Kubernetes Security Fundamentals (LFS460)

Introduction

- Objectives
- The Linux Foundation
- Linux Foundation Training
- Certification Programs and Digital Badging

Course Objectives

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

- Understand Kubernetes architecture and its core components.
- Deploy, manage, and scale containerized applications in Kubernetes.
- Work with Pods, Services, Deployments, and Volumes effectively.
- Understand security concepts, including authentication, RBAC, and policies.
- Use ConfigMaps, Secrets, and probes for application configuration and health checks.
- Troubleshoot, monitor, and expose applications inside and outside the cluster.

About The Linux Foundation

- The Linux Foundation (LF) is a non-profit consortium founded in 2000.
- It promotes open source innovation and protects the Linux ecosystem.
- Hosts hundreds of open source projects, including:
- Kubernetes, Cloud Native Computing Foundation (CNCF)
- Linux Kernel, Hyperledger, Node.js, OpenTelemetry, and more.

Training Philosophy

- Hands-on, practical, real-world focused.
- Created and maintained by active industry experts.
- Supported by major cloud providers (AWS, GCP, Azure).
- Designed to align with CNCF certifications.

Certification Programs and Digital Badging

Why Certification Matters:

- Validates your skills with recognized credentials.
- Demonstrates hands-on expertise, not just theory.
- Recognized globally by employers and the open source community.

Digital Badging:

- Upon certification, you receive a verifiable digital badge.
- Share it on LinkedIn, GitHub, or your company profile.
- Each badge links to an online record of authenticity.

Cloud Security Overview

- Multiple Projects
- What is Security?
- Assessment
- Prevention
- Detection
- Reaction
- Classes of Attackers
- Types of Attacks
- Attack Surfaces
- Hardware and Firmware Considerations
- Security Agencies
- Manage External Access

Multiple Projects

Multiple Project?

Modern Kubernetes environments rarely consist of a single cluster or a single project. Organizations typically manage **multiple clusters** spread across **different cloud projects**, **accounts**, or **regions**, each hosting distinct environments such as development, staging, and production.

From a security standpoint, this multi-project or multi-cluster setup introduces both **stronger isolation possibilities** and **new governance challenges**. Security must be treated as a **system property** that spans clusters, projects, and the networks between them.

The Security Dimension of Multi-Project / Multi-Cluster Architectures

Layer	Scope	Primary Security Goal	Example Controls
Cloud / Project Layer	Separation between administrative domains	Prevent cross-project compromise	Distinct GCP Projects, AWS Accounts, Org Policies
Cluster Layer	Runtime isolation	Contain cluster-level privilege escalation	Dedicated clusters per environment or tenant
Namespace Layer	Logical multi-tenancy inside a cluster	Limit cross-team access	RBAC RoleBindings, ResourceQuotas, NetworkPolicies
Workload Layer	Pod and container runtime isolation	Restrict privilege escalation	PodSecurity, AppArmor, seccomp, non-root enforcement
Policy / Governance Layer	Global enforcement across clusters	Maintain configuration consistency	OPA Gatekeeper, Kyverno, Policy Controller

Why It Matters

1. **Blast Radius Containment**
   Multi-cluster architectures naturally reduce the impact of a breach.
   Compromising a single cluster (e.g., through an unpatched API server or misconfigured workload) should not provide access to others.

2. **Operational and Compliance Isolation**
   Certain clusters may process regulated data or have stricter logging and retention policies.
   Segregating these workloads ensures compliance and simplifies audits.

3. **Shared Responsibility Clarification**
   Even when using managed offerings (GKE, EKS, AKS), everything inside the worker nodes — from Pods to RBAC and Secrets — remains the customer's responsibility.
   Isolation at the project and cluster level prevents one operational domain from affecting another.

4. **Controlled Autonomy**
   Teams can manage their clusters independently while still adhering to central governance standards (e.g., mandatory encryption, required labels, restricted container privileges).

Common Pitfalls

1. **Single Shared Cluster for All Environments**
   Running dev, staging, and production workloads in one cluster collapses isolation boundaries.
   Any compromise in development can threaten production.

2. **Overly Broad RBAC Bindings**
   Granting cluster-wide roles (cluster-admin) to users or service accounts removes namespace protection and enables lateral movement.

3. **Unrestricted Network Access**
   Without default-deny NetworkPolicies, Pods can communicate freely across namespaces — effectively disabling segmentation.

4. **Shared Service Accounts and Secrets**
   Using the same service account across clusters or namespaces exposes credentials globally.

5. **Inconsistent Governance**
   Without centralized policy validation, each cluster drifts toward its own configuration, increasing the attack surface.

Design Recommendations

- **Separate Clusters per Environment or Business Domain**
  - Each cluster runs in its own cloud project or account.
  - Use federation or GitOps pipelines for consistent configuration.
  - Apply environment-specific RBAC, network, and audit policies.
- **Namespace Isolation Within Clusters**
  - Treat namespaces as the smallest unit of tenancy.
  - Apply RoleBindings, ResourceQuotas, LimitRanges, and NetworkPolicies consistently.
  - Default-deny ingress/egress rules; explicitly whitelist inter-namespace traffic.

Design Recommendations

- **Dedicated Service Accounts and Identities**
  - Create workload-specific service accounts per cluster.
  - Never reuse credentials between environments.
  - Integrate with external IAM (GCP Workload Identity, IRSA, AAD Pod Identity) for short-lived credentials.
- **Unified Policy Enforcement**
  - Deploy policy controllers (OPA Gatekeeper, Kyverno) to enforce consistent rules across clusters — for example, image provenance, mandatory annotations, non-root containers.
  - Use admission webhooks for environment-wide validation.
- **Centralized Visibility with Controlled Access**
  - Aggregate logs and audit events into a secure monitoring project.
  - Enforce least privilege for read access — observability does not equal administrative control.

What is Security?

What Is Security? (Cloud Native 4C Security Model)

In cloud-native environments, security is not defined by traditional perimeter defenses or static infrastructure. Instead, it is structured around a layered model where each layer builds upon and protects the others.

The **4C Security Model** — Cloud, Cluster, Container, Code — is the foundational way to understand security in Kubernetes environments. It shows that the **security of each layer depends on the layer beneath it**, and that weaknesses at lower levels propagate upward.

Cloud (The Foundation Layer)

Key Concerns

- **Identity and Access Management (IAM)**
  - Misconfigured cloud IAM can fully compromise Kubernetes, regardless of RBAC or network policies.
- **Network boundaries**
  - VPC/VNet segmentation, firewall rules, private endpoints.
- **Storage encryption and key management**
  - Secrets, container images, persistent volumes.
- **Node isolation and virtualization**
  - VM escape vulnerabilities or weak hypervisor isolation undermine node and Pod security.
- **Audit logging and monitoring**
  - Cloud audit logs reveal misuse of IAM, network anomalies, or privilege escalations.

Relevance to Kubernetes

If the cloud layer is compromised, **the cluster is compromised**.
If IAM permissions allow:

- deleting nodes,
- reading etcd disks,
- modifying networking rules, all Kubernetes protections collapse.

Cluster (Kubernetes Control Plane & Node Layer)

Key Concerns

- **API server access controls**
  - Authentication/authorization (RBAC), audit logging, admission control.
- **Node security**
  - Kubelet hardening, minimal OS surface, sandboxed runtimes.
- **etcd protection**
  - Encryption at rest, TLS for all traffic, strict access control.
- **Network segmentation**
  - CNI enforcement, NetworkPolicy, egress controls.
- **Admission control governance**
  - OPA Gatekeeper/Kyverno enforcing policy baselines across namespaces.

Relevance to Kubernetes

The cluster layer is the **primary security boundary** in Kubernetes.
Even if cloud infrastructure is secure, a cluster with:

- open kubelets,
- permissive RBAC,
- no admission control, is trivial to compromise.

Container (Workload Isolation Layer)

Key Concerns

- **Image provenance**
  - Signed images, trusted registries, Supply Chain Security (SLSA, Sigstore).
- **Runtime restrictions**
  - No privileged mode, no host mounts, seccomp, AppArmor, SELinux.
- **Resource boundaries**
  - CPU, memory limits to prevent DoS.
- **Network boundaries**
  - Pod-level policies, sidecar behavior, egress restrictions.
- **Ephemeral identity**
  - Service accounts bound only to relevant namespaces.

Relevance to Kubernetes

Even if the cluster is secure, a compromised container with:

- privilege escalation,
- hostPath mounts,
- or unrestricted capabilities can break out and compromise the node/kernel.

Code (Application Security Layer)

Key Concerns

- Injection attacks (SQLi, command injection)
- Unsafe deserialization
- Lack of input validation
- Hardcoded secrets
- Insecure configuration
- Unrestricted outbound connections

Relevance to Kubernetes

Kubernetes cannot fix application vulnerabilities.

A flaw in the code can lead to:

- reading secrets,
- lateral movement between services,
- privilege escalation inside the application context.

The 4C model emphasizes that **code is only as secure as the three layers below it**.

Why 4C Model Matters More Than Traditional Models

Traditional models (like CIA triad) explain what security aims to protect.
The 4C model explains where and how security must be implemented in Kubernetes.

It matches the reality of cloud-native systems:

- Distributed environments
- Dynamic infrastructure
- Ephemeral containers
- Declarative configuration
- Heavy reliance on supply chain and automation

This is why 4C is used by:

- CNCF
- SIG-Security
- NIST 800-204
- Linux Foundation training
- Kubernetes Security Audits

The CIA Triad

The classic security model — **Confidentiality, Integrity, Availability** — applies differently in Kubernetes than in traditional monolithic systems.

How the Layers Interact

If the Cloud layer is compromised:

- The attacker gains full access to all clusters.

If the Cluster layer is compromised:

- All containers and workloads are compromised.

If the Container layer is compromised:

- Attackers may escape to the node or move laterally.

If the Code layer is compromised:

- Applications leak data, escalate privileges, or expose services.

Each layer **inherits** the weaknesses of the ones below it.

Assessment

Assessment

Before applying hardening, policies, or runtime protections, organizations must first **understand their current security posture**.

In Kubernetes, Assessment means continuously evaluating the configuration and behavior of the cluster across all layers of the 4C model — **Cloud, Cluster, Container, and Code**.

Assessment is not a one-time audit.

It is an ongoing practice that detects drift, misconfigurations, privilege escalation paths, supply chain risks, and unsafe runtime behavior.

This section focuses on **how to assess a Kubernetes environment** systematically and what must be inspected to identify weaknesses early.

Cloud Layer Assessment

Cloud Layer Assessment

Even though clusters are the main focus, most attacks begin with:

- overly permissive cloud IAM,
- misconfigured VPC networks,
- compromised credentials,
- or unmanaged infrastructure components.

Key Cloud Assessment Areas

- IAM role review (ensure least privilege for cluster nodes and service accounts)
- Verify nodes run with minimal cloud permissions
- Check cloud audit logs for unexpected administrative activity
- Validate encryption settings for disks, storage buckets, and container registries
- Ensure private control plane endpoints where possible
- Confirm network boundaries (VPC, peering, firewall rules)

Why it matters

Compromise at the cloud layer allows attackers to:

- modify node pools,
- inject malicious nodes,
- read etcd disks,
- or change cluster networking - making all cluster-level hardening irrelevant.

Cluster Layer Assessment

Cluster Layer Assessment

The cluster layer is the primary control surface for security.

Assessment here focuses on **API server configuration, RBAC, etcd, kubelet, admission control**, and **network isolation**.

Key Cluster Assessment Areas

API Server

- Verify secure authentication mechanisms (OIDC, IAM, certificates)
- Review RBAC bindings for over-permissioned roles (cluster-admin)
- Check anonymous or unauthenticated access is disabled
- Ensure audit logging is enabled and configured properly

etcd

- Confirm TLS encryption for all communication
- Validate encryption-at-rest for Secrets
- Check access paths — only API server should talk to etcd

kubelet

- Ensure kubelet read-only port is disabled
- Confirm authn/authz is enforced
- Verify webhook authorization mode is enabled

Key Cluster Assessment Areas

Network (CNI)

- Check NetworkPolicies exist and follow a default-deny pattern
- Validate namespace segmentation
- Inspect control-plane ↔ node network boundaries

Admission Control

- Ensure policy-based controls exist (OPA Gatekeeper, Kyverno, Pod Security Admission)
- Validate mutation & validation webhooks are enforced
- Check that privileged Pods, hostPath, and hostNetwork access are restricted

Container Layer Assessment

Container Layer Assessment

This focuses on the **workloads running in the cluster** and how they are built, stored, deployed, and executed.

Key Container Assessment Areas

Image Security

- Are images scanned for CVEs?
- Do images come from approved registries?
- Are image signatures required and verified?
- Is the image supply chain traceable (SLSA levels)?

Runtime Restrictions

- Are Pods running as non-root?
- Are capabilities dropped by default?
- Are seccomp / AppArmor / SELinux profiles enforced?
- Is privilege escalation disabled?

Resource and Isolation Controls

- Do all Pods have CPU/memory requests/limits?
- Are sensitive containers isolated using namespaces, cgroups, and seccomp?

Code Layer Assessment

Key Code Assessment Areas

This layer focuses on the application itself — vulnerabilities and logic errors that the platform cannot mitigate.

