ENTERPRISE SECURITY GUIDE

CONTAINER SECURITY

DOCKER & KUBERNETES HARDENING

Complete Enterprise Security Guide

OKAN YILDIZ

DECEMBER 2025

Introduction

Container technologies have fundamentally transformed the landscape of modern software development, deployment, and operations, representing one of the most significant paradigm shifts in enterprise computing since the widespread adoption of virtualization. Docker and Kubernetes have emerged not merely as tools but as foundational platforms that enable cloud-native architectures, microservices deployments, and the DevOps practices that define contemporary software engineering. This transformation has brought unprecedented benefits: applications can be packaged with all their dependencies, deployed consistently across diverse environments, scaled dynamically in response to demand, and managed through declarative configuration that treats infrastructure as code.

However, this revolution in application delivery has introduced a complex new security landscape that challenges traditional cybersecurity approaches and demands fundamentally different thinking about how we protect workloads, data, and infrastructure. Container security is not simply traditional security applied to new technology; it represents a paradigm shift that requires understanding ephemeral workloads, shared kernel architectures, dynamic network topologies, and distributed security controls that operate at unprecedented scale and velocity. The very characteristics that make containers powerful—their lightweight nature, rapid deployment capabilities, and dynamic orchestration create security challenges that existing tools and methodologies often fail to address adequately.

The ephemeral nature of containerized workloads fundamentally disrupts conventional security monitoring and incident response procedures. Traditional security tools were designed for relatively static infrastructure where servers had lifespans measured in months or years, network configurations changed infrequently, and security teams could establish behavioral baselines over extended observation periods. In container environments, pods can be created, destroyed, and recreated in milliseconds; a production Kubernetes cluster might process thousands of these lifecycle events daily. This velocity renders traditional change management processes obsolete and makes manual security reviews impossible. Security controls must be automated, policy-driven, and capable of making real-time decisions without human intervention.

The shared kernel architecture that makes containers efficient also creates unique attack surfaces that don't exist in virtual machine environments. Unlike hypervisor-based virtualization, which provides hardware-level isolation between guests, containers achieve process-level isolation through Linux kernel features like namespaces, cgroups, and security modules. This architecture means that multiple containers share the same operating system kernel, and a vulnerability

in the kernel or a misconfiguration in isolation mechanisms could potentially allow an attacker to escape container boundaries and compromise the host system or other containers. This shared-kernel model demands deep understanding of Linux security primitives and careful configuration of multiple defense layers to achieve adequate isolation.

The declarative configuration model of Kubernetes, while powerful for automation and GitOps workflows, introduces an entirely new category of security vulnerabilities rooted in misconfiguration. Kubernetes provides hundreds of configuration options across dozens of API resources, and the default settings are often optimized for ease of use rather than security. A single misconfigured Role, a permissive NetworkPolicy, or an inadequately restricted PodSecurityPolicy can expose entire clusters to compromise. Unlike traditional infrastructure where misconfigurations might affect individual systems, Kubernetes misconfigurations can have cluster-wide implications affecting hundreds or thousands of workloads simultaneously.

The distributed nature of container orchestration platforms creates security boundaries that cross traditional network perimeters and organizational boundaries. In cloud-native architectures, applications are decomposed into dozens or hundreds of microservices that communicate over network APIs. Each service-to-service interaction represents a potential attack vector that must be authenticated, authorized, encrypted, and monitored. Traditional perimeter security models that focus on north-south traffic (into and out of the datacenter) are insufficient for protecting east-west traffic (between services within the cluster). This requires implementing zero-trust security models where every interaction is verified regardless of network location.

Why Container Security Demands Specialized Expertise

Understanding why container security requires specialized knowledge and dedicated tools begins with recognizing the architectural differences between containerized and traditional workloads. These differences manifest across multiple dimensions:

Isolation Model Differences: Virtual machines provide strong isolation through hardware virtualization, where each VM runs its own kernel and the hypervisor mediates all access to physical resources. Container isolation depends on Linux kernel features that, while sophisticated, provide weaker boundaries than hardware-enforced isolation. Containers share the host kernel, meaning a kernel vulnerability could potentially be exploited from any container to affect the entire system. This shared-kernel architecture means that container security must

include kernel hardening, strict capability management, and defense-in-depth strategies that assume isolation boundaries might be breached.

Lifecycle Velocity Impact: The rapid creation and destruction of containers creates challenges for security monitoring, vulnerability management, and incident response. Traditional security tools that rely on agents installed on long-lived hosts become ineffective when containers exist for seconds or minutes. Vulnerability scanning that occurs weekly or monthly cannot keep pace with container deployments that happen thousands of times daily. Security monitoring must transition from agent-based to agentless architectures, vulnerability management must be shifted left into the CI/CD pipeline, and incident response must leverage ephemeral forensics techniques that can capture evidence before containers disappear.

Network Complexity Explosion: Traditional datacenter networks had relatively static topologies where security teams could map out network flows, document communication patterns, and implement firewall rules based on known-good behaviors. Container orchestration creates highly dynamic network topologies where services are constantly being scaled up and down, rescheduled to different nodes, and their IP addresses are constantly changing. A single application deployment might create dozens of pods across multiple nodes, each with its own IP address that will change when the pod is rescheduled. This dynamism makes traditional network security controls like static firewall rules impractical, requiring adoption of software-defined networking, service mesh architectures, and declarative network policies.

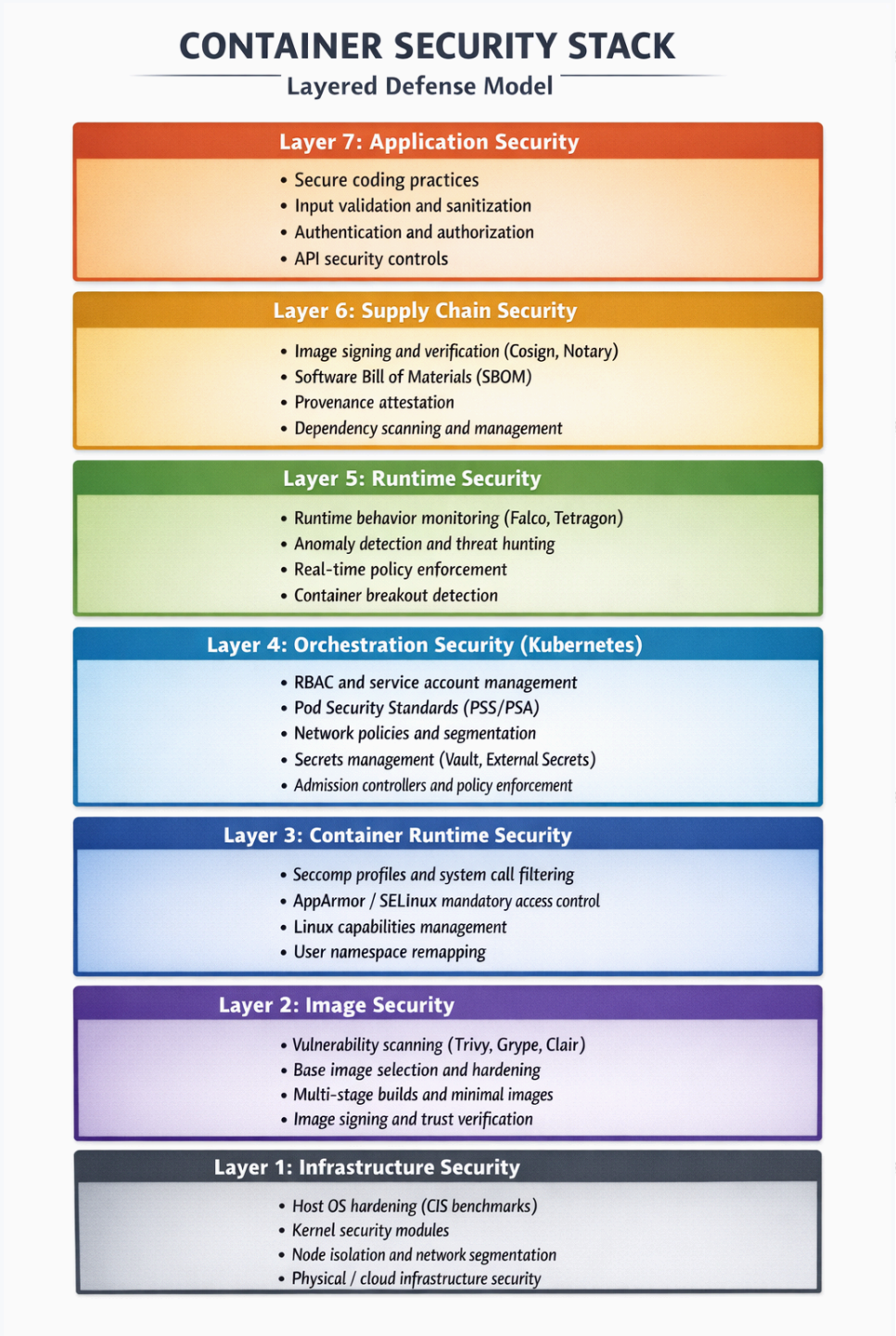
Supply Chain Complexity: Modern containerized applications typically depend on dozens of base images, hundreds of software packages, and thousands of transitive dependencies, each representing a potential point of compromise in the software supply chain. Traditional software supply chain security focused primarily on validating the authenticity of software packages and checking them for known vulnerabilities. Container supply chain security must additionally address image provenance, runtime integrity, and the security of the entire build pipeline that creates container images. This requires implementing software bill of materials (SBOM), image signing and verification, admission control policies, and continuous monitoring for newly discovered vulnerabilities in running containers.

