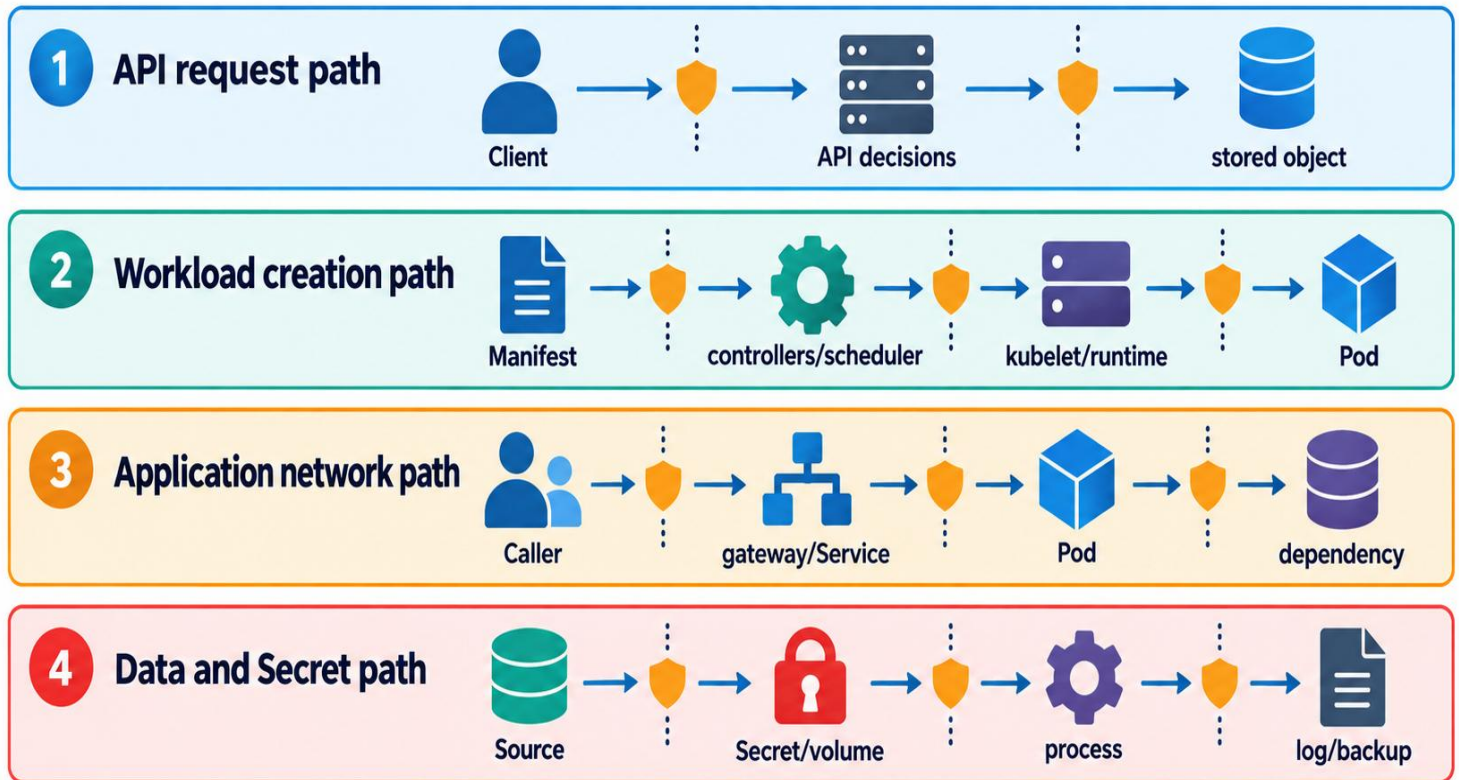
A Namespace is an important policy scope, but it is not a container that holds every dependency or risk associated with an application.

Four paths reveal most security relationships

A practical review can trace four paths through the cluster. Each path highlights different identities and failure modes.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-13 Four paths reveal most security relationships



022

Figure 2-13: Four paths reveal most security relationships

1. The API request path

A client connects to the API server, authenticates, receives an authorization decision, passes applicable admission checks, and reads or writes an object. This path reveals human and automation identities, RBAC, admission policy, and audit evidence.

2. The workload creation path

A Deployment is stored, controllers create a ReplicaSet and Pods, the scheduler selects a node, the kubelet asks the runtime to create containers, and plugins configure network and storage. This path reveals the components that can alter or execute desired state.