Key Code Assessment Areas

- Secret handling (no secrets in config maps, no hardcoded credentials)
- Input validation and API security
- TLS enforcement inside services
- Logging and error handling practices
- Dependency scanning (SCA)
- Misconfigured frameworks or libraries
- Excessive outbound connectivity (e.g., pods calling arbitrary external URLs)

Even with perfect cloud, cluster, and container security, **unsafe code can compromise the entire environment.**

Common Tools Used in Kubernetes Assessments

Cloud Layer

- cloud IAM analyzers (GCP IAM recommender, AWS Access Analyzer)
- CloudTrail / Audit Logs
- Network analyzers (VPC flow logs)

Cluster Layer

- CIS Kubernetes Benchmark tools
- kube-bench, kube-hunter
- RBAC linter tools (rakkess, kubectl-who-can)
- Kubeaudit / Falco rulesets
- Audit log analysis pipelines

Common Tools Used in Kubernetes Assessments

Container Layer

- Trivy, Grype, Clair (image scanning)
- OPA/Gatekeeper policy reports
- Syscalls/runtimes: Falco, Tetragon, Tracee

Code Layer

- SAST, SCA, dependency scanners
- Secret scanners (gitleaks, trufflehog)
- API security testing

What Makes Assessment Challenging in Kubernetes

- Highly dynamic workloads and ephemeral Pods
- Declarative configs with layered inheritance
- Multiple sources of truth (Git, registry, cluster state)
- Shared responsibility across multiple teams
- High privilege components like kubelet and API server
- The attack surface evolves as new CRDs and controllers are added

Assessments must therefore be continuous, automated, and integrated with CI/CD systems.

Prevention

Prevention

Once the security posture of the Kubernetes environment has been assessed, the next step is **Prevention** - implementing controls that limit the attack surface and stop threats before they can reach critical components.

In Kubernetes, prevention is about **designing the system so that unsafe behavior simply cannot occur**, even if:

- a workload is compromised,
- credentials leak,
- or an attacker gains a foothold.

Because Kubernetes is highly dynamic and declarative, preventive measures must be **architectural, policy-driven, and enforced** automatically rather than relying on operators to “remember” best practices.

Prevention is the most important phase of the entire lifecycle because it shapes the cluster’s foundational security guarantees.

Cloud Layer: Restrict the Blast Radius

Key Measures

- **Least-privilege IAM roles** for nodes, controllers, and build systems
- Enforce **no external service account keys**
- Use **private control plane endpoints** when possible
- Block public load balancers unless explicitly required
- Mandatory **encryption at rest** for disks, storage, registries
- **Network segmentation** isolating clusters per environment

Cluster Layer: Control Plane & Node Hardening

Key Cluster and Nodes Prevention Areas

API Server Hardening

- Enforce **authentication and authorization** (RBAC, OIDC, IAM)
- Disable anonymous access
- Enable **audit logging** at the correct verbosity
- Restrict access to sensitive API groups (/logs, /metrics, /proxy)
- Implement **admission controllers** to validate configurations

RBAC Design

- Strict RBAC — developers get namespace-scoped permissions only
- No cluster-admin except for infra/SRE
- Use **RoleBindings**, not ClusterRoleBindings, whenever possible
- Avoid wildcard permissions (verbs: ["*"])

Key Cluster and Nodes Prevention Areas

Node & Kubelet Hardening

- Disable kubelet read-only port
- Enable kubelet authentication & authorization
- Use node OS with a minimized attack surface (COS, Bottlerocket, Flatcar)
- Restrict SSH access to nodes
- Use SELinux/AppArmor at the node level
- Enable kernel lockdown mode (when supported)

etcd Security

- etcd accessible only from API server
- Encrypt Secrets at rest
- Use TLS for all etcd communication
- Regularly rotate certs

Network Prevention Controls

NetworkPolicy

- Enforce default deny for ingress and egress
- Explicitly whitelist namespace-to-namespace traffic
- Restrict access to system namespaces (kube-system, monitoring, etc.)

Pod & Node Boundaries

- Disallow host networking
- Disallow hostPID and hostIPC
- Restrict hostPort usage (rarely required, easy to exploit)

Service Exposure

- Prefer ClusterIP over NodePort / LoadBalancer
- Restrict external access through Ingress with strong TLS
- Use API Gateway / Service Mesh to enforce mTLS

Container Layer: Runtime & Supply Chain Prevention

Supply Chain

- Require signed container images
- Allow only images from trusted registries
- Scan images for vulnerabilities continuously
- Enforce image provenance policy (Kyverno/Gatekeeper)

Runtime Prevention

- Run as non-root
- Disable privilege escalation
- Drop all capabilities except required ones
- Enforce seccomp and AppArmor/SELinux profiles
- Use read-only root filesystems
- For high-risk apps, use gVisor/Kata as an extra sandbox

Resource Boundaries

- Set CPU/memory limits
- Prevent noisy-neighbor attacks
- Prevent node starvation

Code Layer: Application-Level Prevention

Application-Level Prevention

- Input validation and sanitization
- Correct access control in application logic
- Safe secret handling (no secrets in environment variables or code)
- TLS enforcement between services
- Avoid dynamic command execution
- Harden APIs and admin interfaces

Dependency Management

- Remove unused dependencies
- Continuously monitor CVEs in libraries
- Enforce minimal dependency sets

Cross-Layer Preventive Principles

Cross-Layer Preventive Principles

1. Deny-by-Default
   In network access, container capabilities, admission control, and RBAC.

2. Least Privilege
   Every component — cloud IAM, Kubernetes API, workloads — receives only the permissions necessary.

3. Immutability
   Nodes, images, and configurations should be immutable as much as possible.

4. Governance as Code
   Security policies expressed as code:
   - OPA Gatekeeper / Kyverno policies
   - NetworkPolicies
   - RBAC manifests
   - Pod Security Standards
5. Separation of Duties
   Cluster admin ≠ application team.
   Infrastructure ≠ security team.
   CI/CD pipelines ≠ production credentials.

Why Prevention Is Critical in Kubernetes

- Kubernetes is dynamic - new Pods appear constantly, so manual checking is impossible.
- Attack surface is large - API server, kubelet, CRDs, controllers, registries.
- A single misconfiguration (privileged Pod, public NodePort, permissive RBAC) can lead to total compromise.
- Prevention reduces attack vectors before detection or forensics are needed.

In short, prevention makes the cluster **resilient-by-design**.

Detection

Detection

No matter how strong your preventive controls are, **prevention alone is never enough**. Modern Kubernetes environments are too dynamic, too decentralized, and too complex for any preventive configuration to guarantee total safety.

This is where **Detection** comes in.

Detection means **identifying suspicious activity, anomalous behavior, policy violations, or indicators of compromise (IoCs)** inside the cluster as early as possible.

In Kubernetes, detection must operate at multiple layers, across:

- cloud infrastructure,
- control plane,
- nodes,
- network,
- container runtime,
- applications.

Cloud Layer Detection

Cloud Layer Detection

Even if Kubernetes is secure, an attacker may target cloud infrastructure:

- IAM privilege escalation
- Node tampering
- VPC manipulation
- Registry compromise
- Disk snapshot creation (data exfiltration)

Key Detection Mechanisms

- Cloud audit logs (GCP Audit Logs, AWS CloudTrail, Azure Activity Logs)
- Alerts on IAM anomalies (new permissions, excessive access, role chaining)
- Detection of unexpected network configuration changes
- Container registry event monitoring (unexpected image pushes/pulls)
- Node creation/deletion anomalies (potential takeover)

Cluster Layer Detection

Key Cluster Detection Signals

API Server Logs

- The API server is the authoritative source for:
- who did what,
- from where,
- and when.

Critical log types:

- authentication/authorization events
- resource modification events
- failed access attempts
- changes to RBAC, Roles, ClusterRoles
- deployment or DaemonSet creation (often used by attackers)

Key Cluster Detection Signals

Audit Logging

- Audit logs should capture:
- requests to sensitive endpoints (/exec, /proxy, /portforward, /logs)
- modification of secrets
- deletion of workloads
- creation of privileged Pods
- use of service accounts outside their namespace

Kubelet Detection

Kubelet compromise is serious.
Suspicious indicators:

- high rate of exec/session requests
- file access on host paths
- unauthorized TLS connections from kubelet

Key Cluster Detection Signals

Admission Webhook Logs

Both Gatekeeper and Kyverno produce logs:

- denied deployments
- mutated images
- attempted privilege escalation

Repeated denials = attacker reconnaissance or misconfigured CI/CD pipelines.

Container Layer Detection

Node & Container Runtime Detection

Runtime Security Tools

- Falco
- Tetragon (Cilium eBPF)
- Tracee (Aqua Security)
- Sysdig Secure
- Datadog CSM

These tools detect low-level behavior such as:

- unexpected system calls
- modification of sensitive directories
- spawning shells inside containers
- privilege escalation attempts
- access to host namespaces
- node-level file system reads
- crypto-mining patterns
- reverse shells

Node & Container Runtime Detection

Critical Runtime Events

- execve() shells launched inside pods
- changes to /proc, /sys, /etc
- network connections to unapproved destinations
- processes running as root inside containers
- access to Docker socket or CRI socket
- attempts to load kernel modules

Network Detection

Key Network Detection Signals

- unusual namespace-to-namespace connections
- unexpected outbound traffic from pods
- traffic to known malicious IPs
- DNS anomalies
- high-volume traffic spikes
- bypassed or unused NetworkPolicies

Tools

- Cilium Hubble
- Falco (network rules)
- Istio or Linkerd telemetry
- eBPF-based network monitors
- Network flow logs from cloud provider

Goal

Reveal:

Network Detection

Goal

Reveal:

- reconnaissance,
- lateral movement,
- command-and-control traffic,
- and data exfiltration attempts.

Application(code) Layer Detection

Application-Level Detection

Detection at Code Layer

- API gateway logs (unexpected endpoints accessed)
- Application logs with anomalous patterns
- Rate-limiting violations
- Suspicious authentication attempts
- Logic abuse patterns
- Internal admin interfaces accessed externally
- Unexpected DB queries

Tools

- OpenTelemetry combined with SIEM/SOAR
- WAF logs (if Ingress controller supports it)
- API anomaly detection systems (Nginx App Protect, Cloud Armor, etc.)

Cross-Layer Detection Principles

Cross-Layer Detection Principles

- Centralized Logging

  All logs — cloud, node, kubelet, API server, workloads — should aggregate into a central system:

  - Elasticsearch
  - Loki
  - Splunk
  - Cloud-native tools (Cloud Logging, CloudWatch, Sentinel)

Distributed logs → forensic blind spot.

Cross-Layer Detection Principles

- Behavioral Detection Over Signature Detection

  Signatures (CVE matching, known IOCs) are insufficient for Kubernetes because:

  - attacks evolve too quickly,
  - containers are ephemeral,
  - and attackers often use built-in tools (kubectl exec).

  Behavioral detection focuses on:

  - anomaly patterns,
  - resource usage spikes,
  - sudden RBAC changes,
  - unusual node calls,
  - suspicious system calls.

Cross-Layer Detection Principles

- Correlation Between Layers

  A single alert rarely reveals the full attack. Detection becomes powerful when you correlate:

  New IAM role granted → Privileged Pod created → Reverse shell in Pod

  → Outbound traffic spike

Cross-Layer Detection Principles

- Alerting vs. Observability

  Detection is not only about firing alerts.

  It is also about **being able to observe** what is happening:

  - Are Pods being restarted unusually often?
  - Are API requests failing at higher rates?
  - Are deployments being modified unexpectedly?

  Observability is the foundation of incident detection.

Why Detection Is Critical in Kubernetes

Why Detection Is Critical in Kubernetes

Even with perfect prevention:

- A developer may accidentally expose a key.
- An attacker might exploit a zero-day kernel vulnerability.
- A malicious insider might alter RBAC settings.
- A CI pipeline might deploy a tampered container image.

Detection is the only layer that **catches what prevention misses.**

Reaction

Reaction

Detection tells you something is wrong.
Reaction is what you do next — and in Kubernetes, reacting effectively is extremely challenging without prior preparation.

Kubernetes environments are:

- distributed,
- ephemeral,
- declarative,
- fast-moving,
- and often multi-cluster.

If you don't have clearly defined response procedures, a small intrusion can snowball into a cluster-wide compromise.

Reaction is not improvisation.
It is the execution of **pre-planned, automated, and well-rehearsed incident response steps** specific to Kubernetes.

Classes of Attackers

Classes of Attackers

To defend a Kubernetes environment effectively, you must understand who you are defending against. Different attackers have different motivations, levels of sophistication, and preferred techniques. Kubernetes — with its powerful API, distributed components, and rich metadata — is attractive to attackers ranging from low-skill opportunists to highly organized nation-state groups.

This section defines the major attacker classes relevant to cloud-native security and explains how each actor behaves across the 4C security layers (Cloud → Cluster → Container → Code).

Opportunistic Attackers (“Script Kiddies”)

These are low-skill attackers who use automated scanners, public exploits, and botnets to find vulnerable Kubernetes clusters.

Typical Behavior:

- Scan the internet for:
  - exposed dashboards
  - unsecured kubelets
  - unauthenticated API servers
  - open etcd endpoints
  - insecure Ingress controllers
  - Deploy crypto-miners into compromised clusters
  - Use public exploits without deep understanding

Common Targets:

- Public NodePorts
- Misconfigured Ingress
- Open metrics or health endpoints
- Weak container images
- Public registries with poor access control

Cloud Resource Abusers (“Crypto Miners”)

These attackers specifically target cloud-based Kubernetes clusters to consume compute at scale.

Behavior

- Rapid deployment of miners via:
- DaemonSets
- privileged Pods
- CronJobs
- Hide activity through:
- resource limits set to “reasonable” values
- silent network egress
- removal of logs or shell histories

Entry Points

- CI/CD pipeline compromise
- Weak supply chain controls
- Insecure container images
- Misconfigured RBAC allowing Pod creation

Internal Threat Actors

Employees, contractors, or partners who intentionally or unintentionally misuse access.

Two Types

- Malicious insiders: intentional sabotage, data theft
- Accidental insiders: cause harm through mistakes

Behavior

- Abuse RBAC permissions (cluster-admin, secret access)
- Copy sensitive data from internal services
- Reconfigure workloads or policies
- Use kubectl to escalate inside cluster

Supply Chain Attackers

These attackers target CI/CD pipelines, registries, and the build process — not the cluster directly.