State Management Challenges: Containers follow the principle of immutability rather than updating running containers, new versions are deployed and old versions are terminated. This immutability provides security benefits by preventing persistence mechanisms and limiting the impact of compromises, but it also creates challenges for traditional security practices. Log aggregation must be handled externally since container logs are lost when containers terminate. Security state like authentication sessions must be managed in external

services. Incident response procedures must adapt to investigate workloads that no longer exist.

The Container Security Stack: A Layered Defense Model

Effective container security requires implementing controls across multiple layers of the technology stack, from the base infrastructure through the application code. Each layer provides specific security functions, and comprehensive security demands addressing all layers systematically:



This layered security model reflects the principle of defense in depth, where multiple independent security controls work together to protect against threats. A vulnerability or misconfiguration in one layer might be caught by controls in another layer, preventing successful attacks. Understanding this model is essential for developing comprehensive container security strategies that address threats systematically rather than reactively responding to individual incidents.

Layer 1: Infrastructure Security forms the foundation upon which all other security controls depend. If the host operating system or underlying infrastructure is compromised, container isolation mechanisms become irrelevant. This layer includes hardening the host OS according to security benchmarks like CIS or DISA STIGs, enabling and configuring kernel security modules like SELinux or AppArmor, implementing network segmentation to limit blast radius, and ensuring physical or cloud infrastructure security. For cloud-native deployments, this extends to properly configuring cloud provider security controls, implementing strong identity and access management, and enabling audit logging and monitoring.

Layer 2: Image Security addresses the security of container images throughout their lifecycle. Images must be scanned for vulnerabilities during the build process, base images must be selected carefully to minimize attack surface, and multi-stage builds should be used to exclude unnecessary tools and dependencies from production images. Images should be signed to ensure authenticity and integrity, and registries must be secured with proper access controls. This layer is critical because vulnerabilities in container images are among the most common security issues in container environments, and many organizations unknowingly deploy containers with critical CVEs that could be exploited by attackers.

Layer 3: Container Runtime Security implements fine-grained controls over what containers can do at runtime. Seccomp profiles restrict system calls that containers can make, preventing exploitation of kernel vulnerabilities. AppArmor and SELinux provide mandatory access control that limits file system access and other operations. Linux capabilities allow fine-grained division of root privileges, enabling containers to perform specific privileged operations without full root access. User namespace remapping ensures that root inside containers maps to unprivileged users on the host. These controls work together to limit the impact of container compromises and prevent container escape attacks.

Layer 4: Orchestration Security addresses security at the Kubernetes cluster level. Role-Based Access Control (RBAC) ensures that users and service accounts have only the permissions they need to perform their functions. Pod Security Standards and Pod Security Admission provide baseline security

requirements that all pods must meet. Network Policies implement microsegmentation to control communication between pods and external services. Secrets management ensures sensitive data like credentials and API keys are stored and accessed securely. Admission controllers enforce policies that prevent insecure configurations from being deployed. This layer is particularly important because misconfigurations at the orchestration level often have cluster-wide security implications.

Layer 5: Runtime Security monitors container behavior at runtime to detect and respond to suspicious activities. Tools like Falco and Tetragon use eBPF to observe system calls, network connections, and file system operations, comparing observed behavior against defined security policies. This enables detection of container escape attempts, cryptocurrency mining, command-and-control communication, privilege escalation, and other malicious activities. Runtime security provides essential defense against zero-day exploits and insider threats that might bypass other security controls. The real-time nature of runtime security allows for immediate response to threats, potentially blocking attacks before they cause damage.

Layer 6: Supply Chain Security ensures the integrity of the entire software delivery pipeline from source code to production deployment. This includes verifying the provenance of base images and dependencies, signing images to prove authenticity, generating and maintaining SBOMs for vulnerability tracking, and implementing admission control that prevents unsigned or unverified images from being deployed. Supply chain attacks have become increasingly sophisticated, targeting development tools, package repositories, and CI/CD pipelines. Comprehensive supply chain security is essential for maintaining trust in containerized applications.

Layer 7: Application Security addresses vulnerabilities in the application code itself. Even with all infrastructure security controls properly implemented, vulnerabilities like SQL injection, cross-site scripting, or business logic flaws in the application can be exploited by attackers. Secure coding practices, input validation, proper authentication and authorization, and API security controls are essential components of application security. Container environments don't eliminate the need for application security—they simply provide an additional layer of defense that can limit the impact of application vulnerabilities.

The Container Threat Landscape: Understanding Attack Vectors

To implement effective security controls, security professionals must understand the specific threats that target containerized environments. Container security

threats can be categorized based on the attack vector and the security layer being targeted:

Threat Category	Attack Vectors	Typical Impact	Detection Difficulty
Vulnerable Images	Deploying images with known CVEs, outdated packages, malware	High - Direct code execution, data theft	Low - Detectable via scanning
Misconfiguration	Privileged containers, exposed secrets, permissive RBAC	Critical - Cluster compromise	Medium - Requires policy enforcement
Container Escape	Exploiting kernel vulnerabilities, capability abuse	Critical - Host compromise	High - Requires runtime monitoring
Supply Chain Attacks	Compromised base images, malicious dependencies	Critical - Backdoor deployment	High - Requires provenance verification
Network Attacks	Service-to-service exploitation, traffic interception	High - Data exfiltration	Medium - Requires network monitoring
Secrets Exposure	Hardcoded credentials, exposed environment variables	High - Credential theft	Low - Detectable via image scanning
Resource Abuse	Cryptocurrency mining, DDoS participation	Medium - Resource drain	Medium - Detectable via anomaly detection
Data Exfiltration	Copying sensitive data, DNS tunneling	High - Data loss	High - Requires behavioral analysis

Vulnerable Images remain one of the most common security issues in container environments, yet they are also among the most preventable. Organizations

frequently deploy containers built from outdated base images that contain critical vulnerabilities, often because they lack automated vulnerability scanning in their CI/CD pipelines. Attackers routinely scan public container registries and Kubernetes clusters looking for containers running vulnerable software versions. Once identified, these vulnerabilities can be trivially exploited using publicly available exploit code. The solution lies in implementing continuous vulnerability scanning, automatically rebuilding images when base image updates are released, and enforcing policies that prevent deployment of images with critical vulnerabilities.

