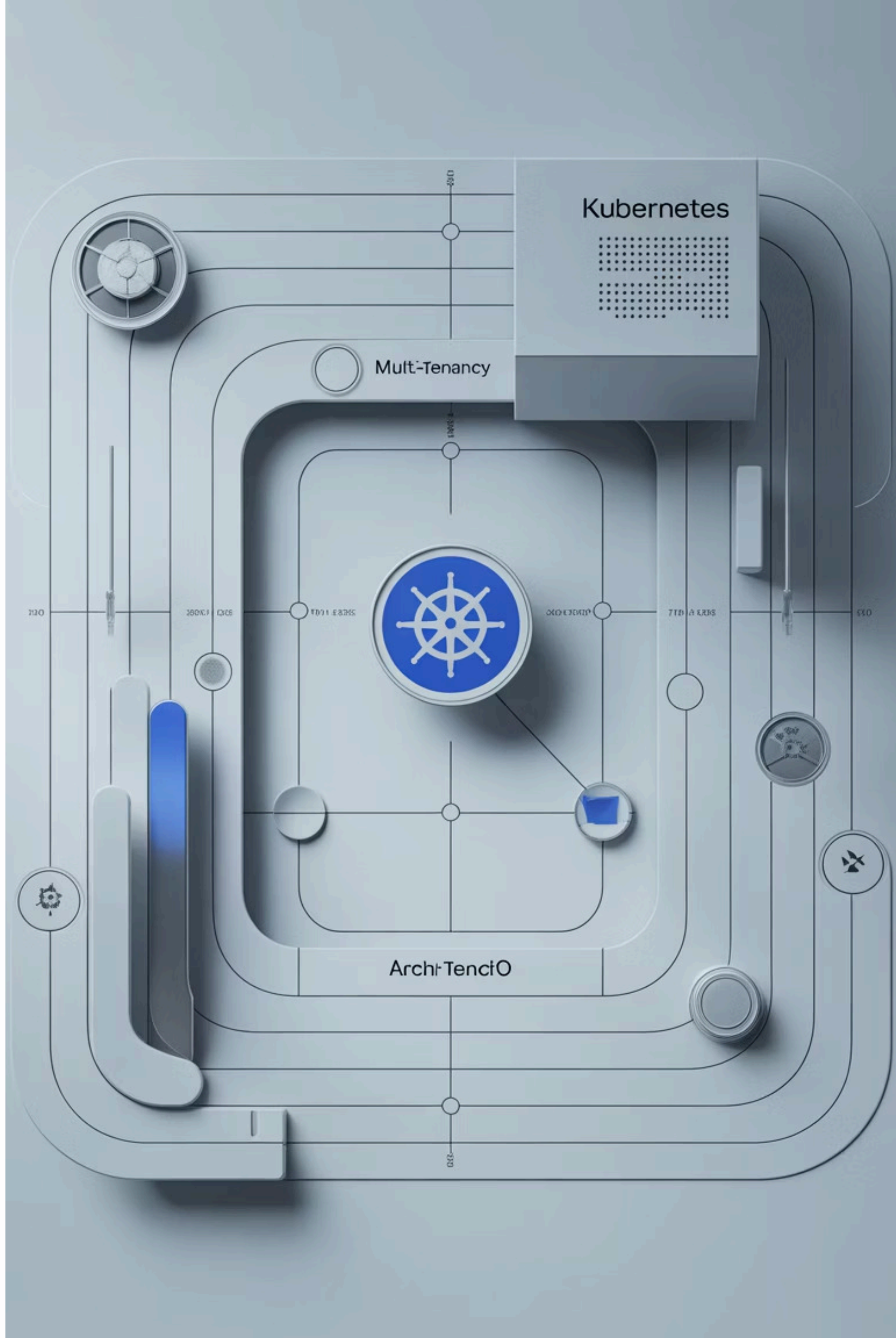
Understanding Isolation Techniques

What Is Multi-Tenancy in Kubernetes?

Multi-tenancy allows multiple teams or applications to share the same Kubernetes cluster while maintaining strict separation. Think of it like an apartment building where each tenant has their own locked unit, but they share the building's infrastructure.