Techniques

- Poisoning container images
- Injecting malicious code into dependencies
- Replacing trusted registries with look-alike domains
- Exploiting build agents to steal credentials
- Injecting malicious admission webhooks or CRDs

Targets

- GitHub/GitLab runners
- Container registries
- Build artifacts
- Helm charts and manifests
- Signing keys and provenance systems

Automated Attack Bots

These are large botnets scanning the internet 24/7 for Kubernetes-related weaknesses.

Common Findings They Exploit:

- Public etcd endpoints
- Exposed kubelets (port 10250)
- Open dashboard (cluster-admin by default if unsecured)
- Default credentials in services (Redis, MongoDB)
- SSRF into the Kubernetes API from misconfigured apps

Targeted Intruders (Organized Crime)

More sophisticated attackers who plan their intrusion and adapt to your environment.

Capabilities

- Exploit zero-days
- Move laterally via API server and internal services
- Dump secrets, tokens, and credentials
- Persist through:
  - DaemonSets
  - Mutating webhooks
  - Compromised node bootstrap scripts
  - Hidden CRDs

Goals

- Data theft
- Ransomware
- Industrial espionage
- Control-plane manipulation

Advanced Persistent Threats (APTs / Nation-State Actors)

The highest sophistication level.

Capabilities

- Kernel-level exploits
- Side-channel attacks on cluster nodes
- Breaking out of VMs (hypervisor escapes)
- Zero-day exploitation in Kubernetes components
- Supply chain infiltration (tampering with registry mirrors)
- Attacking cloud metadata services for credential theft

Behavior

- Extremely patient (months-long presence)
- Multi-vector, multi-cluster attacks
- Custom malware targeting Kubernetes components
- Use encrypted and covert communication channels

Types of Attacks

Attacks on the Kubernetes API Server

The API server is the brain of the cluster — compromising it gives full control.

Attack Types

- Unauthorized access due to misconfigured authentication
  (e.g., anonymous auth left enabled)
- RBAC privilege escalation
  Assigning themselves cluster-admin via a vulnerable RoleBinding.
- CRD and webhook abuse
  Attackers deploy malicious admission controllers to mutate workloads.
- “kubectl exec” as a backdoor
  Logs show repeated exec into Pods.
- API server SSRF exploitation
  App calling internal API endpoints through insecure proxies.

Impact

- Full cluster compromise.
- Ability to deploy DaemonSets, modify secrets, exfiltrate data.

Kubelet Attacks

Kubelet has extremely high privileges. If misconfigured, attackers can:

Attack Types

- Access kubelet's unauthenticated API:
  - /exec
  - /run
  - /logs
- Pull container logs containing secrets or tokens.
- Start new privileged Pods attached to the node.
- Read environment variables and secrets from containers.
- Use kubelet credentials to impersonate workloads.

Impact

- Node takeover → control of all Pods scheduled on that node.

Attacks on etcd (Cluster State Store)

Etcd stores:

- Secrets,
- service accounts,
- certificates,
- API objects.

If exposed:

Attack Types

- Dumping all Kubernetes Secrets.
- Modifying cluster state directly (bypass API server).
- Restoring malicious snapshots.
- Changing RBAC bindings at rest.

Impact

- Complete, irreversible cluster compromise.

Attacks via Container Runtime & Node

Attackers often escalate to the node:

Attack Types

- Privileged Pods
  Containers with full host access.
- HostPath mounts
  Attackers mount /var/run/docker.sock, /etc, /proc, or /sys.
- Escaping through container runtime bugs
  runc and containerd vulnerabilities.
- Kernel exploitation
  DirtyPipe, DirtyCOW, OverlayFS bugs.

Impact

Once on the node:

- attackers control kubelet,
- dump credentials,
- pivot to the entire cluster.

Application-Driven Attacks (Code Layer)

Kubernetes cannot protect an insecure app.

Common Exploits

- SSRF into metadata service → credential theft
- RCE inside the container → pivot to cluster
- JWT token theft → impersonate users
- SQL injection leading to secret extraction
- Misconfigured admin endpoints exposed publicly

Impact

Attackers gain valid credentials or run commands inside Pods.

Supply Chain Attacks

Attackers poison the software delivered into the cluster.

Attack Types

- Malicious container images
- Compromised CI/CD pipelines
- Modified Helm charts
- Tampered admission webhooks
- Dependency poisoning (npm/pypi/Go modules)

Impact

- Malicious workloads running under full trust.

Network-Based Attacks

Kubernetes networking is often wide open internally.

Attack Types

- Lateral movement across namespaces
- Accessing metrics servers or internal admin services
- DNS poisoning/abuse inside the cluster
- Exfiltration through unrestricted egress
- ARP spoofing or CNI-specific attacks (Calico, Flannel, etc.)

Impact

- Full east-west compromise between apps.

Identity & Credential Attacks

Kubernetes relies heavily on identity.

Common Techniques

- Stealing service account tokens from Pods
- Using cloud metadata service to fetch IAM credentials
- Reusing kubeconfig files with admin rights
- Extracting secrets from logs or container images
- Leaking CI/CD credentials

Impact

- Attackers gain legitimate access → extremely hard to detect.

Attacks on Ingress / API Gateways

Ingress is the door to the cluster.

Attack Types

- Exploiting the ingress controller itself (Nginx, Traefik, Istio Gateway)
- Exploiting misconfigured TLS (weak ciphers, no client certs)
- Insecure default backends
- Path traversal or rewrite vulnerabilities
- WAF bypass

Impact

- External intrusion into internal services.

Persistence Attacks

Attackers remain inside cluster even after Pod deletion.

Persistence Techniques

- Malicious DaemonSets
- Hidden mutating webhooks
- Fake system Pods in kube-system
- Node compromise with backdoors
- CronJobs redeploying malicious images
- CI/CD pipeline poisoning
- Etcd state modification

Impact

- Attackers return after every restart, upgrade, or redeployment.

Denial of Service (DoS) Attacks

DoS in Kubernetes is particularly damaging because it affects the control plane.

Attack Types

- Overloading API server with large-list/watches
- Creating massive numbers of small objects
- Exhausting etcd storage
- Pod resource exhaustion (rogue Pod consuming all memory)
- Node starvation by scheduling attacker-controlled Pods
- Ingress layer DoS attacks

Impact

- Cluster meltdown or outage.

Data Exfiltration

Common techniques:

- Exfiltrating Secrets via DNS
- Uploading data to attacker-controlled S3/GCS buckets
- Using sidecar containers for covert communication
- Using cloud IAM credentials stolen via metadata service

Impact

- Silent data theft — often the main attacker objective.

Attack Surfaces

Attack Surfaces

An attack surface is any point in a system where an attacker can interact, probe, influence, or exploit the environment.

Kubernetes - with its distributed architecture, declarative APIs, dynamic workloads, and deep cloud integration — exposes **more attack surfaces than traditional infrastructure**.

Understanding these surfaces is crucial because most real-world compromises begin not with advanced exploits but with basic misconfigurations left exposed.

This section maps all major Kubernetes attack surfaces across the 4C Security Model (Cloud → Cluster → Container → Code).

Cloud Layer Attack Surfaces

Cloud IAM

- Over-permissioned service accounts
- Workload Identity / IRSA misconfigurations
- Leaked cloud credentials from Pods

Cloud Networking

- Public load balancers
- Publicly exposed NodePorts
- Misconfigured firewall rules
- Open cloud metadata servers (no IMDSv2, no metadata concealment)

Cloud Storage & Registries

- Public OCI registries
- Public storage buckets containing manifests or Secrets
- Exposed container images (no auth required)

Impact

Cloud-level compromise enables:

- node creation/deletion,
- reading disks containing etcd snapshots,
- disabling firewall boundaries → full cluster compromise.

Cluster Layer Attack Surfaces

Kubernetes API Server

The API server is the single most important attack surface in the entire platform.

Common Exposures

- Exposed API server endpoint (no firewall, no private access)
- Anonymous requests enabled
- Weak authentication (client certs not enforced, weak OIDC setup)
- Default RBAC roles granting wide access
- Misconfigured API aggregation (extension APIs)

Why Dangerous?

- API server compromise = full cluster control.

Etcd

The datastore for Kubernetes — contains:

- Secrets
- tokens
- cluster configuration
- RBAC bindings

Attack Surfaces

- Public etcd endpoint
- etcd without TLS
- No encryption at rest for Secrets
- etcd backups stored unencrypted

Impact

- Reading etcd = reading every Secret in the cluster.

Kubelet

The kubelet manages containers on each node.

Attack Surfaces

- Read-only kubelet API open (port 10255)
- Exec/log/run endpoints exposed (port 10250)
- Unauthorized access to kubelet credentials
- Insecure kubelet configuration flags

Impact

- Kubelet compromise → node compromise → cluster compromise.

Control Plane Components & Admission Webhooks

Attack Surfaces

- Insecure or malicious admission controllers
- Misconfigured CRDs with unsafe schema validation
- Controller-manager or scheduler with overly broad permissions
- Webhooks reachable over HTTP instead of HTTPS

Impact

Attackers can mutate workloads automatically.

Node & Container Runtime Attack Surfaces

Node & Container Runtime Attack Surfaces

Nodes and runtimes represent the infrastructure behind Pods.

Attack Surfaces

- Privileged containers
- Containers with hostPath mounts
- Containers running with all Linux capabilities
- Unrestricted container runtime socket (Docker, containerd, CRI-O)
- Kernel vulnerabilities exploitable from containers
- Malicious DaemonSets

Impact

- Container → node escape is one of the most dangerous paths.

Network Attack Surfaces

Network Attack Surfaces

Kubernetes networking is complex and often misconfigured.

Attack Surfaces

- Lack of NetworkPolicies (default allow-all)
- Publicly exposed Services via LoadBalancer or NodePort
- Ingress controllers open to the world
- Internal east-west traffic unencrypted and unrestricted
- DNS inside the cluster unprotected
- CNI plugin vulnerabilities

Impact

- Lateral movement is easier than most teams realize.

Supply Chain Attack Surfaces

Supply Chain Attack Surfaces

This is one of the fastest-growing areas of Kubernetes-related compromises.

Attack Surfaces

- OCI registries (public pull/push)
- Unverified container images
- Compromised Helm charts, Kustomize bases
- CI/CD pipeline credentials stored insecurely
- GitOps systems with admin privileges
- Admission webhooks pulling remote policy bundles

Impact

- Attacker gets code execution everywhere.

Application (Code) Attack Surfaces

Application (Code) Attack Surfaces

Kubernetes cannot defend against insecure application logic.

Common Surfaces

- Exposed admin APIs
- SSRF into internal Kubernetes services
- Secrets logged accidentally
- Business logic flaws used to pivot into the cluster
- Insecure default credentials
- Unrestricted outbound network access from Pods

Impact

- Application compromise → cloud credential theft → cluster takeover.

Misconfiguration Attack Surfaces

Misconfiguration Attack Surfaces

Misconfiguration is the #1 cause of Kubernetes breaches.

Examples

- Pods running as root
- Pods allowed to mount /var/run/docker.sock
- No limit ranges
- No NetworkPolicies
- Open Ingress routes to sensitive apps
- RBAC roles granting full access
- Admission control disabled

Impact

- Attackers exploit these “low-hanging fruit” first.

Human & Process Attack Surfaces

Human & Process Attack Surfaces

Kubernetes security is also affected by:

- poor secret hygiene
- storing kubeconfig files in GitHub repos
- developers authenticating with admin-level access
- lack of CI/CD isolation
- weak multi-tenancy
- weak review processes for manifests

Impact

Most breaches start with human error.

Inter-Cluster Attack Surfaces

Inter-Cluster Attack Surfaces

Multi-cluster environments introduce new risks:

- Shared CI/CD pipelines with excessive rights
- Shared registries without proper authentication
- Federation or cross-cluster CRDs
- Shared service mesh trust domains
- Peered VPC networks allowing cross-cluster pivoting

Impact

- Attackers compromise one cluster → pivot into all clusters.

Why Attack Surface Analysis Matters in Kubernetes?

- Kubernetes cannot be secured without understanding where the risks originate.
- Many teams only focus on Pods and containers but ignore:
  - API server,
  - cloud layer,
  - CI/CD pipelines,
  - admission controllers,
  - Ingress,
  - and node layers.
- Attackers simply need one weak point to enter.
- Defenders must secure every surface.

Attack surface awareness is the foundation for:

- prevention (what to harden)
- detection (what to monitor)
- incident response (what to isolate first)

Hardware and Firmware Considerations

Hardware and Firmware Considerations

While Kubernetes is often treated as an abstract, cloud-native orchestration layer, it still ultimately runs on **physical hardware and firmware-controlled components.**

These foundational layers — CPUs, memory controllers, device firmware, BIOS/UEFI, hypervisors — are below the cloud, below the OS, and **below the container runtime**.

If an attacker compromises hardware or firmware, every higher layer of the 4C model becomes irrelevant:

Hardware/Firmware compromise → Cloud compromise → Cluster compromise → Container compromise → Code compromise.

This section focuses on the risks, realities, and responsibilities related to hardware and firmware in Kubernetes environments — especially relevant for organizations running:

- on-prem clusters,
- hybrid clusters,
- high-security workloads,
- regulated workloads,
- or bare-metal Kubernetes.

Even in the cloud, understanding these risks helps clarify what is (and isn't) part of the shared responsibility model.

Why Hardware & Firmware Matter

Kubernetes security usually assumes:

- the node OS is trusted,
- the hypervisor is trusted,
- the CPU executes correctly,
- firmware behaves predictably.

Attackers who bypass these layers gain:

- access to every container,
- the ability to read Secrets from memory,
- control over kernel execution,
- persistence that survives reboots, upgrades, and OS reinstalls.