Misconfiguration Vulnerabilities are perhaps the most insidious threat in container environments because they often result from complexity rather than malice. Kubernetes provides immense flexibility through its configuration options, but this flexibility creates countless opportunities for security misconfigurations. Running containers as root, granting excessive privileges, using the host network namespace, mounting the Docker socket, and configuring permissive RBAC are all common misconfigurations that dramatically weaken security. These issues are particularly dangerous because they often appear to work correctly from a functional perspective while creating serious security vulnerabilities. Organizations must implement strong governance, policy-as-code enforcement through admission controllers, and regular security audits to prevent misconfigurations.

Container Escape Attacks represent the most severe threat to containerized environments because successful exploitation allows attackers to break out of container isolation and compromise the underlying host system. These attacks typically exploit vulnerabilities in the Linux kernel, misconfigurations in container runtime security controls, or design flaws in container orchestration platforms. Recent years have seen several high-profile container escape vulnerabilities including runc exploits (CVE-2019-5736), kernel vulnerabilities exploitable from containers, and Kubernetes privilege escalation bugs. Defense against container escape requires implementing multiple layers of security controls: kernel hardening, seccomp profiles, user namespace remapping, and runtime security monitoring that can detect escape attempts.

Supply Chain Attacks targeting container ecosystems have increased dramatically as attackers recognize that compromising a widely-used base image or popular package can provide access to thousands of downstream applications. These attacks might involve publishing malicious packages to public repositories, compromising official base images through account takeovers, or injecting malware into CI/CD pipelines. The distributed and collaborative nature of modern software development creates many opportunities for supply chain compromise. Organizations must implement comprehensive supply chain security including image signing, SBOM generation

and analysis, provenance verification, and admission control that enforces supply chain security policies.

Network-Based Attacks in container environments are complicated by the dynamic and distributed nature of microservices architectures. Service-to-service communication creates many potential attack vectors that traditional perimeter security controls cannot address. Attackers who compromise one service can often pivot to other services by exploiting weak authentication, lack of encryption, or permissive network policies. Lateral movement is particularly easy in clusters that lack network segmentation. Defense requires implementing zero-trust networking principles, service mesh for mutual TLS, network policies for microsegmentation, and comprehensive network monitoring to detect unusual communication patterns.

Secrets Management Failures continue to be a major source of container security incidents. Developers frequently hardcode credentials in container images, pass secrets through environment variables, or store them in unencrypted Kubernetes secrets. These practices make credentials easily accessible to anyone with access to the container image or cluster. Attackers who gain access to containers can quickly locate and exfiltrate these credentials, using them to access databases, APIs, and other sensitive systems. Proper secrets management requires using dedicated secret stores like HashiCorp Vault, implementing dynamic credential generation, encrypting secrets at rest and in transit, and regularly rotating credentials.

This comprehensive understanding of container security fundamentals, the layered defense model, and the threat landscape provides the foundation for the detailed technical discussions that follow in subsequent chapters. Each chapter examines specific security domains in depth, providing practical guidance, real-world examples, and production-ready configurations for securing containerized environments.

Chapter 1: Docker Security Hardening

Docker security forms the foundation upon which Kubernetes and other container orchestration platforms build their own security models. Understanding how to properly secure Docker installations, build secure images, and configure runtime security controls is essential for anyone working with containerized applications. This chapter provides comprehensive coverage of Docker security hardening, from selecting secure base images to implementing runtime protection mechanisms.

1.1 Understanding Docker's Security Architecture

Before diving into specific hardening techniques, it's crucial to understand Docker's security architecture and how it leverages Linux kernel features to provide container isolation. Docker's security model relies on several key Linux primitives:

Namespaces provide isolation by giving each container its own view of system resources. Docker uses multiple namespace types to achieve comprehensive isolation. The PID namespace ensures containers cannot see processes running in other containers or on the host. The network namespace gives each container its own network stack, including interfaces, routing tables, and firewall rules. The mount namespace isolates the filesystem view, preventing containers from seeing or accessing filesystems of other containers or sensitive host paths. The UTS namespace provides hostname isolation. The IPC namespace isolates inter-process communication mechanisms like message queues and semaphores. Finally, the user namespace (though not enabled by default in all configurations) allows mapping container user IDs to different IDs on the host, enabling root inside containers to be unprivileged users on the host.

Control Groups (cgroups) limit the resources containers can consume, preventing resource exhaustion attacks where a compromised or malicious container attempts to monopolize CPU, memory, or I/O resources. Cgroups are essential security controls because they prevent denial-of-service attacks and ensure workload stability. Docker automatically creates cgroup hierarchies for each container, applying limits specified in the `docker run` command or Docker Compose configuration.

Capabilities provide fine-grained privilege control by dividing root's traditional monolithic privilege into dozens of distinct capabilities. Instead of running containers as full root or completely unprivileged, capabilities allow granting specific privileges like binding to low-numbered ports or changing file ownership without granting all root privileges. Docker drops many dangerous capabilities by default but still grants more than necessary for most applications.

Seccomp Profiles filter system calls that containers can make to the kernel, providing a critical defense layer against kernel exploits. Docker applies a default seccomp profile that blocks approximately 50 system calls known to be dangerous or unnecessary for normal container operations. Custom seccomp profiles can further restrict system call access based on application requirements.

AppArmor and SELinux provide mandatory access control that restricts container operations regardless of user privileges. These Linux Security Modules enforce security policies that can prevent containers from accessing

sensitive files, making network connections, or performing other potentially dangerous operations even if the container is compromised.

Understanding these security primitives is essential because effective Docker hardening requires configuring each mechanism appropriately. Relying solely on default settings leaves containers vulnerable to numerous attack vectors.

1.2 Secure Base Image Selection

The choice of base image fundamentally determines a container's security posture. Base images vary dramatically in their attack surface, update frequency, and security characteristics. Selecting appropriate base images and keeping them updated is one of the most impactful security decisions in container development.

Minimizing Attack Surface: The principle of least functionality suggests using the smallest possible base image that can support your application. Traditional base images like ubuntu:latest or centos:latest include hundreds of packages, system utilities, and services that typical containerized applications never use. Each additional package represents potential vulnerabilities and additional attack surface. Package managers, compilers, and debugging tools included in full OS images provide useful capabilities for attackers who compromise containers.

Distroless Images: Google's distroless images represent the extreme of minimalism, containing only the application runtime and its dependencies without even a shell or package manager. These images dramatically reduce attack surface but require careful planning during development since they don't include debugging tools. Distroless images are excellent for production deployments where security takes priority over convenience.

Alpine Linux: Alpine-based images provide a middle ground between full OS images and distroless images. Alpine uses musl libc instead of glibc and BusyBox instead of GNU utilities, resulting in images that are typically 5-10x smaller than equivalent Debian or Ubuntu images. However, Alpine's different libc can occasionally cause compatibility issues with applications that assume glibc. Alpine is regularly updated and maintains a good security track record.