This approach maximizes resource utilization while preventing one team's workload from interfering with another's. However, it requires careful security controls to maintain proper boundaries.



Multi-Tenancy Isolation Strategies

01

Namespace Isolation

Create separate namespaces for different teams or environments. Each namespace acts as a virtual cluster with its own resources and access controls.

02

Network Policies

Define rules that control traffic between pods. By default, deny all traffic and explicitly allow only necessary communication paths.

03

Resource Quotas

Set limits on CPU, memory, and storage per namespace. This prevents resource exhaustion attacks where one tenant consumes all cluster resources.

04

RBAC Controls

Implement role-based access control to ensure users can only access resources in their designated namespaces.

Real-World Example: E-Commerce Platform

Imagine an e-commerce company running multiple services in one cluster:

- **Payment processing** in namespace "payments" with strict network isolation
- **Product catalog** in namespace "catalog" accessible to multiple services
- **Customer service tools** in namespace "support" with limited permissions

Network policies ensure the payment service only communicates with authorized services. Even if the catalog service is compromised, attackers cannot reach payment data.

Sandboxed Containers: Extra Protection

Sandboxed containers add an additional isolation layer beyond standard container runtimes. Technologies like gVisor and Kata Containers create a virtualized environment that limits what containers can access on the host system.

Standard containers share the host kernel, creating potential security risks. Sandboxed containers run in their own isolated kernel space, preventing malicious code from exploiting kernel vulnerabilities to escape the container.

Use sandboxed containers for untrusted workloads, multi-tenant environments, or applications handling sensitive data.

Static Analysis & Vulnerability Scanning

Why Scan Before Deployment?

Catch Issues Early

Finding vulnerabilities in development costs 10x less than fixing them in production. Static analysis catches security flaws before they reach your cluster.

Prevent Configuration Errors

95% of container security incidents result from misconfigurations. Automated scanning identifies these issues before deployment.

Enforce Security Standards

Ensure every workload meets your organization's security baseline. Block deployments that violate policies automatically.

Introducing Kubesec: Security Scoring

Kubesec analyzes Kubernetes resources and assigns a security score based on best practices. It evaluates your YAML manifests and provides actionable recommendations.

What Kubesec checks:

- Privileged container settings
- Security context configurations
- Resource limits and requests
- Capability restrictions
- Read-only root filesystems

Kubesec in Action: Before & After

❌ Insecure Configuration (Score: -30)

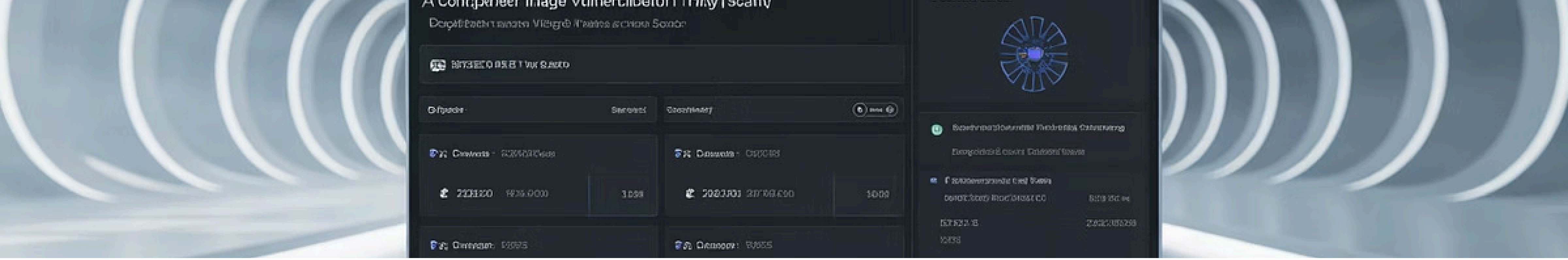
```
apiVersion: v1
kind: Pod
metadata:
name: webapp
spec:
containers:
- name: app
image: myapp:latest
securityContext:
privileged: true
runAsUser: 0
```

Issues: Running as root, privileged mode enabled, no resource limits

✅ Secure Configuration (Score: 10)