This is what makes hardware-level attacks **strategic**, **rare**, and **high-impact.**

CPU and Hardware-Level Attack Surfaces

Microarchitectural Attacks

Modern CPUs have complex optimizations (speculative execution, branch prediction). These optimizations lead to a class of attacks known as:

- Spectre
- Meltdown
- Foreshadow
- ZombieLoad
- Retbleed
- Downfall

Impact

These vulnerabilities allow attackers to:

- read data from other processes or containers,
- bypass memory isolation,
- steal secrets directly from RAM.

Cloud providers patch microcode and hypervisors frequently — but not instantly, and not for every hardware family.

Firmware (BIOS/UEFI) and Microcode Attacks

Firmware sits below the operating system.

Attack Vectors

- compromised UEFI firmware
- malicious bootloaders
- outdated firmware with exploitable bugs
- firmware implants (persistent malware)
- supply-chain tampering (hardware shipped with modified firmware)

Impact

- survives OS reinstall
- bypasses secure boot
- persistent backdoor with full node control
- ability to manipulate kernel and container runtime

Firmware-level persistence is exceptionally hard to remove.

Hardware Root of Trust & Secure Boot

Secure Boot

- Ensures only signed, trusted boot components load.
- Prevents:
- malicious bootloaders
- tampered kernels
- early-stage rootkits

TPM / vTPM

- Trusted Platform Module provides:
- measured boot
- cryptographic attestation
- hardware-stored keys

Hypervisor Security (Cloud Environments)

Even in managed Kubernetes (GKE, EKS, AKS), the hypervisor remains an attack surface.

Risks

- hypervisor escape vulnerabilities
- co-tenant side-channel attacks
- compromised underlying hardware in the data center
- malicious insiders at cloud provider

Impact

- An attacker escaping the hypervisor can:
- read memory of all tenants
- control VMs hosting Kubernetes nodes
- dump kubelet credentials
- tamper with node OS images

Cloud providers mitigate these risks with:

- hardened hypervisors
- microcode updates
- platform-level attestation
- isolated compute pools (confidential VMs)

On-Prem Kubernetes Hardware Risks

Organizations running bare-metal Kubernetes face additional risks.

Attack Surfaces

- IPMI/BMC interfaces exposed
- Outdated BIOS/UEFI firmware
- Lack of hardware attestation
- Physical access (USB implants, hardware keyloggers)
- Supply-chain tampering in delivered servers

BMC (Baseboard Management Controller)

- BMC vulnerabilities (OpenBMC, iDRAC, iLO) can:
- bypass OS security
- inject malicious firmware
- control power state
- read/write system memory

GPU, NIC, and Peripheral Firmware

Each device often runs its own firmware.

Firmware Targets

- NIC firmware
- GPU firmware
- SSD controller firmware
- RAID card firmware
- SmartNIC/DPUs running Linux kernels

Risks

- hidden persistence
- unauthorized DMA access
- memory scraping
- evading OS-level logging
- firmware-level backdoors

Supply-Chain Risks at the Hardware Level

Sophisticated attackers may compromise hardware before it reaches the data center.

Risks

- malicious components inserted in transit
- compromised firmware shipped from factory
- tampered hardware in the vendor supply chain

High-profile examples:

- GrayFish, MATA malware families
- BMC implants
- Supermicro allegations
- nation-state firmware implants

Shared Responsibility Model for Hardware & Firmware

Cloud Providers Handle:

- hypervisor security
- hardware patching
- CPU microcode updates
- physical data center protection
- supply-chain validation
- hardware replacement
- firmware patch rollout

You Handle:

- selecting node families
- enabling secure boot (if optional)
- configuring confidential nodes/VMs
- validating OS images
- preventing kubelet privilege escalation
- isolating workloads

Bare-Metal Environments:

- You handle everything.

What Kubernetes Administrators Must Do

Even though some responsibilities fall on cloud providers, Kubernetes operators must still:

Ensure Node Integrity

- choose hardened OS images
- enable secure boot wherever available
- enforce measured boot & attestation
- patch kernels promptly

Reduce Node Attack Surface

- disable unnecessary drivers
- disable unused kernel modules
- minimize host utilities
- use distroless or slim OS images (COS, Bottlerocket, Flatcar)

What Kubernetes Administrators Must Do

Detect Hardware/Firmware Anomalies

- monitor for unexpected reboots
- monitor for hardware events (disk resets, NIC resets)
- watch for unusual DMA or PCI activity (via eBPF tools)

Isolate High-Risk Workloads

- use separate node pools
- use gVisor/Kata for untrusted workloads
- avoid GPU sharing for mixed-trust tenants

Use Confidential Computing Where Available

(e.g., GKE Confidential Nodes, AWS Nitro Enclaves)

Security Agencies

Security Agencies

Kubernetes security does not exist in isolation.

A wide ecosystem of organizations, agencies, foundations, and research groups continuously produces standards, best practices, threat analyses, guidelines, and hardening recommendations.

These entities are crucial because:

- Kubernetes evolves rapidly,
- attack techniques evolve even faster,
- cloud-native environments span multiple industries and compliance regimes.

The goal of this section is to explain **who** influences cloud-native security, **what** they publish, and **why** their work is essential for designing and maintaining hardened Kubernetes environments.

CNCF TAG Security

Role

- The most important community-driven body for cloud-native security.

Contributions

- publishes major whitepapers (e.g., Cloud Native Security Whitepaper, Software Supply Chain Security Paper)
- maintains security reviews of CNCF projects
- develops best practices & threat modeling guides
- evaluates tools (OPA, Kyverno, Falco, Cilium, etc.)

Kubernetes SIG Security

Role

- Security-focused governance body within the Kubernetes project itself.

Contributions

- responsible for Kubernetes security processes
- owns the Kubernetes Threat Model
- manages vulnerability reports (CVEs)
- participates in Kubernetes releases and hardening features
- publishes security guidance, KEPs (enhancements)

CIS (Center for Internet Security)

Role

Creates prescriptive hardening benchmarks for systems and platforms.

Key Asset

- CIS Kubernetes Benchmark
  (Industry standard for cluster audits)

CIS Benchmark Covers

- API server flags
- etcd configuration
- kubelet settings
- RBAC rules
- node hardening
- logging & auditing
- restricted policies

NIST (National Institute of Standards and Technology)

Role

Publishes U.S. federal cybersecurity guidelines.

Key Kubernetes Documents

- NIST SP 800-190: Application Container Security Guide
- NIST SP 800-204A/B/C: Cloud-Native Application Security
- NIST Zero Trust Architecture

NSA & CISA (US National Security Agency & Cybersecurity and Infrastructure Security Agency)

Role

- Publish security guidance for critical infrastructure.

Key Document

- NSA + CISA Kubernetes Hardening Guide

Content Highlights

- disable anonymous kube-apiserver access
- enforce RBAC least privilege
- restrict kubelet access
- apply NetworkPolicies everywhere
- use strong TLS & mTLS
- restrict NodePorts
- use separate node pools
- ensure Secrets encryption at rest
- harden container runtime

MITRE ATT&CK Framework (Containers & Cloud)

Role

Provides a globally-recognized framework of attacker tactics and techniques.

Relevant Matrices

- MITRE ATT&CK for Containers
- MITRE ATT&CK for Cloud (AWS/GCP/Azure)

OWASP (Open Web Application Security Project)

Role

Focuses on application-level vulnerabilities.

Relevant Projects

- OWASP Top 10
- OWASP Kubernetes Top 10
- OWASP API Security Top 10
- OWASP Dependency-Check

Security Vendors & Research Labs

Examples

- Aqua Security (Tracee, Trivy)
- Sysdig (Falco)
- Palo Alto Prisma
- Datadog
- Wiz
- CrowdStrike
- NCC Group
- IBM X-Force

Why Understanding These Agencies Matters

1. **They Define Industry Standards**

   CIS Benchmarks, NSA/CISA guides, NIST frameworks → widely used for compliance and audits.

2. **They Shape Kubernetes Evolution**

   SIG Security and TAG Security influence how Kubernetes improves over time.

3. **They Provide Threat Intelligence**

   MITRE, vendors, and federal agencies track real attacker behavior.

4. **They Provide Hardening & Operational Guidance**

   SRE and DevOps teams rely on proven recommendations rather than guesswork.

5. **They Influence Tools & Best Practices**

   Many security tools in Kubernetes follow:

   - NIST principles
   - CIS standards
   - MITRE attacker techniques

Manage External Access

External Access to the Kubernetes API Server

Prefer Private API Endpoints

- Use private API access (cloud-managed networks).
- Restrict API endpoint to corporate VPN or VPC.
- Never expose kube-apiserver to the public internet unless absolutely required.

Strong Authentication

- Use OIDC or cloud IAM identities.
- Disable anonymous authentication.
- Remove legacy authentication methods (static passwords, tokens).

Limit the CIDR Range

Allow API access only from:

- corporate IP ranges,
- jump hosts,
- VPN gateways.

Audit the API Continuously

Log all requests.
Alert on:

- high-volume list/watch requests
- unauthorized attempts
- use of privileged verbs (create, delete, exec, proxy)

External Access to Workloads (Ingress)

TLS Everywhere

- Enforce TLS on all Ingress endpoints.
- Use Let’s Encrypt or internal CA with proper rotation.
- Mandate strong cipher suites.

Require Authentication for Sensitive Endpoints

Ingress controllers support:

- mTLS
- external authentication (OIDC)
- custom auth middlewares

Restrict Public Exposure

Only expose necessary services.

Prefer:

- ClusterIP
- Internal LoadBalancers
- Private Ingress classes

L7 Protection

Enable:

- WAF rules
- rate limiting
- header validation
- request size limits

Ingress Policies for Zero Trust

Use policies to enforce:

- no unencrypted backends
- no wildcard hosts
- no plain HTTP

NodePorts and LoadBalancers

NodePort

Never use NodePort in production environments, unless:

- you enforce firewall rules,
- strict CIDR whitelisting,
- and node isolation.

LoadBalancer Services

- Use internal load balancers (private IP mode).
- Restrict allowed source ranges.
- Avoid exposing Pods directly.
- Monitor LB creation events in the cloud (unintentional exposure is common).

External Access via Egress (Outbound Traffic)

External Access via Egress (Outbound Traffic)

- Enforce egress NetworkPolicies (default deny).
- Use egress gateways (Istio, Cilium) for internet-bound traffic.
- Restrict DNS resolution to internal resolvers.
- Block access to cloud metadata endpoints unless explicitly allowed.
- Monitor DNS logs and egress flows.

Multi-Cluster & Cross-Environment External Access

Multi-Cluster & Cross-Environment External Access

- Separate trust domains per cluster.
- Use cluster-local identities for mTLS.
- Avoid shared service accounts across clusters.
- Validate traffic between clusters using:
  - mTLS
  - authorization policies
  - peer validation

External Access via CI/CD and GitOps

External Access via CI/CD and GitOps

- CI/CD should only have namespace-scoped permissions.
- GitOps controllers should use read-only Git deploy keys.
- Validate signed manifests (Sigstore, Cosign).
- Block public webhooks unless behind authenticated tunnels.

Preparing to Install

- Image Supply Chain
- Runtime Sandbox
- Verify Platform Binaries
- Minimize Access to GUI
- Policy Based Control

Image Supply Chain

Image Supply Chain

The image supply chain is one of the most critical components of Kubernetes security.
Every workload — including system components, sidecars, CI/CD artifacts, controllers, operators, and application Pods — ultimately starts with a **container image.**

If an attacker can influence:

- the base image,
- the build process,
- the registry,
- or the deployment pipeline,

they can introduce **malicious code inside** your cluster before the Pod even runs.

Kubernetes itself **does not verify image trust by default.**
This means securing the supply chain must be done before installation and before enabling workloads.

Why Image Supply Chain Security is Foundational

- Containers are immutable artifacts → a compromised image spreads everywhere.
- Registries are global shared resources → a single compromise affects all clusters.
- CI/CD systems often run with privileged access → perfect targets for attackers.
- Kubernetes downloads images automatically → no manual inspection opportunity.

Core Principles of Image Supply Chain Security

Trust the Source

Only pull images from:

- trusted private registries,
- approved upstream sources,
- vendor-provided verified images.

Trust the Build

Build images using:

- isolated CI/CD runners,
- ephemeral build agents,
- defined build recipes (Dockerfiles, Buildpacks, Bazel, Nix),
- reproducible builds where possible.

Trust the Artifact

Images should be:

- signed,
- versioned,
- scanned,
- verified before running in the cluster.

Trust the Deployment

Admission controllers must enforce:

- “only signed images,”
- “only images from approved registries,”
- “no mutable tags (‘latest’)”.

Key Components of Secure Image Supply Chains

- Image Provenance and SLSA Framework
- Image Signing (Cosign / Sigstore)
- Image Scanning
- Base Image Selection and Hardening
- Controlling Registries
- Enforcing Immutable Tags

Kubernetes Controls for Image Supply Chain

Admission Controllers

Used to enforce:

- signed images only
- registry allowlisting
- block latest tags
- block privileged containers
- enforce SBOM requirements

Tools:

- Kyverno
- OPA Gatekeeper
- Kubewarden

Pod Security Admission

Blocks known risky behaviors:

- privilege escalation
- hostPath mounts
- running as root

Runtime Enforcement

Tools like:

- Falco,
- Tetragon,
- Tracee,

monitor image execution behavior.

SBOM (Software Bill of Materials)

Mandatory for regulated environments.

Tools:

- Syft
- Anchore - Grype
- CycloneDX

Runtime Sandbox

Runtime Sandbox

Even if your container images are trusted and your supply chain is secure, you still can't trust the workloads themselves.

Applications contain bugs, dependencies contain vulnerabilities, and attackers may exploit runtime behavior after container start-up.

In Kubernetes, the runtime sandbox provides an additional layer of isolation between:

- workloads and the node,
- workloads and each other,
- workloads and the kernel,
- and in some cases even workloads and the container runtime.