3. The application network path

Traffic may enter through a load balancer, ingress or gateway controller, Service, and Pod. East-west traffic may pass between several workloads and external dependencies. This path reveals exposure, TLS termination, service identity, and NetworkPolicy requirements.

4. The data and Secret path

A credential or data set moves from its authoritative source into a Kubernetes object, external store, volume, environment variable, process, log, backup, or support bundle. This path reveals where confidential material exists and which identities can copy it.

Tracing these paths is more effective than reviewing objects in isolation. A narrowly permitted Pod may still be unsafe if a GitOps controller can replace it from an unprotected repository. A well-protected Secret may still leak through application logs.

Walkthrough: from `kubectl apply` to a running container

Consider an engineer applying a Deployment for the internal API.

1. **The local client selects a context.** The kubeconfig determines the cluster endpoint and credential source. A wrong context can send the change to the wrong cluster.
2. **The client opens a TLS connection to the API server.** The client must trust the server certificate and present or obtain credentials.
3. **The API server authenticates the request.** The request becomes a username and group set.
4. **Authorization evaluates the requested action.** RBAC or another authorizer decides whether that identity may create or update the Deployment in the target Namespace.
5. **Admission evaluates the object.** Built-in controls, policies, or webhooks may default fields, reject unsafe settings, or require metadata.
6. **The accepted object is persisted.** The desired Deployment state becomes part of cluster data stored through the API server.
7. **The Deployment controller acts.** Its identity creates or updates a ReplicaSet.
8. **The ReplicaSet controller creates a Pod.** The Pod carries a ServiceAccount name, image reference, security context, volumes, and scheduling requirements.
9. **The scheduler binds the Pod to a node.** Placement rules and available capacity influence the decision.
10. **The kubelet prepares the Pod.** It obtains the Pod specification, arranges credentials and volumes, and calls local plugins.
11. **The runtime starts the containers.** The node kernel enforces the configured process boundaries.
12. **The network and storage plugins complete setup.** The Pod receives connectivity and any required volumes.
13. **The workload begins communicating.** Services, NetworkPolicies, application authentication, and external controls now influence its reachable paths.

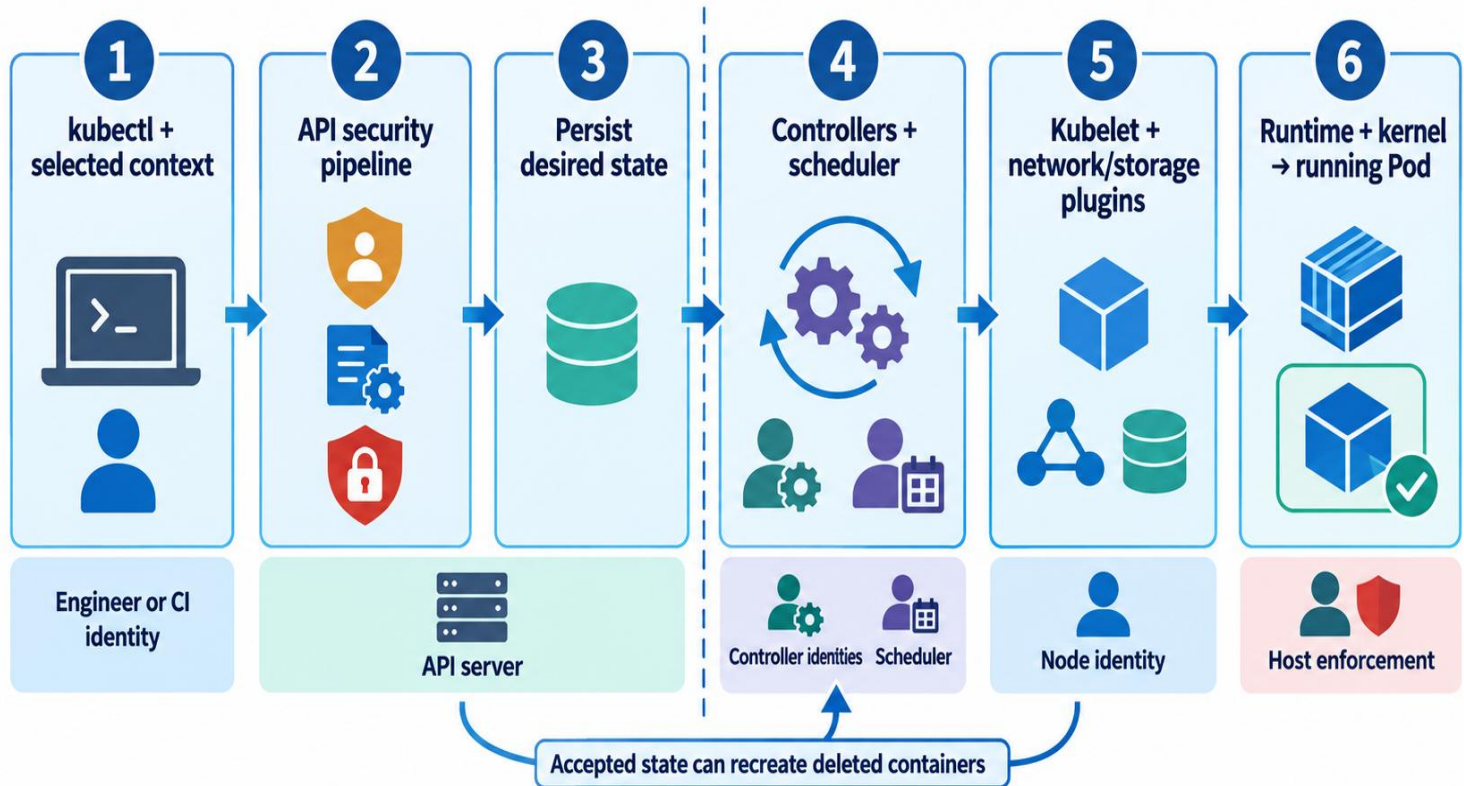
At every step, ask which identity acts and what compromise would mean:

Step	Acting identity or trust	Consequence of compromise
Client request	Engineer or CI/CD identity	Unauthorized desired-state changes
Admission	Policy component identity and configuration	Unsafe objects accepted or legitimate changes blocked
Reconciliation	Controller identity	Harmful resources created repeatedly at scale
Scheduling	Scheduler and placement metadata	Sensitive workloads placed on unsuitable nodes
Node execution	Kubelet, runtime, kernel, plugins	Cross-workload or host-level compromise
Application traffic	Workload and network identities	Data access, lateral movement, or exfiltration

This flow also explains why deleting a running container is not enough to remove risk. The accepted desired state and the controllers above it may recreate the same container.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-14 Walkthrough: from kubectl apply to a running container



023

Figure 2-14: Walkthrough: from kubectl apply to a running container

Inventory privileged components before trusting them

A useful cluster review starts with components that can affect many workloads. For each add-on or controller, record:

- Whether it can create or modify Pods and workload controllers.
- Whether it can read Secrets, ConfigMaps, or token-bearing objects.
- Whether it can change Roles, bindings, admission policy, or cluster-scoped resources.
- Whether it runs on every node.
- Whether it uses host networking, host namespaces, host paths, or runtime sockets.
- Whether it calls external systems using cloud, registry, or repository credentials.
- Which team owns it and how it is updated.