Scratch Images: For statically compiled languages like Go, building from the scratch base image (which is literally empty) creates the smallest possible images containing only your application binary. This approach provides the best security posture but requires your application to be completely self-contained with no external dependencies.

Base Image Update Strategy: Regardless of which base image you choose, establishing a systematic update process is critical. Base images receive

security updates regularly, and applications must be rebuilt to incorporate these updates. Automated image rebuilds triggered by base image updates ensure applications benefit from security patches promptly. Many organizations scan registries weekly or even daily, automatically rebuilding images when updated base images become available.

Here's an example demonstrating the security difference between different base image choices:

```
# ❌ INSECURE: Full Ubuntu base image
# This image includes ~200MB of packages, including:
# - Package managers (apt, dpkg)
# - System utilities (systemctl, bash)
# - Development tools (make, gcc)
# - Unnecessary daemons and services
# Attack surface: HIGH - Hundreds of binaries, multiple privilege escalation
vectors

FROM ubuntu:22.04
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    && rm -rf /var/lib/apt/lists/*
COPY requirements.txt /app/
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
CMD ["python3", "app.py"]

# Result: ~600MB image with numerous attack vectors


# ✅ BETTER: Alpine-based minimal image
# Alpine includes minimal packages with smaller binaries
# Attack surface: MEDIUM - Basic utilities but much reduced package count

FROM python:3.11-alpine
# Install only runtime dependencies, no build tools in final image
RUN apk add --no-cache libpq
COPY requirements.txt /app/
WORKDIR /app
# Install Python packages without caching
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
# Run as non-root user
RUN adduser -D appuser
USER appuser
CMD ["python", "app.py"]

# Result: ~150MB image with reduced attack surface
```

```
# ✅✅ BEST: Multi-stage build with distroless final image
# This approach uses full image for building but distroless for runtime
# Attack surface: MINIMAL - No shell, no package manager, only runtime

# Build stage: Full image with build tools
FROM python:3.11-slim as builder
WORKDIR /app
COPY requirements.txt .
# Install dependencies to a specific location
RUN pip install --user --no-cache-dir -r requirements.txt

# Runtime stage: Distroless image for production
FROM gcr.io/distroless/python3-debian11
# Copy only the application and dependencies, nothing else
COPY --from=builder /root/.local /root/.local
COPY app.py /app/
WORKDIR /app
# Distroless images run as non-root by default
ENV PATH=/root/.local/bin:$PATH
CMD ["app.py"]

# Result: ~50MB image, no shell or package manager, minimal attack surface
```

Detailed Analysis of the Security Differences:

The first approach using a full Ubuntu base image creates approximately 600MB container with hundreds of installed packages. This image includes complete GNU utilities, bash shell, package managers, and potentially numerous services. An attacker who compromises the application running in this container gains access to extensive system utilities that facilitate lateral movement, privilege escalation, and persistence. The package manager allows installing additional tools, the shell enables running complex attack scripts, and the numerous SUID binaries provide potential privilege escalation vectors.

The second approach using Alpine Linux significantly reduces the image size to approximately 150MB and eliminates many unnecessary packages. Alpine's minimalist philosophy means fewer installed binaries, smaller attack surface, and faster image pulls. However, Alpine still includes a shell (sh via BusyBox) and package manager (apk), which could be leveraged by attackers. The alpine approach represents a practical balance between security and operational convenience, as the shell facilitates debugging and troubleshooting.

The third approach using multi-stage builds and distroless base images achieves the smallest attack surface at approximately 50MB. The multi-stage build allows using a full-featured image (python:3.11-slim) during the build stage

where development tools are needed, then copying only the application artifacts to a minimal distroless image for the runtime stage. The distroless image contains only the Python runtime and application code—no shell, no package manager, no system utilities. This means an attacker who compromises the application container cannot execute shell commands, install additional tools, or leverage system utilities for lateral movement. The absence of these tools significantly limits attacker capabilities.

Important Security Considerations: While distroless images provide excellent security, they present operational challenges. Without a shell, traditional debugging techniques like `docker exec` into containers to run commands don't work. Teams using distroless images must adapt their troubleshooting workflows, relying more on structured logging, distributed tracing, and external debugging tools. During development, teams often use regular base images to enable interactive debugging, then switch to distroless images for production deployments.

1.3 Multi-Stage Build Security Pattern

Multi-stage builds are a Docker feature that dramatically improves both security and efficiency by separating the build environment from the runtime environment. This pattern is considered a best practice for production containerized applications and is essential for achieving minimal attack surface.

Understanding Multi-Stage Build Security Benefits: Traditional single-stage Dockerfiles include all build dependencies in the final image. This means development tools, compilers, build scripts, source code, and intermediate build artifacts all end up in production containers. These components serve no purpose at runtime but create security risks: build tools might have vulnerabilities, source code could reveal intellectual property or security flaws, and build dependencies add unnecessary attack surface.

Multi-stage builds solve this by using multiple FROM statements in a single Dockerfile. Early stages contain the full development environment needed for compilation and building. Later stages start from minimal base images and copy only the compiled application binaries from earlier stages. This pattern ensures build tools and source code never appear in production images.

Here's a comprehensive example demonstrating multi-stage build security for a Go application:

```
# ==============================================================================
# STAGE 1: BUILDER - Full development environment with build tools
# ==============================================================================
# Use golang image with complete build tools
```

```
FROM golang:1.21-alpine AS builder

# Security hardening for build stage
# Install only essential build dependencies and security tools
RUN apk add --no-cache \
    git \
    ca-certificates \
    # Add build dependencies
    make \
    # Security scanning tools for build-time checks
    && apk add --no-cache
--repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing trivy

# Create non-root user for building (defense in depth even in build stage)
RUN adduser -D -g '' appuser

# Set working directory
WORKDIR /build

# Copy dependency definition files first (better layer caching)
COPY go.mod go.sum ./

# Download dependencies (cached unless go.mod/go.sum change)
# Verify dependencies with checksums
RUN go mod download && \
    go mod verify

# Run dependency vulnerability scan during build
RUN trivy fs --severity HIGH,CRITICAL --no-progress /go/pkg/mod

# Copy source code
COPY . .

# Run static security analysis on source code
# This fails the build if critical issues are found
RUN go vet ./...

# Build the application with security flags
# CGO_ENABLED=0: Disable CGO for static binary (better security)
# -ldflags='-w -s': Strip debug info and symbols (smaller, harder to reverse
engineer)
# -extldflags "-static": Create fully static binary
# -a: Force rebuild of all packages
# -installsuffix cgo: Use different install suffix for CGO-disabled build
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
    -ldflags='-w -s -extldflags "-static"' \
    -a \
    -installsuffix cgo \
    -o /app/server \
    ./cmd/server
```

```
# Verify the binary is statically linked (critical for scratch base image)
RUN ldd /app/server 2>&1 | grep -q "not a dynamic executable"

# ==================================================================================
# STAGE 2: RUNTIME - Minimal production image
# ==================================================================================
# Start from empty scratch image (literally nothing except kernel interface)
FROM scratch

# Copy CA certificates for HTTPS connectivity
# Even minimal images need these for external API calls
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/

# Copy user configuration from builder stage
# This enables running as non-root even in scratch image
COPY --from=builder /etc/passwd /etc/passwd

# Copy only the compiled binary (single file, nothing else)
COPY --from=builder /app/server /server

# Set non-root user
USER appuser

# Document which port the application uses (does not actually publish)
EXPOSE 8080

# Define health check endpoint for container orchestration
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD ["/server", "--healthcheck"]

# Set the entrypoint to our application
ENTRYPOINT ["/server"]

# Result: ~10MB image containing ONLY:
# - The application binary
# - CA certificates
# - User configuration
# Nothing else - no shell, no utilities, no package manager
```

Security Analysis of This Multi-Stage Build:

This Dockerfile implements multiple security best practices in both the build and runtime stages:

Build Stage Security Measures:

1. ***Dependency Verification:*** The `go mod download` and `go mod verify` commands ensure dependencies match expected checksums, preventing

supply chain attacks where dependencies are tampered with during download.