Types of Runtime Sandboxes

Types of Runtime Sandboxes

- seccomp
- AppArmor / SELinux

Strong Sandbox via Virtualization (gVisor, Kata Containers)

gVisor (Google)

A “kernel in userspace” that intercepts syscalls and handles them in a sandbox.

Mechanism

- Containers run in a user-space kernel
- Host kernel sees only sanitized operations
- Limits kernel attack surface

Pros

- Very strong isolation
- Compatible with many workloads
- Excellent for multi-tenancy

Cons

- Not suitable for all apps (e.g., high-perf workloads)
- Slight performance overhead

When to Use

- Untrusted code
- Multi-tenant SaaS
- Public workloads
- CI/CD runners

Kata Containers (OpenStack / Intel)

Lightweight micro-VMs for each Pod.

Mechanism

- Each Pod runs inside a dedicated micro-VM
- Strongest kernel and process isolation

Pros

- VM-grade isolation
- Best defense against container escapes

Cons

- Higher resource overhead
- Slower startup time
- Needs node types with virtualization support

When to Use

- Strict multi-tenancy (financial, regulated, hostile code)
- Security-critical workloads
- Hosting customer-provided code

Verify Platform Binaries

Verify Platform Binaries

Before installing Kubernetes — before the API server is deployed, before kubelet starts, before containerd runs — you must ensure the underlying binaries and components are authentic, verified, and untampered.

Kubernetes is a distributed system composed of:

- control plane binaries (kube-apiserver, kube-scheduler, kube-controller-manager),
- node binaries (kubelet, kube-proxy),
- container runtimes (containerd, CRI-O, runc),
- system utilities, kernel modules, CNI plugins,
- and third-party controllers and add-ons.

What Needs to Be Verified

- Kubernetes Binaries
- Container Runtime
- CNI Plugins
- CSI Drivers
- Node OS Components
- Add-On Controllers

How to Verify Kubernetes Binaries

- Signature Verification
  - hashes (SHA256)
  - PGP signatures
- Vendor Distribution Integrity
  - runtime version consistency
  - signed node images
- Container-Based Component Verification

Minimize Access to GUI

Minimize Access to GUI

Kubernetes is designed to be managed via API-driven workflows, not GUIs.

While command-line tools (kubectl), GitOps pipelines, and automation provide secure, auditable, and minimal-privilege access, GUI-based tools — especially the Kubernetes Dashboard — introduce significant security risks.

This subsection explains:

- what Kubernetes Dashboard is,
- why it is a high-risk component,
- how attackers exploit it,
- and how it must be secured if used at all.

What Is Kubernetes Dashboard?

The Kubernetes Dashboard is a web-based UI for managing the cluster.

It allows users to:

- view workloads, nodes, ConfigMaps, Secrets, logs
- edit deployments and Pods
- delete or scale workloads
- deploy new applications
- manage RBAC and resources
- access logs and metrics

Why Kubernetes Dashboard Is Dangerous

Historical Default Misconfigurations

The Dashboard has a notorious history of:

- shipping insecure RBAC defaults
- using service accounts with cluster-admin privileges
- exposing internal APIs unintentionally
- running without proper authentication

Attackers frequently scan for exposed dashboards.

Dashboard Has Full Read Access to Cluster Metadata

Even with restricted privileges, Dashboard exposes:

- Pod logs (may contain secrets)
- environment variables (tokens, credentials)
- image names, internal endpoints
- namespace relationships
- cluster topology
- node names and internal IPs

This is high-value reconnaissance for attackers.

Dashboard Is Often Exposed Publicly

Teams frequently expose Dashboard using:

- NodePort
- LoadBalancer
- Insecure reverse proxy

A single misconfigured ingress → full cluster takeover.

Dashboard Encourages Dangerous Operational Patterns

GUI-based cluster management leads to:

- manual changes,
- no auditability,
- privilege escalation via convenience,
- bypassing GitOps,
- configuration drift.

This conflicts with all Kubernetes security principles:

- immutability
- declarative state
- automated control loops

How to Secure Dashboard

How to Secure Dashboard

- Never Expose Dashboard Publicly
- Remove Privileged Service Accounts
- Use SSO/OIDC Authentication
- Protect Dashboard With Network Controls
- Audit Dashboard Access
- Use Time-Limited Access Methods

Safer Alternatives to Dashboard

- kubectl with RBAC
- GitOps dashboards (ArgoCD, FluxUI)
- observability dashboards (Grafana, Kiali, Lens local client)
- custom internal platform UIs with strong gateway authentication

Policy-Based Control

Policy-Based Control

Policy-based control ensures that only secure, compliant, and approved workloads are allowed into the cluster.
Without policies, Kubernetes will deploy anything the API server accepts.
This means developers, CI/CD pipelines, or compromised credentials can deploy:

- privileged Pods,
- containers running as root,
- untrusted images,
- workloads with host access,
- insecure configurations,
- or malicious components.

Policies must be defined before the first workload arrives, as part of the installation preparation.
Trying to “bolt on” governance later is painful and often requires breaking existing workloads.

Why Policy-Based Control Is Foundational

Kubernetes does not enforce secure defaults

By design, Kubernetes is flexible:

- Pods can run as root
- Privileged mode is allowed
- hostPath mounts are allowed
- Containers can execute arbitrary syscalls
- Any image can be pulled
- Mutable tags are accepted
- Secrets can be mounted anywhere

CI/CD mistakes propagate instantly

Without policy:

- a misconfigured CI pipeline can deploy privileged Pods,
- a typo can accidentally expose a service to the internet,
- a wrong image tag can bypass signing,
- insecure manifests can land in production instantly.

Policies enable multi-tenancy

You cannot safely host multiple teams, environments, or tenants without:

- namespace guardrails,
- resource quotas,
- Pod Security enforcement,
- image restrictions,
- network segmentation.

Policies enforce compliance

Regulated workloads (PCI-DSS, GDPR, HIPAA, ISO 27001, SOC2) require:

- non-root containers
- restricted capabilities
- network segmentation
- mandatory logging
- encrypted traffic
- signed artifacts
- prevention of insecure settings

Core Categories of Kubernetes Policies

Pod Security Policies / Pod Security Standards (PSS)

Pod Security Admission (PSA) enforces baseline, restricted, privileged policy levels.

Restricted Mode Blocks:

- privileged: true
- running as root
- host namespaces
- hostPath mounts
- CAP_SYS_ADMIN and other dangerous capabilities
- unsafe syscalls
- unsafe volume types
- unsafe Linux security options

Admission Control Policies

Use admission controllers to inspect, mutate, or reject workloads.

Common Implementations

- OPA Gatekeeper (Rego policies)
- Kyverno (policy-as-yaml)
- Kubewarden (WASM policies)

Policy Examples

- Only signed images allowed
- Only images from private registries
- No :latest tags
- No privileged containers
- Enforce resource limits/requests
- Enforce seccomp/AppArmor profiles
- Require labels/annotations (owner, team, env)
- Block host networking

Network Policies

Control Pod-to-Pod and Pod-to-external communication.

Capabilities

- isolate namespaces
- enforce microsegmentation
- block lateral movement
- restrict egress
- enforce zero-trust architecture

Image Policies

Policies that enforce secure supply chain use.

Key Controls

- signed images only
- approved registries
- immutable digests
- enforced SBOM presence
- vulnerability thresholds (fail if critical CVEs)

Tools

- Kyverno image verify
- Cosign policy
- Admission control
- OPA

Resource Governance Policies

These protect cluster stability and prevent noisy-neighbor scenarios.

Examples

- resource quotas per namespace
- mandatory CPU/memory requests
- limit ranges
- pod disruption budgets
- anti-affinity and topology spreads (for HA)

Runtime Security Policies

Using tools like:

- Falco
- Tetragon
- Tracee
- Sysdig Secure

Runtime policies enforce:

- allowed syscalls
- allowed processes
- allowed networking actions
- expected runtime behavior

Installing the Cluster

- Update Kubernetes
- Tools to Harden the Kernel
- Kernel Hardening Examples
- Mitigating Kernel Vulnerabilities

Update Kubernetes

Update Kubernetes

Keeping Kubernetes up to date is a primary security requirement, not an operational convenience.

Kubernetes is a fast-moving project with a ~4-month release cycle, frequent security patches, constant API deprecations, and regular updates to:

- the API server,
- kubelet,
- controller-manager,
- scheduler,
- container runtime,
- CNI / CSI plugins,
- and core system add-ons.

Running outdated Kubernetes versions is the single most common cause of cluster compromises, second only to misconfigurations.

This section explains why Kubernetes must be updated consistently, how versioning works, what security patches mean, and how upgrades affect cluster security.

Upgrading Kubernetes Securely

Control Plane First

Upgrade order:

1. kube-apiserver
2. controller-manager + scheduler
3. etcd (if separate upgrade path)
4. kube-proxy
5. kubelets

Node Pools Must Be Upgraded

Leaving old kubelets in place is a common mistake.

Old kubelets =

- missing seccomp enforcement
- incomplete PodSecurity behavior
- outdated TLS config
- vulnerable container runtime
- outdated CNI plugins

Nodes must be drained, patched, and rotated.

Use Surge Upgrades, Not In-Place Mutations

Use:

- GKE surge upgrades
- EKS managed node groups
- AKS auto-upgrade infrastructure
- kOps rolling upgrades
- Cluster API rolling updates

Never manually upgrade nodes in-place unless required.

Rolling replacements ensure:

- fresh kernel
- fresh runtime
- fresh node configuration

Validate Policies After Upgrade

Policy engines (Kyverno, Gatekeeper) must be upgraded too.

After cluster upgrade:

- run conformance tests
- validate PSA modes
- test admission policies
- test networking policies
- test TLS configs

Security controls often change subtly between versions.

Risks of Not Updating Kubernetes

- Known CVEs remain unpatched
- Pod Security features missing
- Incompatibility with secure admission controls
- Deprecated APIs cause breakage
- No vendor support
- Node compromise becomes easier

Tools to Harden the Kernel

Tools to Harden the Kernel

The Linux kernel is the true security boundary for container workloads.

All Pods on a node share the same kernel, and a kernel exploit means:

- container escape,
- node compromise,
- kubelet compromise,
- and ultimately cluster compromise.

Because Kubernetes depends entirely on kernel isolation features (namespaces, cgroups, capabilities, seccomp, LSMs), hardening the kernel is mandatory during cluster installation — not something you add later.

Why Kernel Hardening Matters in Kubernetes

Containers share the host kernel

Kernel CVEs are frequent

Privileged Pods bypass all user-space protections

Kernel choices must be made before installation

Primary Kernel Hardening Tools

- seccomp (System Call Filtering)
- AppArmor/SELinux (Mandatory Access Control)
- Linux Capabilities
- Kernel Lockdown Mode
- Disabling Unused Kernel Modules
- eBPF Restrictions
- cgroup Hardening

Kernel Hardening Starts Before kubelet Runs

Most of these protections require:

- systemd unit modifications
- kernel boot arguments
- LSM ordering
- runtime settings baked into node images
- containerd/cri-o integration

Common Hardening Misconfigurations

- leaving unprivileged eBPF enabled
- not enforcing seccomp profiles
- not enabling SELinux or AppArmor
- running privileged Pods
- allowing hostPath mounts
- using generic Ubuntu/Red Hat images with full kernels
- nodes built with GUI dependencies
- running containerd with default (permissive) seccomp

Kernel Hardening Examples

Kernel Hardening Examples

Kernel hardening is not theoretical — it must be implemented concretely, with specific configurations applied at install time.
This section provides real-world examples of kernel hardening used in production Kubernetes environments across cloud, hybrid, and on-prem installations.
These examples illustrate how OS configuration, LSM systems, kernel flags, cgroup settings, and runtime restrictions combine to create a hardened node baseline.

The examples are grouped into:

- Minimalist cloud images
- Hardened enterprise distributions
- Security-focused custom builds
- LSM enforcement patterns
- Safe defaults for container runtimes

This gives participants practical reference patterns they can use to build secure Kubernetes node images.

Hardened On-Prem Nodes

Kernel Settings

- AppArmor enforced
- seccomp profile runtime/default applied cluster-wide
- Dangerous syscalls blocked (keyctl, bpf, mount)
- kernel lockdown in “integrity” mode

Hardened On-Prem Nodes

Host OS Hardening

- SSH disabled or limited to bastion
- password authentication disabled
- fs.protected_fifos=2, fs.protected_hardlinks=1

Disabling Unprivileged eBPF

Multiple recent container escape CVEs exploited:

- unprivileged_bpf
- raw packet generation
- JIT compilation of malicious BPF programs

Kernel Lockdown Mode

- blocks /dev/mem
- blocks kernel module loading
- prevents writing to kernel memory
- restricts kexec
- enforces signed kernel modules

Minimal Kernel Modules

The kernel should include only modules required by:

- networking stack
- storage device drivers
- cgroup controllers
- container runtime

Modules commonly disabled:

- USB subsystems
- sound drivers
- virtualization drivers (if not needed)
- firewire
- legacy network protocols (DECnet, IPX)
- unnecessary filesystems (hfs, jfs, minix)

Mitigating Kernel Vulnerabilities

Disable Unprivileged eBPF and User Namespaces

Modify via sysctl:

/etc/sysctl.d/99-kubernetes-hardening.conf:

```
kernel.unprivileged_bpf_disabled = 1

kernel.unprivileged_userns_clone = 0

kernel.kptr_restrict = 2

kernel.dmesg_restrict = 1

fs.protected_hardlinks = 1

fs.protected_symlinks = 1
```

Disable Module Loading

Add to GRUB:

/etc/default/grub:

```
GRUB_CMDLINE_LINUX="... module.sig_enforce=1 modules_disabled=1"
```

Blacklist Allowed Modules

Create:

/etc/modprobe.d/blacklist.conf:

```
blacklist usb_storage
blacklist firewire_core
blacklist thunderbolt
blacklist bluetooth
blacklist soundcore
blacklist floppy
blacklist nfs
blacklist cifs
blacklist jffs2
```

Enable Kernel Lockdown Mode

Edit GRUB:

/etc/default/grub:

```
GRUB_CMDLINE_LINUX="... lockdown=integrity"
```

Securing the kube-apiserver

- Restrict Access to API
- Enable Kube-apiserver Auditing
- Configuring RBAC
- Pod Security Policies
- Minimize IAM Roles
- Protecting etcd
- CIS Benchmark
- Using Service Accounts

Restrict Access to API

Restrict Access to API

The kube-apiserver is the single most valuable target in a Kubernetes cluster.