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
spec:
  containers:
  - name: app
    image: myapp:v1.2.3
    securityContext:
      runAsNonRoot: true
      runAsUser: 1000
      readOnlyRootFilesystem: true
      allowPrivilegeEscalation: false
    resources:
      limits:
        memory: "512Mi"
        cpu: "500m"
```

Improvements: Non-root user, read-only filesystem, resource limits, specific version tag



Container Image Scanning

Beyond configuration analysis, scan container images for known vulnerabilities. Tools like Trivy, Clair, and Anchore identify security issues in your application dependencies and base images.

These scanners check against CVE databases and flag outdated packages with known exploits. Integrate scanning into your CI/CD pipeline to catch vulnerabilities before images reach production.

Set policies to automatically reject images with critical or high-severity vulnerabilities. This creates a security gate that prevents risky deployments.

Minimize Your Base Image Footprint

The Problem With Bloated Images

80%

Unused packages

Average percentage of dependencies in standard base images that applications never use

5x

Larger attack surface

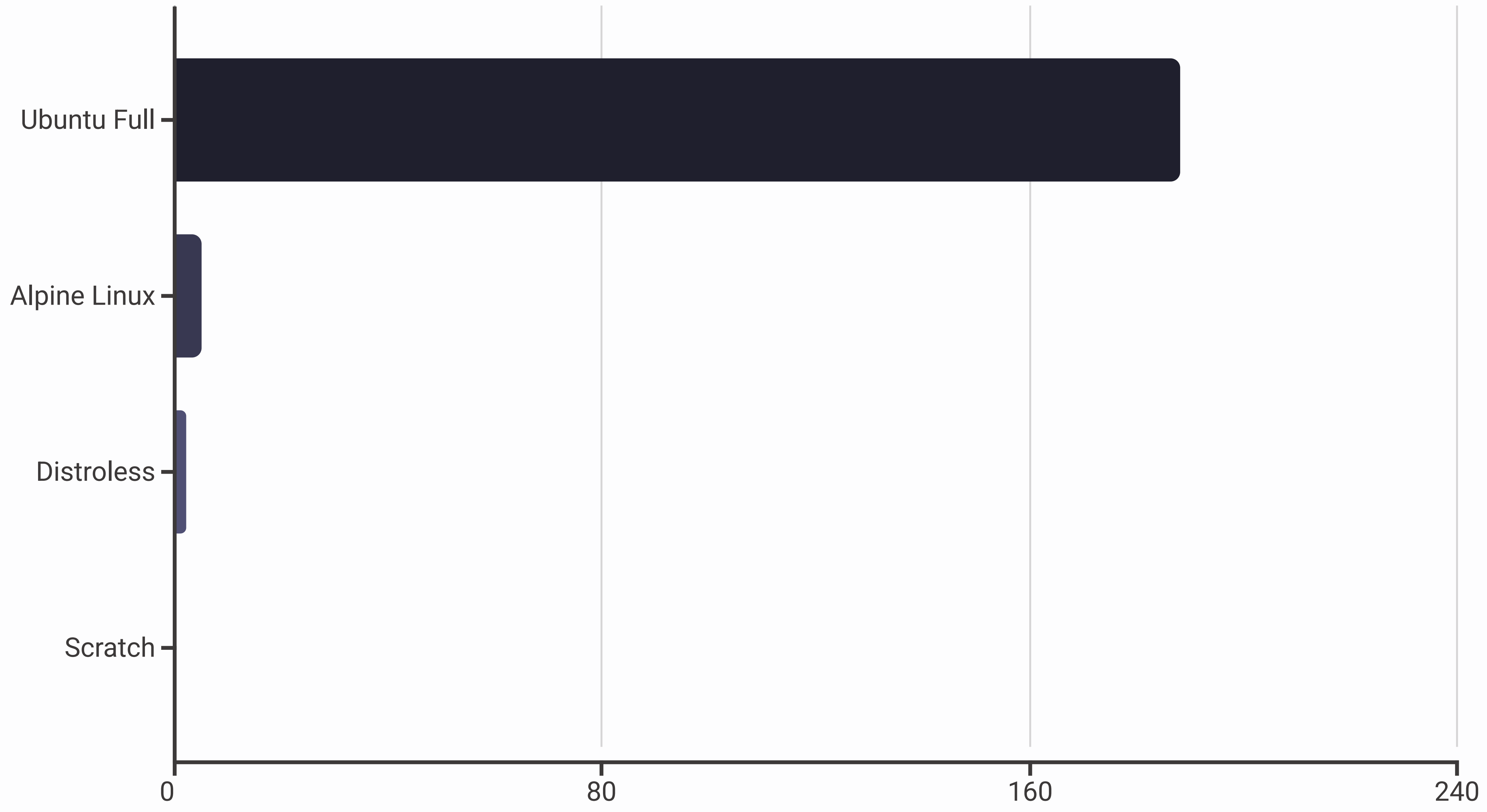
Vulnerability exposure increase in full OS images compared to minimal images

300MB

Wasted storage

Average size difference between standard and minimized container images

Base Image Comparison



Smaller images mean faster deployments, reduced storage costs, and fewer vulnerabilities. Every package you remove is one less potential security hole.

Minimization Strategies

Use Alpine Linux

Replace Ubuntu/Debian with Alpine for a 95% size reduction. Alpine includes only essential packages.

Try Distroless

Google's distroless images contain only your application and runtime dependencies—no shell, no package manager.

Multi-Stage Builds

Build in one image, run in another. Copy only the compiled application to a minimal final image.