2. ***Build-Time Security Scanning:*** Integrating Trivy into the build stage ensures dependencies are scanned for vulnerabilities before the application is compiled. If critical vulnerabilities are detected, the build fails, preventing vulnerable images from being created.
3. ***Static Analysis:*** Running `go vet` performs static analysis on the source code to catch potential bugs and security issues before compilation. This catches common issues like SQL injection vulnerabilities, race conditions, and improper error handling.
4. ***Secure Compilation Flags:*** The build command uses multiple security-focused flags:
    - `CGO_ENABLED=0` disables CGO, creating a fully static binary with no external dependencies. This is critical for running from scratch base images and reduces the risk of library-level vulnerabilities.
    - `-ldflags='-w -s'` strips debug information and symbol tables, making the binary smaller and more difficult to reverse engineer.
    - `-extldflags "-static"` ensures the binary is fully statically linked.
    - The `ldd` check verifies the binary is indeed static, which is essential for the scratch base image to work.

Runtime Stage Security Benefits:

1. ***Scratch Base Image:*** Starting from the scratch image means the container contains literally nothing except the kernel interface and what we explicitly copy. There is no shell, no package manager, no system utilities—nothing an attacker could leverage for lateral movement or privilege escalation.
2. ***Minimal Content:*** Only three things are copied to the runtime image: CA certificates (required for HTTPS connections), user configuration (required for running as non-root), and the application binary itself. This represents the absolute minimum required for the application to function.
3. ***Non-Root User:*** Even in a scratch image, the application runs as a non-root user, providing defense in depth. If the application is compromised, the attacker has limited capabilities even within the already-minimal container.
4. ***Binary-Only Deployment:*** With no source code, build scripts, or intermediate artifacts in the production image, there's no risk of intellectual property exposure or attackers analyzing source code for vulnerabilities.
5. ***Immutable Deployment:*** The lack of package managers and shells makes it impossible to modify the container at runtime. Any attempts to install tools or modify the system will fail, preventing many persistence and lateral movement techniques.

Operational Considerations: While this approach provides excellent security, it requires teams to adapt their operational practices. Traditional debugging with

`docker exec` doesn't work in scratch-based containers since there's no shell. Teams must rely on:

- Structured logging sent to external systems
- Distributed tracing for understanding application behavior
- External debugging and profiling tools
- Development containers with full tooling for pre-production testing

Many organizations use multi-stage builds with regular base images during development for easy debugging, then switch to scratch-based images for production deployments where security is paramount.

1.4 Dockerfile Security Best Practices

Beyond base image selection and multi-stage builds, numerous Dockerfile instructions and patterns impact container security. Writing secure Dockerfiles requires understanding these patterns and consistently applying security best practices.

USER Directive - Never Run as Root: Running containers as root is one of the most common security misconfigurations and one of the easiest to fix. When containers run as root, a compromised application gains root privileges within the container, making privilege escalation and container escape attacks more likely to succeed. Even though container isolation provides some protection, defense in depth dictates running with minimum necessary privileges.

COPY vs ADD - Preventing Archive Exploits: The ADD instruction has automatic extraction capabilities for archives and supports URL sources, but these features create security risks. ADD will automatically extract tar archives, which could be exploited to overwrite system files if not carefully controlled. The COPY instruction is more predictable and should be preferred. Only use ADD when you specifically need its special features, and always validate archive contents.

RUN Command Security - Avoiding Unnecessary Tools: Each RUN instruction creates a layer in the Docker image, and these layers persist even if later instructions remove files. This means secrets accidentally included in intermediate layers remain accessible in the image. Security-sensitive operations should never write secrets to disk, should clean up after themselves in the same RUN instruction, and should combine related commands to minimize layers.

```
# ================================================================================
# COMPREHENSIVE DOCKERFILE SECURITY EXAMPLE
# Demonstrates all major security best practices for production containers
# ================================================================================
```

```
# Start from official, version-pinned base image
# Always use specific versions, never use 'latest' tag
# Pin to digest for immutability: image content cannot change
FROM node:18.17.0-alpine@sha256:abc123...

# ================================================================================
# STAGE 1: DEPENDENCY INSTALLATION (runs as root, needed for system packages)
# ================================================================================

# Install system dependencies in a single layer to minimize image size
# --no-cache prevents cache poisoning and reduces image size
# Pin package versions to ensure reproducible builds
RUN apk add --no-cache \
    # Only include absolutely necessary system packages
    dumb-init=1.2.5-r2 \
    && rm -rf /var/cache/apk/*

# Create non-root user and group with fixed UID/GID
# Fixed IDs ensure consistency across container restarts
# Never rely on automatic UID assignment which varies
RUN addgroup -g 10001 -S appgroup && \
    adduser -u 10001 -S appuser -G appgroup

# Create application directory with proper ownership
# Set permissions before switching to non-root user
RUN mkdir -p /home/appuser/app && \
    chown -R appuser:appgroup /home/appuser/app

# ================================================================================
# STAGE 2: APPLICATION SETUP (switch to non-root user)
# ================================================================================

# Switch to non-root user for all subsequent operations
# This ensures package installation and app setup run without root privileges
USER appuser

# Set working directory owned by appuser
WORKDIR /home/appuser/app

# Copy package definition files
# Use COPY instead of ADD (more predictable, no automatic extraction)
COPY --chown=appuser:appgroup package*.json ./

# Install Node.js dependencies with security flags
# --only=production: Exclude devDependencies from production image
# --no-optional: Skip optional dependencies (reduce attack surface)
# npm audit: Check for known vulnerabilities during build
RUN npm ci --only=production --no-optional && \
    # Audit dependencies and fail build if HIGH/CRITICAL vulns found
    npm audit --audit-level=high && \
```

```
    # Remove npm cache to reduce image size and eliminate temp files
    npm cache clean --force && \
    # Remove package manager files that aren't needed at runtime
    rm -f package-lock.json

# Copy application source code
# .dockerignore should exclude: .git, tests, docs, CI configs
COPY --chown=appuser:appgroup . .

# Remove any files that shouldn't be in production image
# This includes test files, documentation, development configs
RUN rm -rf \
    test/ \
    tests/ \
    docs/ \
    *.md \
    .git/ \
    .github/ \
    .gitignore \
    .dockerignore

# ==============================================================================
# SECURITY METADATA AND CONFIGURATION
# ==============================================================================

# Document application metadata using OCI labels
# These labels provide security and operational context
LABEL maintainer="security@company.com" \
      org.opencontainers.image.title="Secure Node Application" \
      org.opencontainers.image.description="Production Node.js microservice" \
      org.opencontainers.image.version="1.0.0" \
      org.opencontainers.image.vendor="Company Inc" \
      org.opencontainers.image.created="2024-01-01" \
      security.contact="security@company.com" \
      security.scanningRequired="true"

# Expose port - documentation only, does not actually publish
# Always use non-privileged port (>1024) to avoid requiring root
EXPOSE 8080

# Set environment variables for security
ENV NODE_ENV=production \
    # Prevent npm from sending telemetry
    NPM_CONFIG_LOGLEVEL=error \
    # Ensure Node.js doesn't run as PID 1 (handled by dumb-init)
    NODE_OPTIONS="--max-old-space-size=512"

# Configure health check for orchestration platforms
# Kubernetes/Docker can use this to determine container health
HEALTHCHECK --interval=30s \
            --timeout=5s \
```

```
            --start-period=10s \
            --retries=3 \
            CMD node /home/appuser/app/healthcheck.js

# Use dumb-init to properly handle signals and reap zombie processes
# This ensures graceful shutdown and prevents PID 1 zombie problems
# CRITICAL: PID 1 has special responsibilities in Linux that apps don't handle
ENTRYPOINT ["dumb-init", "--"]

# Define the command to run the application
# Use exec form (JSON array) not shell form to avoid shell process
CMD ["node", "server.js"]

# ==============================================================================
# SECURITY SUMMARY OF THIS DOCKERFILE:
# ✓ Version-pinned base image with digest
# ✓ Minimal system packages (only dumb-init)
# ✓ Runs as non-root user (UID 10001)
# ✓ Fixed UID/GID for consistency
# ✓ COPY instead of ADD (no automatic extraction)
# ✓ Single-layer package installation
# ✓ Production dependencies only (no devDependencies)
# ✓ Vulnerability scanning during build (npm audit)
# ✓ Removed package manager cache
# ✓ Removed unnecessary files (tests, docs, etc.)
# ✓ OCI labels for security tracking
# ✓ Health check configuration
# ✓ Proper signal handling with dumb-init
# ✓ Non-privileged port (8080, not 80)
# ✓ Exec form CMD (no shell wrapper)
# ==============================================================================
```