Every security mechanism — RBAC, admission control, Pod Security, Secrets, certificate signing, service account token validation — depends entirely on the API server functioning correctly and being accessed only by trusted clients.

If the API server is exposed or misconfigured, attackers can:

- bypass authentication,
- escalate privileges,
- access or modify Secrets,
- deploy privileged workloads,
- impersonate nodes,
- extract service account tokens,
- pivot to etcd,
- join rogue nodes to the cluster.

Therefore, the first step in securing the API is restricting access to it — both network access and logical access.

Enable Kube-apiserver Auditing

Enable Kube-apiserver Auditing

Audit logging is one of the most powerful and most underused Kubernetes security features.

Without audit logs, you are effectively blind:

- You cannot see who accessed Secrets.
- You cannot track privilege escalation.
- You cannot detect token abuse.
- You cannot reconstruct incidents.
- You cannot prove compliance.
- You cannot detect attackers before it’s too late.

The API server is the control point for the entire cluster — so every meaningful security event flows through it.

Proper auditing transforms the API server into a full forensic and monitoring system.

How Audit Logging Works in Kubernetes

Audit Policy File

Defines what categories of API requests are logged and at what level:

- None
- Metadata
- Request
- RequestResponse

Audit Backend

Defines where logs go:

- filesystem
- webhook
- log collector

Enable Audit Logging — API Server Configuration

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml
--audit-log-path=/var/log/kubernetes/audit.log
--audit-log-maxage=30
--audit-log-maxsize=100
--audit-log-maxbackup=10
```

Example Audit Policy

```yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:

# Log all requests to Secrets (full content for writes)
- level: RequestResponse
 resources:
   - group: ""
     resources: ["secrets"]

# Log access to pods/exec and pods/attach
- level: Request
 verbs: ["create"]
 resources:
   - group: ""
     resources: ["pods/exec", "pods/attach"]

# Log any changes to RBAC
- level: Request
 resources:
   - group: "rbac.authorization.k8s.io"
     resources: ["roles", "rolebindings", "clusterroles",
"clusterrolebindings"]

# Log new Pods created (especially important for privilege checks)
- level: Metadata
 resources:
   - group: ""
     resources: ["pods"]

# Default rule: log metadata for everything else
- level: Metadata
 omitStages:
   - "RequestReceived"
```

What You MUST Log

- Access to Secrets
- Privilege Escalation Attempts
- Execution Inside Pods
- Authentication Failures
- Node Join / Certificate Signing Requests
- Mutating Webhooks Failing
- Unexpected high-volume access patterns

Protect Audit Logs

Audit logs themselves must be protected because they contain:

- Secrets
- tokens
- sensitive object bodies
- full HTTP request content
- user identities
- bearer tokens in headers
- kubeconfig content for users

Configuring RBAC

Configuring RBAC

Role-Based Access Control (RBAC) is the primary authorization mechanism in Kubernetes.

It determines who can do what in the cluster.

If RBAC is misconfigured:

- developers accidentally become cluster-admin
- service accounts gain permissions they should not have
- CI/CD systems escalate privileges
- attackers move laterally using stolen tokens
- Pods gain the ability to modify workloads in other namespaces
- secrets leakage becomes trivial
- privilege escalation chains appear everywhere

RBAC mistakes are among the top 3 causes of Kubernetes breaches.

Guiding Principles of Secure RBAC

- Least Privilege
  - Every identity gets only the minimum permissions needed.
- Separation of Duties
  - Split roles between:
    - developers
    - platform engineers
    - SRE
    - security
    - CI/CD
    - automation
- Namespace Isolation
  - RBAC boundaries must match namespace boundaries.
- No Shared Credentials
  - No shared kubeconfigs, no shared service accounts.
- Human vs Machine Separation
  - Humans authenticate via OIDC → RBAC roles.
  - Machines authenticate via service accounts → restricted roles.
- Default Deny
  - No permissions unless granted explicitly.

The Four RBAC Object Types

RBAC is built from 4 primitives:

- Role – namespace-scoped permissions
- ClusterRole – cluster-wide permissions
- RoleBinding – binds users/groups/SAs to Roles in namespace
- ClusterRoleBinding – binds users/groups/SAs to ClusterRoles globally

Pod Security Policies / Pod Security Admission

Pod Security Policies / Pod Security Admission

Pod-level security controls are one of the most misunderstood — and most misconfigured — parts of Kubernetes security.

Historically, Kubernetes used PodSecurityPolicy (PSP) to enforce security rules on Pods. PSP was complex, confusing, and allowed dangerous bypasses. It was deprecated in 1.21 and removed in 1.25.

Today, the correct mechanism is Pod Security Admission (PSA) — a built-in admission controller that enforces three standardized profiles:

- Privileged
- Baseline
- Restricted

PSA provides a simple, predictable, standardized way to restrict Pod permissions and eliminate dangerous patterns, such as privileged containers, hostPath volumes, unsafe capabilities, or host networking.

Why Pod Security Still Matters (Even with RBAC)

Even if RBAC is perfectly configured, an attacker with permission to create Pods can:

- schedule privileged Pods
- mount the host filesystem
- access hostPath volumes
- use hostNetwork to bypass network policies
- escalate via capabilities
- exploit container runtime vulnerabilities
- break out through unconfined LSM profiles

PSA prevents this by blocking Pods that violate security rules.

RBAC controls who can create Pods, PSA controls what Pods they are allowed to create.

Pod Security Admission Profiles

Privileged

Allows all Pod capabilities.

Equivalent to disabling pod-level security.

Use cases:

- kube-system components requiring special access
- low-level system daemons

Should not be used for application workloads.

Baseline

Blocks known high-risk privileges but allows many common patterns.

Disallows:

- privileged containers
- hostPID, hostIPC
- hostNetwork
- dangerous volume types (hostPath, device)
- adding new capabilities except safe ones

Allows:

- container ports
- read-only root filesystems (optional)
- basic runtime features

Use case:

- general-purpose namespaces
- development namespaces

Restricted

Enforces strict, minimal security requirements.

Requires:

- non-root user
- no privilege escalation
- read-only root filesystem (recommended)
- safe capabilities only (NET_BIND_SERVICE)
- seccomp profile set to “RuntimeDefault”
- AppArmor or SELinux enforcing
- disallows emptyDir writable unless needed

Use case:

- production workloads
- multi-tenant systems
- regulated environments
- zero-trust workloads

Enforcing PSA in Namespaces

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

kubectl label namespace dev pod-security.kubernetes.io/audit=baseline

kubectl label namespace staging pod-security.kubernetes.io/warn=restricted
```

Why PSP Failed and PSA Succeeded

PSP problems:

- too complex
- API allowed unexpected bypasses
- cluster-wide, not namespace-scoped
- required cluster-admin to modify
- inconsistent behavior across versions
- removed from Kubernetes entirely

PSA improvements:

- simple labels
- clear behavior
- predictable policy
- align with security standards (CIS, NSA, NIST)
- namespace scoped
- no PSP bypasses
- easy to audit cluster security posture

Minimize IAM Roles

Minimize IAM Roles

In Kubernetes, “IAM” refers to identity and access management for every actor interacting with the cluster:

- Human users (OIDC)
- Service Accounts (workloads)
- Nodes (kubelet identities)
- External systems (CI/CD, GitOps, automation)
- Cloud provider identities (if running on cloud)

IAM in Kubernetes is not just about cloud permissions — it is also about:

- how ServiceAccounts are used,
- how tokens are issued and rotated,
- how RBAC binds permissions,
- how workload identities are scoped,
- how access is delegated between systems.

Protecting etcd

Protecting etcd

etcd is the single most sensitive component in the entire Kubernetes ecosystem.

It stores all cluster data, including:

- Secrets (in plaintext unless encryption-at-rest is enabled)
- Service account tokens
- ConfigMaps
- Pod specifications
- RBAC configuration
- Admission configuration
- API server state
- Node credentials
- Every object stored in the Kubernetes cluster

Why etcd Security Is Critical

Everything the cluster does flows through etcd.

If etcd is compromised, an attacker can:

- read all Secrets
- grant themselves cluster-admin via RBAC edits
- schedule privileged Pods
- modify controller configuration
- remove audit policies
- steal service account tokens
- tamper with admission webhooks
- rewrite CRDs
- poison cluster state
- delete everything instantly

Key etcd Security Principles

- Isolation — etcd must never be reachable outside a secure control-plane network.
- TLS everywhere — client and peer communication must use mutual TLS.
- Encryption at Rest — Secrets must be encrypted using KMS or envelope encryption.
- Compartmentalization — only API server should access etcd.
- Auditing and hardening — file system security, permissions, SELinux/AppArmor, systemd protection.
- No co-location with workloads — etcd must not run on worker nodes.

Configure Mutual TLS Correctly

etcd configuration:

```
--cert-file=/etc/kubernetes/pki/etcd/server.crt
--key-file=/etc/kubernetes/pki/etcd/server.key
--client-cert-auth=true
--trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

Peer (etcd-to-etcd) communication:

```
--peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
--peer-key-file=/etc/kubernetes/pki/etcd/peer.key
--peer-client-cert-auth=true
--peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

API server TLS config (client side):

```
--etcd-servers=https://etcd-0:2379,https://etcd-1:2379
--etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
--etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
--etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
```

CIS Benchmark

CIS Benchmark

The CIS (Center for Internet Security) Kubernetes Benchmark is the most widely adopted security standard for Kubernetes.

It provides a comprehensive checklist of configuration controls for:

- kube-apiserver
- kube-controller-manager
- kube-scheduler
- kubelet
- etcd
- worker nodes
- network configuration
- Pod security settings
- RBAC configuration

What the CIS Benchmark Covers (High-Level)

- Control Plane Components
- Control Plane — etcd
- Control Plane — Configuration Files
- Worker Node
- Policies

Networking

- Firewalling Basics
- Network Plugins
- iptables
- Mitigate Brute Force Login Attempts
- Netfilter rule management
- Netfilter Implementation
- nft Concepts
- Ingress Objects
- Pod to Pod Encryption

Networking

Kubernetes networking is one of the most critical and misunderstood security domains.

Modern clusters have:

- Pod-to-Pod communication
- Node-to-Pod
- Node-to-API-server
- Ingress/Egress
- Service networking
- Overlay networks (CNI plugins)
- Sidecar communication
- Metrics and webhook pathways
- Control-plane and data-plane separation

Because Kubernetes networking is extremely permissive by default, it creates multiple attack surfaces

Firewalling Basics

Firewall Basics

When Kubernetes runs on virtual machines (bare-metal hypervisors, VMware, OpenStack, KVM, cloud-like private IaaS), network security starts outside the node — at the virtualization layer.

In this model, the VM's virtual NIC, hypervisor-level firewall, and virtual network segmentation are more important than firewalls inside the OS.

This is how production clusters actually isolate:

- control-plane VMs
- worker VMs
- ingress VMs
- load balancers
- management VMs (bastion, logging, monitoring)

VM networking ≠ Pod networking

VM NICS control traffic to and from nodes

Examples:

- API server VM
- etcd VM
- worker VMs
- load balancer VM
- bastion VM

Pod networking happens inside the VM, through:

- CNI plugins
- overlay networks
- routing tables

Therefore:

VM firewalls control macro-segmentation (node-to-node, API-to-node, external ingress).
NetworkPolicies control micro-segmentation (Pod-to-Pod).

Never mix the two.

Required VM Firewall Rules (The Minimum Matrix)

Management Network → Control Plane

Allow:

- SSH (if used)
- API server (6443)
- metrics/monitoring ports as needed
- no direct access to worker VMs

Control Plane VMs → Worker VMs

Allow:

- kubelet API (10250)
- node heartbeat
- overlay network traffic (VXLAN, IPIP, etc.)

Block:

- control-plane → NodePort range (usually unnecessary)
- direct access from worker to scheduler/CM

Worker VMs → Control Plane

Workers need:

- HTTPS to API server (6443 only)

Block:

- worker → etcd (2379–2380)
- worker → controller-manager (10252)
- worker → scheduler (10251)
- worker → anything not required

Worker VMs → Internet

Restrict to:

- container registries
- explicit endpoints (webhooks, external APIs)
- OS package mirrors (if necessary)

Prefer:

- egress gateways
- NAT gateways
- firewall rules per VM subnet

Node-to-Node Traffic

CNI	VM-Level Ports Required
Flannel VXLAN	UDP 8472
Calico VXLAN	UDP 4789
Calico IPIP	Protocol 4
Cilium	BPF-native (minimal), possibly UDP 8472
Weave	TCP/UDP 6783, 6784

Ingress / Load Balancer VMs

Allow:

- ports 80/443 inbound
- load balancer → worker NodePort or backend Pod ports
- no ingress→control-plane except 6443 if you expose the API behind LB (rare in on-prem)

VM Firewall Principles for Kubernetes

- Always isolate the control plane from workers
- Only API server should talk to etcd
- Worker VMs should not accept traffic from other tenants
- Restrict NodePort exposure
- Protect VM → Host Hypervisor Path

VM Firewall Anti-Patterns

Using flat Layer 2 networks for all nodes - Results in lateral movement between nodes.

Allowing workers to reach etcd - Full compromise risk.