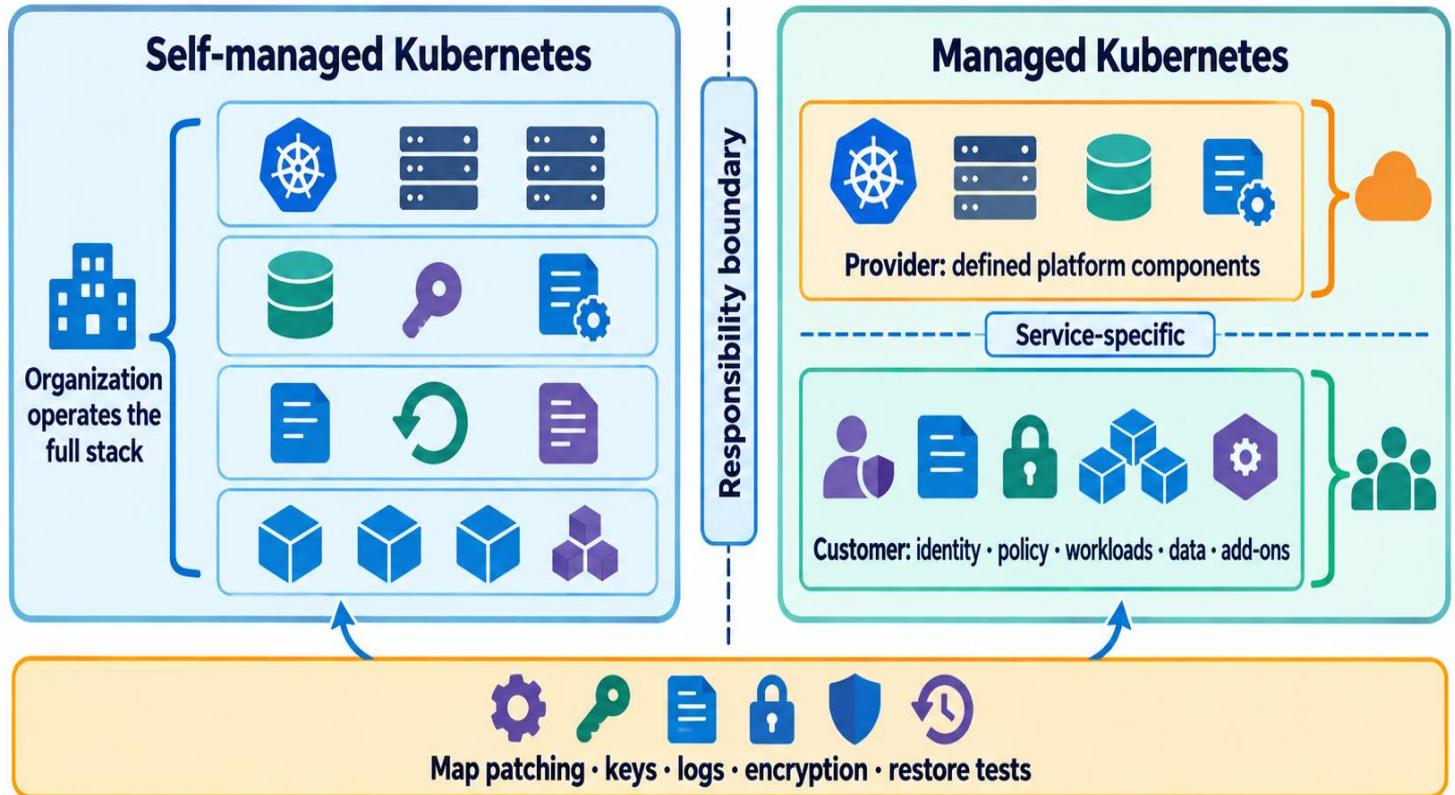
A component with several of these properties should be treated as security-critical even when its visible business function appears read-only. A metrics collector, for example, may require node and Pod access that creates a meaningful attack path.

Self-managed and managed responsibility boundaries

In a self-managed cluster, the organization may operate the API server, etcd, scheduler, controllers, certificates, backups, and nodes. In a managed service, the provider may hide or operate some of those components.

Chapter 2: Read the Cluster as a Set of Trust Boundaries

2-16 Self-managed and managed responsibility boundaries



024

Figure 2-15: Self-managed and managed responsibility boundaries

The architectural path still exists even when a provider manages part of it. The security review should replace inaccessible implementation details with responsibility questions:

- Who patches each component?
- Who controls network exposure?
- Who manages keys and certificates?
- Who can retrieve control-plane or audit logs?
- Who configures encryption at rest?
- Who tests backup restoration?
- Which customer-created extensions enter the control-plane trust path?

This avoids two errors: assuming the provider handles a customer setting, and duplicating controls that the provider already operates.

Misconceptions that hide trust relationships

“Everything for an application is inside its Namespace.” Cluster-scoped resources, node components, shared controllers, storage classes, and external systems can all affect a namespaced workload.

“A system Namespace makes an add-on trustworthy.” Namespace placement is organization, not proof. Trust must follow image origin, permissions, host access, configuration, and ownership.

“A component that only collects data is low risk.” Read access can expose credentials, workload content, topology, or sensitive telemetry. Collection agents may also need powerful node access.

“The scheduler provides hard isolation.” Placement controls are useful, but they do not replace authorization, node trust, or runtime isolation.

The cluster as connected authority

Kubernetes security follows authority and communication paths. The API server, etcd, controllers, nodes, runtimes, CNI and CSI plugins, and add-ons each occupy a distinct trust position. Trace requests, workload creation, network traffic, and sensitive data from source to destination. When the identity and consequence at each step are visible, later controls such as RBAC, Pod hardening, admission policy, NetworkPolicy, and Secret handling become much easier to design correctly.

Chapter Summary

A Kubernetes cluster is a connected set of privileged components, not one uniform boundary. Security reviews should follow authority, credentials, data, and host access across each path.

Review Questions

1. Why is the control-plane/data-plane distinction useful but incomplete as a trust model?
2. Which components can cross from Kubernetes API authority into host or data access?
3. How should responsibility be verified in a managed Kubernetes service?