Detailed Analysis of Security Controls:

Image Version Pinning: The Dockerfile uses `FROM node:18.17.0-alpine@sha256:abc123...` which pins not just the version (18.17.0) but also the specific image digest. This ensures the exact same image is used every time, preventing potential attacks where an attacker compromises the image registry and replaces images with malicious versions. Without digest pinning, even a version-pinned tag like `18.17.0` could potentially be overwritten in the registry.

Non-Root User Implementation: The Dockerfile creates a dedicated user (`appuser`) with a fixed UID (10001) and runs all application operations as this user. Fixed UIDs are important because they ensure consistent file ownership across container restarts and different nodes in a cluster. Random or automatic UID assignment can cause permission issues with persistent volumes. The

specific UID (10001) is chosen to be high enough to avoid conflicts with system users but consistent enough to use in security policies.

COPY vs ADD Security: Using COPY instead of ADD prevents several potential security issues. ADD automatically extracts tar archives and supports fetching URLs, both of which create attack vectors. An attacker who can control archive contents could potentially overwrite system files during the extraction process. COPY is more predictable and safer.

Dependency Security - Production Only: The `npm ci --only=production` command installs only production dependencies, excluding devDependencies that might include testing frameworks, build tools, or other packages not needed at runtime. Development dependencies often have higher vulnerability rates and larger attack surfaces. The `--no-optional` flag further reduces attack surface by skipping optional dependencies.

Build-Time Security Scanning: Including `npm audit --audit-level=high` in the Dockerfile ensures the build fails if high or critical vulnerabilities are found in dependencies. This "shift-left" approach catches vulnerabilities before images are published to registries or deployed to production. Teams must maintain updated dependencies to keep builds passing.

Cache and Temporary File Cleanup: After installing dependencies, the Dockerfile removes the npm cache (`npm cache clean --force`) and the package-lock.json file. These files serve no purpose at runtime but increase image size and potentially expose information about the build environment. All cleanup operations happen in the same RUN instruction as the installation to ensure they affect the same image layer.

Security Labels: OCI (Open Container Initiative) labels provide standardized metadata that security tools can use to track and manage container images. The `security.contact` label indicates who to notify about security issues, while `security.scanningRequired` can trigger automated security scanning workflows.

Healthcheck Configuration: The HEALTHCHECK instruction enables container orchestrators to verify application health without relying on the application's ability to accept network connections. This is more reliable than port checks and can detect scenarios where the application is running but unable to process requests correctly.

PID 1 and dumb-init: In Linux, PID 1 has special responsibilities including reaping zombie processes and handling signals correctly. Most applications aren't designed to handle these responsibilities. Using dumb-init as the entrypoint ensures signals are handled correctly (enabling graceful shutdown)

and zombie processes are reaped (preventing resource leaks). This is a subtle but important security consideration.

Exec vs Shell Form: The CMD uses exec form (`["node", "server.js"]`) rather than shell form (`CMD node server.js`). Exec form runs the application directly as PID 1 (or PID 2 with dumb-init), while shell form wraps it in a shell process. The shell wrapper interferes with signal handling and creates an unnecessary additional process. Exec form is more efficient and provides better security.

1.5 Docker Daemon Security Configuration

The Docker daemon (dockerd) runs with root privileges and has extensive control over the host system. Securing the daemon is critical because vulnerabilities or misconfigurations can lead to complete host compromise. Docker daemon security involves configuring the daemon itself, securing the Docker socket, and implementing proper access controls.

Docker Socket Security: The Docker socket (`/var/run/docker.sock`) is a Unix socket that provides API access to the Docker daemon. Any process with access to this socket has full control over the Docker daemon and, by extension, the host system. Mounting the Docker socket into containers is extremely dangerous—it's effectively giving the container root access to the host. Despite this, mounting the Docker socket is unfortunately common in Docker-in-Docker scenarios and various dev tools.

Docker Daemon Configuration: The daemon configuration file (`/etc/docker/daemon.json` on Linux) controls numerous security-relevant settings. Proper configuration can significantly improve security posture.

```
{

  "live-restore": true,
  "userland-proxy": false,
  "no-new-privileges": true,

  "icc": false,
  "iptables": true,
  "ip-forward": false,

  "userns-remap": "default",

  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3",
    "labels": "production",
    "env": "os,customer"
```

```
  },

  "authorization-plugins": ["docker-auth-plugin"],

  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ],

  "default-ulimits": {
    "nofile": {
      "Name": "nofile",
      "Hard": 64000,
      "Soft": 64000
    },
    "nproc": {
      "Name": "nproc",
      "Hard": 16384,
      "Soft": 16384
    }
  },

  "exec-opts": ["native.cgroupdriver=systemd"],

  "seccomp-profile": "/etc/docker/seccomp-profile.json",

  "selinux-enabled": true,

  "cgroup-parent": "/docker",

  "default-runtime": "runc",
  "runtimes": {
    "runc": {
      "path": "/usr/bin/runc"
    }
  },

  "bridge": "none",
  "ip": "127.0.0.1",
  "registry-mirrors": [
    "https://registry-mirror.company.internal"
  ],
  "insecure-registries": [],

  "metrics-addr": "127.0.0.1:9323",
  "experimental": false,
  "debug": false
}
```

Detailed Explanation of Security-Critical Configuration Options:

User Namespace Remapping (`userns-remap`): This is perhaps the single most important security configuration for Docker. When enabled, Docker remaps container user IDs to different user IDs on the host. For example, root (UID 0) inside a container might be mapped to UID 100000 on the host, which is an unprivileged user. This dramatically reduces the impact of container escape vulnerabilities an attacker who breaks out of the container still has only unprivileged access to the host system.

User namespace remapping is not enabled by default because it can cause compatibility issues with applications that make assumptions about UIDs, and it requires reconfiguring file ownership for existing volumes. However, the security benefits are so significant that organizations should prioritize enabling it and addressing any resulting compatibility issues.