Allowing developers to reach kubelets on VMs - Leads directly to container breakout or secrets theft.

Not separating admin traffic from data-plane - Admin traffic becomes sniffable inside overlay networks.

Exposing API server directly on VM public interface - Common but extremely dangerous.

Allowing unrestricted outbound internet access from worker VMs -Enables malware distribution and data exfiltration.

VM-Level Firewalling + Pod-Level NetworkPolicies = Complete Model

- VM-Level Firewalls
  - macro segmentation
  - restrict node-to-node, worker-to-control-plane
  - limit access to critical ports (API, etcd)
- Kubernetes NetworkPolicies
  - micro segmentation
  - Pod-to-Pod
  - namespace isolation
- CNI Security Features
  - encryption (WireGuard/IPsec)
  - eBPF enforcement
  - anti-spoofing

Network Plugins (CNI)

Network Plugins (CNI)

In Kubernetes, Pod networking is not implemented by Kubernetes itself.
Instead, it is delegated to Container Network Interface (CNI) plugins.

The CNI plugin controls:

- Pod IP assignment
- Pod-to-Pod routing
- node-to-Pod routing
- overlay or underlay networks
- NAT or no-NAT behavior
- IPAM (IP management)
- network policy enforcement (if supported)
- encryption (if supported)
- BPF/iptables rules applied to Pods
- MTU and encapsulation logic

What CNI Actually Does

When a Pod starts, the CNI plugin:

1. Assigns an IP to the Pod
2. Creates network namespace
3. Creates veth interfaces
4. Configures routes
5. Adds firewall rules (iptables/nft/BPF based)
6. Applies network policy rules (if supported)
7. Integrates with control-plane events (if required)

Without CNI, **Pods have no networking at all**.

Not All CNIs Support Network Policies

NetworkPolicy is a Kubernetes API.
Enforcing it is the CNI plugin's responsibility.

CNIs that support NetworkPolicies:

- Calico
- Cilium
- Weave Net
- Kube-Router
- Antrea
- OVN-Kubernetes
- Canal (Calico + Flannel hybrid)

If your CNI doesn't support NetworkPolicies:

- Kubernetes remains a flat, fully open L2/L3 network
- Pod-to-Pod isolation is impossible
- Zero-trust networking cannot be implemented

Quick Overview: What Are NetworkPolicies?

NetworkPolicies are layer 3/4 firewall rules for Pods.

They allow you to define:

- which Pods can talk to which Pods
- which namespaces are allowed
- which IP blocks are allowed
- which ports and protocols
- ingress and egress rules

They do NOT handle:

- L7 filtering
- hostNetwork Pods
- node traffic
- external traffic to Services
- DNS rules (unless explicit IP match)

CNI Features That Matter for Security

- NetworkPolicy enforcement
- Pod-to-Pod encryption
- Anti-spoofing and identity validation
- eBPF datapath
- Host endpoint enforcement

Security Pitfalls When Choosing a CNI

Using Flannel in production - No isolation, no encryption, no policy enforcement.

Mixing multiple CNIs - Breaks routing and security controls.

Assuming NetworkPolicies work everywhere - They only work if CNI supports them.

Using default CNI configuration - CNIs often allow all traffic unless policies are enforced.

Not enabling encryption - Internal Pod traffic can be intercepted in some VLAN/virtualized environments.

iptables

iptables

Even though Kubernetes networking feels abstracted, underneath the cluster the majority of CNIs still rely heavily on iptables to implement:

- Pod-to-Pod routing
- NAT rules for Services (ClusterIP, NodePort, ExternalIP)
- DNAT for kube-proxy
- SNAT for egress traffic
- NetworkPolicy enforcement (for iptables-based CNIs)
- Masquerading and overlay encapsulation logic

iptables in Kubernetes: What It Actually Does

- Service NAT (kube-proxy)
- Pod Egress Traffic (SNAT/Masquerade)
- NetworkPolicy Enforcement (for certain CNIs)
- Overlay Networking
- Node Local Rules

Key iptables Tables Used by Kubernetes

nat Table

Handles NAT for:

- ClusterIP
- NodePort
- ExternalIP
- Pod SNAT

Critical chains:

- KUBE-SERVICES
- KUBE-NODEPORTS
- KUBE-MARK-MASQ
- KUBE-POSTROUTING
- KUBE-PORTALS-HOST

filter Table

Handles firewall packet filtering.

Used by:

- CNIs for NetworkPolicies
- host firewall rules
- anti-spoofing

Critical chains:

- KUBE-FORWARD
- KUBE-POD-FW-* (inserted by CNIs)

mangle Table

Used occasionally for marking traffic.

Some CNIs use it for:

- QoS marking
- routing decisions
- anti-spoofing

Common Issues Caused by iptables

Conflicting host firewall rules - Host UFW/firewalld rules override or collide with CNI rules.

iptables rule explosion - Large clusters have thousands of rules → slow sync → dropped packets.

Node reboot or kubelet restart resets chains - CNI may rebuild rules incorrectly or partially.

Overlay encapsulation NAT conflicts - Misconfigured iptables NAT interferes with VXLAN/IPIP.

MTU problems - iptables-based overlays + encapsulation + NAT → fragmentation issues.

NetworkPolicies not applying - CNI does not enforce policies correctly, often due to broken iptables chains.

iptables and Security Implications

A compromised node can modify iptables

- attacker can bypass NetworkPolicies
- inject malicious rules
- expose Pods
- intercept traffic
- redirect Services

Future: Moving Away from iptables

Modern CNIs increasingly replace iptables with eBPF due to:

- performance
- reliability
- simplified datapath
- more precise policy enforcement
- avoiding thousands of NAT rules

Mitigate Brute Force Login Attempts

Mitigate Brute Force Login Attempts

Even though Kubernetes abstracts away most infrastructure, the VMs hosting control-plane and worker nodes remain critical assets.

If attackers brute-force their way into a VM:

- they gain access to kubelet credentials
- can read ServiceAccount tokens
- can modify iptables
- can fetch secrets from local disk
- can pivot into the control-plane network
- can compromise the entire cluster

SSH Is the Primary Brute-Force Target in Kubernetes Infrastructures

In enterprise clusters, every node VM (worker + control-plane) exposes SSH via:

- the management network
- the hypervisor network
- a bastion host
- or an internal admin subnet

Brute-force attacks typically originate from:

- the internet (if VM reachable)
- internal network scanners
- compromised Pods pivoting outward
- misconfigured jump hosts
- compromised employee laptops

The Golden Rule: Nodes Should Never Be Directly SSH-Accessible

Never allow:

- SSH from employee laptops directly to nodes
- SSH open to entire internal network
- SSH open to external networks
- SSH on public IPs attached to node VMs
- password authentication

SSH should be reachable ONLY from a dedicated bastion host.

Mandatory SSH Hardening Settings (VM OS Level)

- Disable password authentication
  - `PasswordAuthentication no`
  - `ChallengeResponseAuthentication no`
  - `KbdInteractiveAuthentication no`
- Disable root SSH login
  - `PermitRootLogin no`
- Allow only specific users
- Change default shell for system accounts
- Use SSH key-based authentication ONLY

Rate Limiting and Intrusion Detection on SSH

- Rate-limit connections
  - sshguard/fail2ban

```
bantime = 1h
findtime = 1m
maxretry = 3
```

- Detect brute-force attempts
  - repeated failed login attempts
  - source IP concentration
  - connections per minute

Netfilter Rule Management

Why Netfilter Matters in Kubernetes

Every Kubernetes node VM relies on Linux netfilter to implement:

- Service NAT (ClusterIP, NodePort, ExternalIP)
- Pod egress SNAT
- Pod ingress DNAT
- NetworkPolicy enforcement (for iptables-based CNIs)
- Overlay encapsulation (VXLAN/IPIP/IPsec)
- Anti-spoofing and identity validation
- Routing decisions for Pod ↔ Pod and Pod ↔ Node traffic

In practice, multiple system components write to the same netfilter tables, often simultaneously.

Sources of Netfilter Rules on Kubernetes Nodes

- **kube-proxy** - Services NAT, NodePort routing, load balancing
- **CNI plugin** - Pod networking, overlay routing, anti-spoofing
- **NetworkPolicies** - Implemented via iptables chains in many CNIs
- **Host firewall** - UFW / firewalld / nftables rules
- **Security tooling** - IDS/IPS agents, monitoring agents
- **Automation** - Cloud-init, Ansible, Puppet
- **Kernel defaults** - conntrack, rpfilter, sysctl-driven behaviors

Common Problems in Netfilter Rule Management

- **Rule order conflicts** - one chain overrides another, breaking Services or Pod routing.
- **Host firewall interference** - UFW or firewalld dropping traffic the CNI expects.
- **iptables-legacy vs iptables-nft mismatch** - double or inconsistent tables.
- **Rule drift** - leftover chains after CNI/kube-proxy restart.
- **Conntrack exhaustion** - massive Service NAT usage causing packet drops.
- **Node compromise** - attacker rewrites netfilter rules, bypassing all Pod isolation.

Best Practices for Managing Netfilter on Kubernetes Nodes

- Do not manually modify iptables rules on nodes - use hypervisor or VM firewall instead.
- Prefer eBPF-based CNIs to reduce iptables dependency (Cilium, Calico eBPF).
- Avoid using UFW/firewalld on nodes unless carefully configured not to override CNI chains.
- Use ephemeral nodes (immutable infrastructure) to avoid rule drift.
- Ensure consistent iptables mode (legacy OR nft, never mixed).
- Monitor chain health (Calico/Felix, Cilium agent, kubelet logs).

Netfilter Implementation

Netfilter Hooks: The Core Processing Points

Packets traverse netfilter via hooks:
`PREROUTING → INPUT → FORWARD → OUTPUT → POSTROUTING`

Kubernetes uses these hooks as follows:

- **PREROUTING** - kube-proxy DNAT for Services
- **FORWARD** - Pod-to-Pod routing & NetworkPolicy enforcement
- **POSTROUTING** - SNAT/MASQUERADE for Pod egress
- **INPUT** - traffic destined for the node VM itself
- **OUTPUT** - traffic generated by the node

Key iptables Chains Used by Kubernetes

nat table

- KUBE-SERVICES
- KUBE-NODEPORTS
- KUBE-SEP-* (per-endpoint rules)
- KUBE-SVC-* (load-balancing chains)
- KUBE-MARK-MASQ

filter table

- KUBE-FORWARD
- KUBE-POD-FW-* (NetworkPolicies)
- cali-* (Calico)
- WEAVE-* (Weave)

NAT in Kubernetes

Kubernetes uses NAT extensively:

- **DNAT** - ClusterIP, NodePort, ExternalIP → Pod IP
- **SNAT** - Pod egress to external network
- **MASQUERADE** - Overlay networks, Pod → Node
- **NAT in CNI** - VXLAN/IPIP encapsulation adjustments

Identity & Anti-Spoofing in Netfilter

CNIs use netfilter to enforce:

- IP address validation
- MAC spoofing prevention
- Source IP verification (reverse-path filtering)
- Pod identity enforcement

nft Concepts

nftables: The Modern Interface to Netfilter

nftables is the successor to iptables:

- atomic updates
- cleaner rule structure
- higher performance
- native sets/maps
- unified interface to all netfilter hooks

Many Linux distributions now use iptables-nft under the hood.

nftables and Kubernetes: Real-World Implications

Kubernetes itself does not yet natively use nftables.
CNIs and kube-proxy still interact with netfilter primarily via iptables.

Problems appear when mixing:

- iptables-legacy
- iptables-nft
- nftables rules

This can lead to:

- duplicate tables
- inconsistent chain behavior
- broken CNI behavior
- NAT rules not applied properly

Clusters must choose one mode consistently.

Core nftables Concepts (Mapped to Kubernetes Use Cases)

nftables introduces:

- **Tables** - Equivalent to iptables “filter/nat/mangle”.
- **Chains** - Equivalent to iptables chains but more flexible.
- **Sets** - Huge advantage over iptables.

Example:
Instead of 1,000 DNAT rules for endpoints:

```
set pod_endpoints {
    type ipv4_addr;
    elements = { 10.0.1.5, 10.0.1.6, 10.0.1.7 }
}
```

nftables Migration in Kubernetes Ecosystems

Current patterns:

- Cilium (eBPF) → bypasses both iptables and nftables for datapath
- Calico eBPF → minimal iptables usage
- Traditional Calico/Weave/Kube-Router → require iptables backing
- Managed Kubernetes distributions → may rely on nft backend implicitly

In VM-based clusters, nftables becomes important primarily for:

- host-level firewalls
- hypervisor-integrated security
- IDS/IPS deployments
- reducing rule complexity

Recommended nft Practices for Kubernetes

1. Do not mix iptables-legacy and iptables-nft across nodes.
2. If OS defaults to nftables, validate the CNI's compatibility first.
3. Prefer Cilium/Calico eBPF where possible.
4. Use nftables sets for VM-level firewalling for scalability.
5. Do not manually insert nft rules on worker nodes unless isolated from CNI chains.

Ingress Objects

Ingress Objects

Ingress is the primary mechanism for handling HTTP(S) traffic entering a Kubernetes cluster.
Where a Service or NodePort exposes a workload at the network level, Ingress provides L7 routing, TLS termination, and advanced traffic management.

However, Ingress is also one of the largest attack surfaces in a Kubernetes environment, because:

- it is directly exposed to external clients
- it terminates TLS (sensitive certificate handling)
- it routes requests to internal Services
- it is often misconfigured to allow unintended access paths
- its controller runs with elevated privileges
- it may be used as an unintentional pivot point if isolation is weak

What Ingress Actually Does

Ingress defines L7 routing rules, such as:

- host-based routing (api.company.com → api-service)
- path-based routing (/payments → payments-service)
- TLS certificate bindings
- rewrites, redirects
- rate limits, request size limits
- auth layers (OIDC, BasicAuth, mTLS, OAuth2 proxies)

Ingress resources themselves do nothing until an Ingress Controller interprets them.