Additional Benefits of Minimal Images

Faster Deployment Times

Smaller images download and start faster. Your pods reach running state in seconds instead of minutes, improving scaling responsiveness.

Lower Storage and Transfer Costs

Minimal images reduce registry storage needs and network bandwidth consumption. This saves money at scale when you're running thousands of pods.

Simplified Security Audits

Fewer components mean less to audit and patch. Security teams can focus on actual application dependencies rather than unnecessary OS packages.

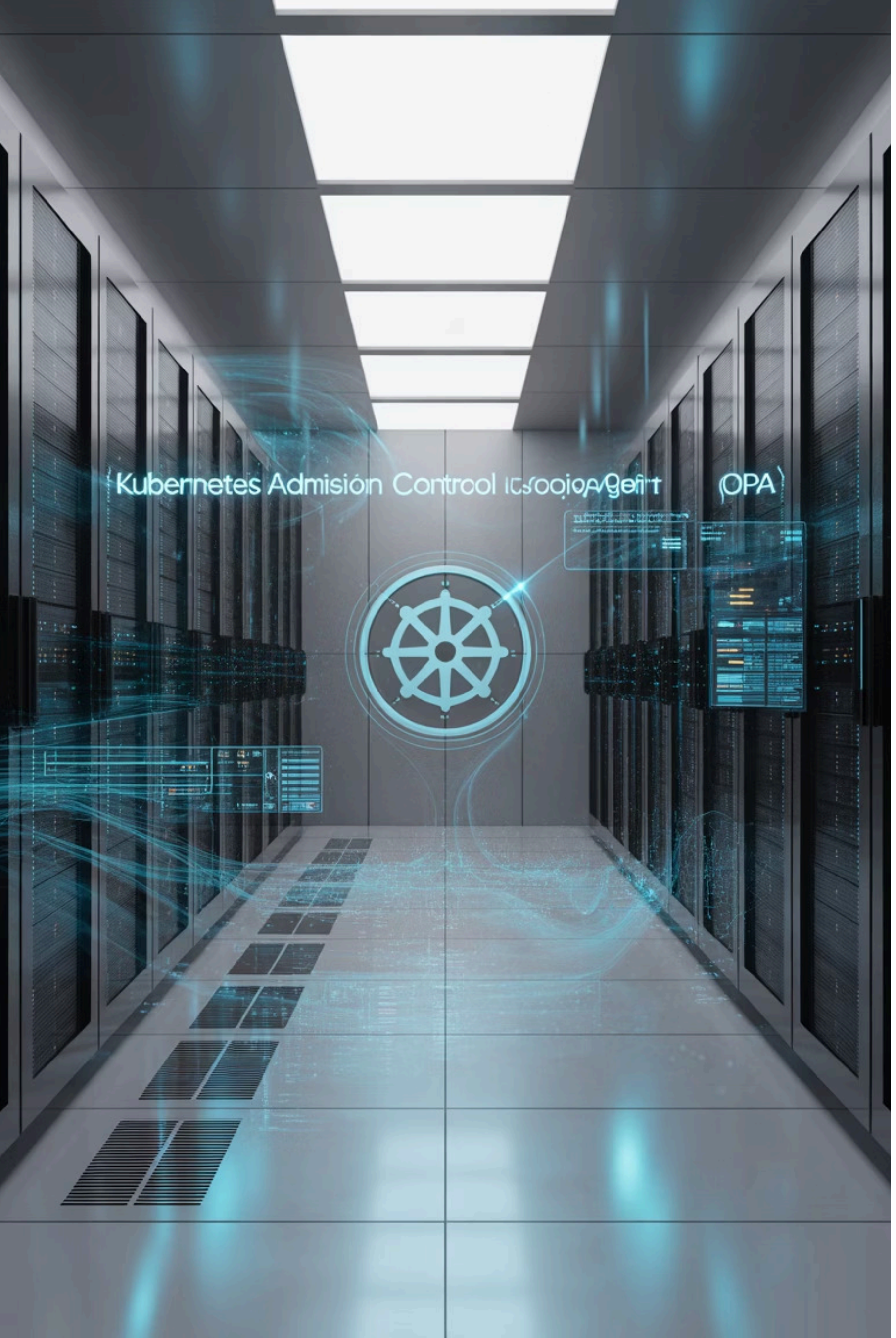
Securing Your Supply Chain

Supply Chain Attacks: The Growing Threat

Container supply chain attacks occur when malicious code enters through compromised base images, dependencies, or registries. Attackers target the build pipeline rather than the application itself.

Recent incidents include compromised npm packages, malicious Docker images on public registries, and backdoored base images that steal credentials.

Organizations must verify every component in their container supply chain to prevent these sophisticated attacks.



OPA Gatekeeper: Policy Enforcement

Open Policy Agent (OPA) Gatekeeper enforces custom security policies in your Kubernetes cluster. It acts as an admission controller that validates resources before they're created.

Gatekeeper uses policies written in Rego language to define what's allowed in your cluster. When a resource doesn't meet policy requirements, Gatekeeper rejects it automatically—no human intervention needed.

This creates a security barrier that prevents policy violations from ever reaching your cluster, even if developers bypass other checks.

Registry Whitelisting with Gatekeeper

01

Define Trusted Registries

Create a list of approved container registries. For example: your private registry, verified Docker Hub official images, and specific GCR repositories.

02

Write Gatekeeper Policy