Default Security Options (`no-new-privileges`): Setting `no-new-privileges: true` prevents processes in containers from gaining additional privileges through setuid binaries, file capabilities, or other privilege escalation mechanisms. This is equivalent to using the `--security-opt no-new-privileges` flag with every container, providing consistent baseline security.

Inter-Container Communication (`icc`): Setting `icc: false` disables direct communication between containers on the default bridge network. Containers can only communicate if explicitly linked or connected to the same user-defined network. This provides better network isolation and prevents lateral movement attacks where a compromised container attempts to access other containers on the same host.

IP Forwarding (`ip-forward`): Disabling IP forwarding (`ip-forward: false`) prevents containers from routing traffic between different networks. This should only be enabled if containers legitimately need to act as routers, which is rare in typical application scenarios.

Authorization Plugins: Docker supports authorization plugins that can enforce fine-grained access control policies for Docker API operations. These plugins can restrict who can create privileged containers, mount sensitive host paths, or access the Docker socket. Authorization plugins are essential for multi-tenant environments or when multiple teams share Docker hosts.

Logging Configuration: Proper logging configuration (`log-driver` and `log-opts`) ensures container logs are captured appropriately for security monitoring. The `max-size` and `max-file` options prevent logs from consuming excessive disk space (which could be used for denial-of-service attacks). Sending logs to external systems ensures they survive container termination and can't be tampered with by attackers.

Storage Driver Security: The `overlay2` storage driver is the recommended storage driver for production use. It provides better performance and security characteristics than older drivers like `aufs` or `devicemapper`. Proper storage driver configuration is important for performance and for preventing storage exhaustion attacks.

Seccomp Profile: The `seccomp-profile` option specifies a custom seccomp profile for all containers. Seccomp profiles restrict which system calls containers can make, providing critical defense against kernel exploits. Custom profiles can be tailored to specific application needs, blocking even more system calls than Docker's default profile.

SELinux Integration: Enabling SELinux (`selinux-enabled: true`) provides mandatory access control that restricts container operations. SELinux policies can prevent containers from accessing sensitive files, making certain network connections, or performing dangerous operations even if the container is compromised. SELinux is particularly important in highly regulated industries.

Resource Limits (`default-ulimits`): Setting default ulimits ensures all containers have reasonable resource limits even if not explicitly specified. This prevents resource exhaustion attacks where containers consume all available file descriptors or create excessive processes. These limits should be tuned based on expected application behavior.

Registry Security: The `registry-mirrors` and `insecure-registries` options control how Docker interacts with container registries. Using registry mirrors can improve performance and reliability, but only trusted mirrors should be configured. The `insecure-registries` array should be empty in production—all registry communication should use HTTPS with proper certificate validation.

Metrics and Debugging: Binding metrics to localhost (`metrics-addr: 127.0.0.1:9323`) prevents unauthorized access to Docker daemon metrics, which could reveal sensitive information about running containers and system resource usage. Similarly, debug mode should be disabled in production as it can expose sensitive information and reduce performance.

This comprehensive Docker daemon configuration provides strong baseline security for Docker hosts. However, configuration alone is insufficient—proper operational practices including regular updates, monitoring, and access control are equally important for maintaining secure Docker deployments.

Chapter 2: Container Image Security Scanning

Vulnerability scanning is a fundamental component of container security, enabling organizations to identify and remediate known security issues before they reach production. Container images frequently contain vulnerable software components, base OS packages, language runtimes, application dependencies that attackers can exploit. Comprehensive image scanning integrated into CI/CD pipelines ensures vulnerabilities are caught early in the software lifecycle, where they're cheapest and easiest to fix.

2.1 Understanding Container Vulnerability Scanning

Container vulnerability scanning tools analyze image layers to identify installed software packages and their versions, then compare this inventory against databases of known vulnerabilities (primarily the National Vulnerability Database maintained by NIST). The scanning process extracts package manifests from various package managers (apt, yum, apk, pip, npm, etc.), identifies all installed versions, and correlates them with CVE (Common Vulnerabilities and Exposures) records.

Types of Vulnerabilities Detected:

Container scanners typically detect several categories of vulnerabilities:

1. ***Operating System Package Vulnerabilities:*** These are vulnerabilities in system packages installed via OS package managers (apt, yum, apk, etc.). Base images often contain numerous packages, many of which may have known vulnerabilities. OS package vulnerabilities are usually well-documented in distributions' security advisories and can often be fixed by updating the base image or installing package updates.
2. ***Application Dependency Vulnerabilities:*** These affect packages installed via language-specific package managers (pip, npm, gem, maven, etc.). Application dependencies are frequently the source of critical vulnerabilities because they're numerous, deeply nested (dependencies of dependencies), and applications often use outdated versions. Tools like Trivy and Grype can detect vulnerabilities in application dependencies across multiple languages.
3. ***Embedded Binaries and Libraries:*** Some images contain compiled binaries or shared libraries that aren't managed by any package manager. These can be challenging to scan because there's no manifest indicating what they are or what version they are. Advanced scanners use techniques like binary analysis and signature matching to identify these components.
4. ***Configuration Issues:*** Beyond code vulnerabilities, some scanners detect security misconfigurations like weak cryptographic algorithms, insecure protocol versions, or dangerous default configurations. These issues don't have CVE identifiers but can be equally serious.

Vulnerability Severity Ratings:

Vulnerabilities are typically rated using the Common Vulnerability Scoring System (CVSS), which assigns scores from 0-10 based on factors like exploitability, impact, and scope. Scores are generally categorized as:

- Critical (9.0-10.0): Remotely exploitable with severe impact
- High (7.0-8.9): Significant security risk requiring prompt attention
- Medium (4.0-6.9): Moderate risk that should be addressed
- Low (0.1-3.9): Minor risk with limited impact
- Informational: Not a vulnerability but noteworthy security information

Organizations typically prioritize remediation based on severity, focusing first on critical and high-severity vulnerabilities in internet-facing applications, then working through medium and low severity issues based on risk assessments.

False Positives and Vulnerability Context:

Vulnerability scanners inevitably produce false positives—reported vulnerabilities that don't actually affect the application. This happens because:

- The vulnerable code path isn't accessible in the application's deployment context
- The vulnerability requires specific conditions that don't exist in the environment
- The scanner incorrectly identifies a package version
- The vulnerability was already patched through backports or vendor-specific fixes

Managing false positives is a significant operational challenge. Organizations need processes for triaging scan results, verifying exploitability, documenting false positives, and creating exceptions for issues that can't be fixed immediately.

2.2 Trivy: Comprehensive Vulnerability Scanner

Trivy is an open-source vulnerability scanner developed by Aqua Security that has become the de facto standard for container image scanning. Trivy detects vulnerabilities in OS packages, application dependencies, Infrastructure as Code misconfigurations, and more. Its comprehensive detection capabilities, fast scanning speed, and ease of use make it an excellent choice for CI/CD integration.

Trivy Architecture and Capabilities:

Trivy operates by analyzing filesystem contents to identify installed packages and comparing them against its vulnerability database. The database is regularly updated from multiple sources including the National Vulnerability

Database, distribution security advisories (Debian Security Tracker, Red Hat Security Advisories, etc.), and language-specific security databases.