Ingress Controllers: Security Characteristics

Ingress Controllers: Security Characteristics

Not all Ingress controllers are equal.

They vary significantly in:

- privilege level
- resource consumption
- isolation features
- default security posture
- protocol support
- request inspection

NGINX Ingress Controller

Strengths:

- most widely used
- massive ecosystem and docs
- mature
- lots of security annotations (WAF, rate-limits, CORS headers)

Risks:

- single point of external exposure
- controls TLS certs
- runs as privileged DaemonSet in many installations
- misconfigurations are common

HAProxy Ingress

Strengths:

- efficient and fast
- strong L7 security features
- very stable configuration model

Risks:

- fewer users = fewer eyes
- still a central chokepoint if compromised

Traefik

Strengths:

- modern, dynamic config
- good middleware model
- built-in mTLS and auth features

Risks:

- flexible config can lead to accidental exposure
- extensive annotation usage → security drifts

Envoy-based / API Gateway style

Strengths:

- very strong L7 filters (WAF, authentication, routing logic)
- Envoy is hardened and well-audited
- supports advanced identity and mTLS
- deep request inspection

Risks:

- significantly more complex
- misconfiguration can expose internal services without clear visibility

Ingress Attack Surface & Risks

- Misconfigured TLS
- Open path routing
- Lack of authentication
- Excess privileges
- NodePort exposure
- L7 vulnerabilities

Hardening Ingress

- Strict TLS Configuration
- Enforce Authentication at the Edge
- Use a Default Deny Ingress Policy
- Harden RBAC for Ingress Controller Service Account
- Isolate Ingress Controller Pods
- Rate Limiting and Request Filtering

Pod-to-Pod Encryption

Pod-to-Pod Encryption

By default, Kubernetes does not encrypt Pod-to-Pod traffic.

All east–west communication inside the cluster is:

- unencrypted
- unauthenticated
- often transmitted over shared virtual networks
- sometimes visible to other tenants in multi-tenant environments
- sometimes passing through physical networks (depending on overlay/underlay design)

This means anyone who gains access to:

- the worker node VM
- a compromised Pod
- the underlying virtual switch
- the hypervisor network
- misconfigured VLANs

Why Pod-to-Pod Traffic Is Not Secure by Default

Inside a Kubernetes cluster, traffic flows through:

- virtual Ethernet pairs
- bridges
- overlay tunnels (VXLAN/IPIP)
- routing tables
- the host's netfilter subsystem
- node-to-node connections

All of this traffic is:

- plaintext
- not integrity-protected
- not authenticated

Because Kubernetes treats the cluster network as a trusted internal network, which is no longer acceptable in modern security environments.

Attack Surface Without Encryption

- A compromised node can sniff Pod traffic
- Hypervisor-level sniffing
- Multi-tenant clusters
- Pod escapes / privileged Pods
- Hybrid environments

Workload Considerations

- Minimize Base Image
- Static Analysis of Workloads
- Runtime Analysis of Workloads
- Container Immutability
- Mandatory Access Control
- SELinux
- AppArmor
- Generate AppArmor Profiles

Workload Considerations

This chapter focuses on securing the workloads themselves, not the cluster infrastructure.

While previous sections covered:

- cluster installation
- kernel hardening
- API server security
- networking security

this chapter shifts the focus to containers, Pods, and application-level security inside Kubernetes.

Minimize Base Image

Why Minimal Images Matter

- Reduces attack surface
- Limits available binaries for attackers
- Minimizes dependencies and CVEs
- Smaller images → faster CI/CD and fewer supply-chain risks
- Encourages predictable runtime behavior

Preferred Base Image Types

- **Scratch**: zero userspace, ideal for static Go binaries
- **Distroless**: no shell, package manager, or utilities
- **Alpine**: minimal, but beware musl/glibc compatibility
- **Vendor minimal images** (ubi-micro, wolfi, slim variants)

Anti-Patterns

- Using ubuntu, debian, centos for apps
- Leaving package manager and shell inside (apt, apk, bash)
- RUN curl ... | bash during build
- Running as root by default

Static Analysis of Workloads

What Is Static Analysis

- Scanning images before deployment
- Detecting CVEs, malware, secrets, misconfigurations
- Examining Dockerfile for bad patterns
- Auditing SBOM, dependencies, libraries

Tools & Ecosystem

- Trivy, grype, clair, anchore, syft
- CI pipeline integration
- Auto-fail builds on high-severity CVEs
- Signed SBOM required for regulated workloads

What to Scan

- Base image layers
- Language dependencies (npm, pip, go mod)
- Hardcoded secrets
- Embedded private keys
- Misconfigured permissions
- Unpinned versions

Output & Gatekeeping

- Severity thresholds
- Allow-list only when justified
- Enforce signing (Sigstore, Cosign)
- Store reports in artifact repositories

Runtime Analysis of Workloads

Why Analyze at Runtime

- Detect unexpected syscalls
- Identify anomalous behavior (crypto miners, reverse shells)
- Identify privilege escalation attempts
- Detect malware / rootkits inside containers

Runtime Tools

- Falco, Tetragon, Tracee, KubeArmor
- Audit events (exec, network calls, file writes)
- Behavior baselining with ML (advanced setups)

Runtime Events of Interest

- Shell spawning inside containers
- Writing to system paths (/proc, /sys)
- Outbound connections to unknown destinations
- Privilege escalation attempts
- Unexpected file modifications

Enforcement Models

- Detect-only
- Detect + alert
- Prevent (block syscalls)
- Prevent + quarantine Pod

Container Immutability

What Immutability Means

- Containers are not patched in place
- No changes to filesystem after build
- No interactive sessions for “fixing” containers
- All updates = new image build

Benefits

- Reproducibility
- Predictable behavior
- Reduced drift and debugging overhead
- Simplifies compliance audits
- Prevents malicious modifications in runtime

How to Enforce

- Read-only root filesystem
- Disable shell access
- Disable package managers
- Use GitOps to control image rollout
- Block kubectl exec in production

Mandatory Access Control

Components

- AppArmor (profile-based restrictions)
- SELinux (context-based labeling)
- Seccomp (syscall whitelisting/blacklisting)

Why MAC Matters

- Limits blast radius when a container is compromised
- Prevents privilege escalation
- Controls access to host resources
- Blocks unsafe syscalls (namespace breakout attempts)

Common MAC Misconfigurations

- Containers running with “unconfined” profiles
- Seccomp disabled by default in many clusters
- HostPath mounts bypassing MAC
- Privileged containers nullifying MAC

SELinux

SELinux Basics

Label-based enforcement

Types: enforcing, permissive, disabled

Uses context on processes, files, sockets

SELinux in Kubernetes

- Required in OpenShift
- Supported in most major distros
- Enforces Pod isolation at the host level
- Policies applied via context (spc, system_u, container_t)

Benefits

- Strong defense against host escape
- Granular enforcement
- Mandatory isolation regardless of container privileges

Challenges

- Complex to write custom policies
- Limited adoption outside Red Hat ecosystems
- Incorrect labeling breaks apps unexpectedly
- Requires SELinux-aware storage drivers

AppArmor

AppArmor

- Profile-based MAC system
- Easier than SELinux
- Controls file, network, and syscall access
- Required on many hardened Kubernetes deployments

Typical Profile Structure

- Define allowed file paths
- Allowed capabilities
- Network access rules
- Ingress/egress syscall restrictions

Benefits

- Easy to audit
- Simple syntax
- Deployable via annotations
- Excellent compromise detection

Limitations

- Less granular than SELinux
- Not supported on all node OS types
- Needs manual profile tuning for complex apps

Generate AppArmor Profiles

Why Generate Profiles

Default profiles often too permissive

Workloads differ in runtime needs

Prevent lateral movement

Prevent unauthorized filesystem and syscall access

How to Generate

- Capture behavior using tools:
  - aa-genprof
  - aa-logprof
  - Tracee/Tetragon syscalls
  - auditd logs
- Convert observed behavior into a minimal policy

Best Practices

- Start in permissive mode → observe → tighten
- Keep separate profiles per workload type
- Re-generate regularly after app updates
- Validate with CI before deployment

Issue Detection

- Understanding Phases of Attack
- Preparation
- Understanding an Attack Progression
- During an Incident
- Handling Incident Aftermath
- Intrusion Detection Systems
- Threat Detection
- Behavioral Analytics

Understanding Phases of Attack

The Kill Chain Model

- Reconnaissance
- Initial access
- Privilege escalation
- Lateral movement
- Persistence
- Impact / exfiltration

Relevance to Kubernetes

- Weak workload or image → initial entry
- Misconfigured RBAC → privilege escalation
- Flat network or no NetworkPolicies → lateral movement
- Access to node → cluster takeover
- Poor monitoring → late detection

Cluster-Specific Attack Paths

- Exploiting exposed Services
- Compromised Ingress/Ingress Controller
- Exploiting container runtime
- Abusing service account tokens
- Accessing node metadata services

What Detection Must Cover

- Malicious Pod behavior
- Unexpected system calls
- Strange DNS or outbound traffic
- New privileged workloads
- API server misuse or anomalies

Preparation

Prerequisites for Effective Detection

- Clear visibility into cluster components
- Unified logging (API, kubelet, runtime)
- Centralized alerting
- Defined retention + SIEM integration
- Baseline of normal workload behavior

Instrumentation Needed

- API audit logs
- Container runtime logs
- Node OS logs (journald / syslog)
- eBPF monitors
- Network flow logs
- Admission controller webhook logs

Pre-Attack Readiness

- Predefined incident playbooks
- Known escalation paths
- Reliable alert routing
- Access to forensics tools
- Immutable logs for compliance

Understanding an Attack Progression

Step 1: Initial Access

- Vulnerable image or library
- Exposed workload endpoints
- Misconfigured Ingress
- Exploited API server routes

Step 2: Privilege Escalation

- Running as root
- Lack of seccomp/AppArmor
- Privileged Pods
- Access to node filesystem
- Misconfigured PodSecurity

Step 3: Lateral Movement

- No NetworkPolicies
- Weak identity boundaries
- Using mounted tokens
- Reaching kubelet ports
- Abuse of cloud provider metadata

Step 4: Impact

- Data exfiltration
- Persistent backdoors
- Node compromise
- API server impersonation
- Destructive actions / ransomware

During an Incident

Immediate Goals

- Contain
- Identify
- Isolate
- Preserve artifacts
- Avoid accidental destruction of evidence

Immediate Technical Actions

- Quarantine workloads
- Block node egress
- Snapshot node VMs (if on-prem)
- Revoke compromised tokens
- Kill suspicious connections (conntrack)

Kubernetes-Specific Actions

- Capture Pod / node logs
- Dump API audit logs
- Retrieve Pod specs from etcd
- Save node filesystem mounts
- Export network flow logs (CNI)

Short-Term Mitigation

- Scale down compromised workloads
- Disable external ingress routes
- Restrict RBAC
- Temporarily enforce stricter PodSecurity

Communication & Coordination

- Notify SRE/platform teams
- Inform security teams
- Document timeline
- Maintain evidence chain-of-custody
- Avoid premature cleanup

Handling Incident Aftermath

Key Objectives

- Assess root cause
- Evaluate blast radius
- Identify compromised nodes, tokens, or workloads
- Determine data exposure / regulatory impact
- Produce remediation plan

Kubernetes-Specific Postmortem Tasks

- Audit service account usage
- Rotate cluster and node-level credentials
- Rebuild compromised nodes from clean images
- Patch workloads and dependencies
- Reinforce PodSecurity and RBAC

Long-Term Improvements

- Enforce signing (images, SBOMs, manifests)
- Add runtime enforcement (seccomp, AppArmor, SELinux)
- Tighten network segmentation
- Introduce admission controls for risky Pods

Intrusion Detection Systems

IDS in Kubernetes Context

- Detect malicious Pod activity
- Monitor system calls and kernel behavior
- Observe node-level anomalies
- Act as early-warning system

Common Kubernetes IDS Tools

- Falco (rule-based syscall detection)
- Tetragon (eBPF-based policy enforcement)
- Tracee (eBPF event tracing)
- KubeArmor (MAC-based enforcement + detection)

What IDS Should Detect

- Reverse shells
- Privilege escalation attempts
- File tampering
- Unexpected network connections
- Lateral movement attempts
- Suspicious process execution

IDS Placement Requirements

- Installed as DaemonSets
- Access to kernel events
- Low overhead monitoring
- Immutable rulesets and tamper-proof logs

Threat Detection

Threat Sources

- Compromised Pods
- Insecure images
- Insider threats
- Supply chain attacks
- Container runtime vulnerabilities
- Node-level breaches

Detection Techniques

- API server anomaly detection
- Log analysis and correlation
- Container behavior analytics
- DNS anomaly detection
- Process tree analysis
- eBPF event tracing

Indicators of Compromise

- Sudden elevated privileges
- Unexpected process trees
- Failed authentication bursts
- Abnormal outbound traffic
- Workloads spawning shells
- Missing network policies

Threat Categorization Models

- MITRE ATT&CK for Containers
- Kubernetes Threat Matrix
- CIS Benchmark alignment
- Kill-chain mapping

Behavioral Analytics

Purpose

Build baseline of normal behavior

Detect deviations that signature-based IDS will miss

Identify zero-days and unknown threats

Catch insider threats and slow anomalies

Techniques

- Time-series analysis
- Machine learning models
- Auto-profiling of workloads
- Correlation across logs, metrics, and traces
- Outlier detection on network patterns

Limitations

- Requires tuning to avoid false positives
- ML models may not understand Kubernetes semantics
- Overhead in large clusters
- Behavioral drift needs recalibration

TEF

https://sube.nobleprog.com/pl/open-tef/39955

Kontakt

https://www.linkedin.com/in/michal-rudz/