Define a ConstraintTemplate that checks image origins. The policy extracts the registry from container image names and validates against your whitelist.

03

Apply Constraints

Create Constraint resources that apply the policy to specific namespaces or cluster-wide. Configure enforcement mode (deny or audit).

04

Monitor Violations

Track policy violations through Gatekeeper audit results. Review attempts to use unauthorized registries and refine policies accordingly.

Example: Registry Whitelist Policy

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: allowed-registries
spec:
  match:
    kinds:
    - apiGroups: [""]
      kinds: ["Pod"]
  parameters:
    repos:
    - "mycompany.azurecr.io/"
    - "gcr.io/my-project/"
    - "docker.io/library/"
```

This policy ensures pods only use images from your Azure Container Registry, specific GCR project, or official Docker Hub images. Any pod using a different registry is automatically rejected.

Beyond Registry Control

Image Signing

Use tools like Sigstore/Cosign to cryptographically sign images. Verify signatures before deployment to ensure images haven't been tampered with.

Software Bill of Materials (SBOM)

Generate and attach SBOMs to images. Track all components and dependencies to quickly identify affected images when new vulnerabilities emerge.

Provenance Verification

Verify that images were built by authorized CI/CD pipelines. Reject images that lack build provenance attestations.

Why Encrypt Internal Traffic?

By default, traffic between Kubernetes pods is unencrypted. While your cluster perimeter may be secure, internal network traffic is vulnerable to eavesdropping.

Threat scenarios:

- Compromised nodes can sniff traffic from other pods
- Network plugins or CNI vulnerabilities may expose data
- Compliance requirements mandate encryption in transit
- Multi-tenant environments need strong isolation

Pod-to-pod encryption ensures sensitive data remains protected even if the underlying network is compromised.