Key Trivy capabilities include:

- Multi-layer Scanning: Trivy analyzes each layer in a Docker image independently, identifying which layer introduced each vulnerability. This helps developers understand where vulnerabilities come from and how to fix them.
- Multiple Package Manager Support: Trivy supports vulnerability detection for OS packages (Alpine, Debian, Ubuntu, Red Hat, CentOS, Amazon Linux, SUSE, Oracle Linux) and application dependencies (npm, yarn, pip, pipenv, composer, bundler, cargo, go modules, maven, gradle, NuGet).
- SBOM Generation: Trivy can generate Software Bill of Materials in various formats (CycloneDX, SPDX), which are essential for supply chain security and vulnerability management.
- License Detection: Beyond vulnerabilities, Trivy can detect software licenses, helping organizations ensure compliance with licensing requirements.
- IaC Scanning: Trivy can scan Kubernetes manifests, Docker files, Terraform configurations, and other Infrastructure as Code for security misconfigurations.
- Secret Detection: Trivy can find accidentally committed secrets like API keys, passwords, and tokens in images.

Basic Trivy Usage Examples:

```
# ================================================================================
# TRIVY BASIC IMAGE SCANNING
# ================================================================================

# Simple scan of an image showing all vulnerabilities
trivy image nginx:latest

# Scan with severity filtering - only show HIGH and CRITICAL
# This is the most common usage for CI/CD gates
trivy image --severity HIGH,CRITICAL nginx:latest

# Scan and output to JSON for automated processing
# JSON output can be consumed by security dashboards and tracking systems
trivy image --format json --output scan-results.json nginx:latest

# Scan with exit code control for CI/CD
# Fails (exit 1) if HIGH or CRITICAL vulnerabilities found
# This causes CI/CD pipelines to fail, preventing vulnerable images from
deployment
trivy image --exit-code 1 --severity HIGH,CRITICAL nginx:latest
```

```
# ================================================================================
# TRIVY ADVANCED OPTIONS
# ================================================================================

# Scan specific image layers to identify where vulnerabilities were introduced
# Useful for understanding which Dockerfile instruction created the problem
trivy image --format json nginx:latest | jq '.Results[].Vulnerabilities[] |
select(.PkgName == "openssl")'

# Generate SBOM (Software Bill of Materials)
# CycloneDX and SPDX formats are industry standards for SBOM
trivy image --format cyclonedx --output nginx-sbom.json nginx:latest

# Scan for specific vulnerability IDs
# Useful when you need to verify if a specific CVE affects your images
trivy image --vuln-type os --severity HIGH --ignore-unfixed \
  nginx:latest | grep CVE-2024-1234

# Scan with custom policy
# Policies can define organization-specific rules beyond just vulnerabilities
trivy image --config trivy-config.yaml nginx:latest

# ================================================================================
# TRIVY SCANNING DOCKER FILES
# ================================================================================

# Scan Dockerfile for best practice violations
# Identifies security issues in Dockerfile instructions before building
trivy config Dockerfile

# Scan entire directory of Infrastructure as Code
# Useful for scanning Kubernetes manifests, Terraform configs, etc.
trivy config ./k8s-manifests/

# ================================================================================
# TRIVY KUBERNETES CLUSTER SCANNING
# ================================================================================

# Scan running Kubernetes cluster for vulnerabilities
# Analyzes all running container images in the cluster
trivy k8s --report summary cluster

# Scan specific Kubernetes resources
trivy k8s --report all deploy/nginx

# ================================================================================
# TRIVY CI/CD INTEGRATION EXAMPLE
# ================================================================================

# Typical CI/CD scanning command with multiple checks
```

```
# This command:
# 1. Scans for HIGH/CRITICAL vulnerabilities only
# 2. Ignores vulnerabilities without fixes (reduces noise)
# 3. Fails the build if issues found
# 4. Generates JSON report for security dashboards
# 5. Timeout after 10 minutes (prevents hanging in CI)
trivy image \
  --severity HIGH,CRITICAL \
  --ignore-unfixed \
  --exit-code 1 \
  --format json \
  --output /reports/trivy-scan.json \
  --timeout 10m \
  myapp:${CI_COMMIT_SHA}
```

Detailed Explanation of Trivy Options:

Severity Filtering (`--severity`): The severity filter is crucial for managing false positives and focusing on actionable issues. In practice, organizations typically use `--severity HIGH,CRITICAL` for CI/CD gates that block deployments, while running periodic scans with all severities for comprehensive vulnerability tracking. Blocking on MEDIUM severity can create too much friction in development processes unless the organization has mature vulnerability management.

Exit Code Control (`--exit-code`): Setting `--exit-code 1` causes Trivy to return a non-zero exit code when vulnerabilities matching the severity criteria are found. CI/CD systems interpret non-zero exit codes as failures, automatically blocking the pipeline. This prevents vulnerable images from being published to registries or deployed to production.

Ignoring Unfixed Vulnerabilities (`--ignore-unfixed`): The `--ignore-unfixed` flag tells Trivy to suppress vulnerabilities that don't have available fixes. This significantly reduces noise in scan results because many reported vulnerabilities (especially in base images) don't have patches available yet. However, organizations should still track unfixed vulnerabilities because they might need to switch base images or implement compensating controls.

JSON Output (`--format json`): JSON format output enables automation and integration with security tools. JSON results can be:

- Sent to vulnerability management platforms
- Stored in security dashboards
- Analyzed with tools like jq
- Tracked over time to measure vulnerability trends
- Used for compliance reporting

Timeout Configuration (`--timeout`): Setting appropriate timeouts prevents CI/CD pipelines from hanging indefinitely if Trivy encounters problems. Default timeouts might be too short for very large images or slow networks. Production systems should use timeouts of 10-15 minutes for typical images, longer for exceptionally large images.

2.3 Advanced Trivy Configuration and Policies

While basic Trivy scanning catches many vulnerabilities, production environments benefit from advanced configuration that tailors scanning to organizational policies and reduces false positives.

Trivy Configuration File:

```
# trivy-config.yaml - Advanced Trivy Configuration
# This configuration demonstrates production-ready settings for enterprise use

# Cache settings for faster subsequent scans
cache:
  # Cache backend - Redis for distributed caching in CI/CD
  backend: "redis://redis-server:6379"
  # TTL for vulnerability database cache
  ttl: 24h

# Database configuration
db:
  # Skip database update if recent (for faster scans in CI/CD)
  skip-update: false
  # Database path (can be shared across runners)
  repository: "ghcr.io/aquasecurity/trivy-db"

# Vulnerability settings
vulnerability:
  # Vulnerability types to scan
  type:
    - os        # Operating system packages
    - library   # Language-specific libraries
  # Severity levels to report
  severity:
    - CRITICAL
    - HIGH
    - MEDIUM
  # Ignore vulnerabilities without fixes
  ignore-unfixed: true

# Secret scanning configuration
secret:
  # Enable secret scanning
  enabled: true
```

```yaml
  # Custom secret patterns
  config: ".trivy/secret-patterns.yaml"

# License scanning configuration
license:
  # Enable license scanning
  enabled: true
  # Forbidden license types (fail build if found)
  forbidden:
    - GPL-3.0
    - AGPL-3.0
  # Restricted licenses (warn but don't fail)
  restricted:
    - LGPL-3.0

# Scan settings
scan:
  # Security checks to perform
  security-checks:
    - vuln      # Vulnerability scanning
    - config    # Configuration scanning
    - secret    # Secret scanning
    - license   # License scanning

# Ignore specific vulnerabilities (exception management)
ignores:
  # Ignore specific CVEs with justification
  - id: CVE-2023-12345
    # Justification required for audit trail
    reason: "False positive - vulnerable code path not accessible in our
deployment"
    # Expiration date - forces periodic review
    expired-at: "2024-12-31"

  - id: CVE-2023-67890
    reason: "Waiting for upstream patch - compensating controls in place (WAF
rules block exploit)"
    expired-at: "2024-06-30"

# Policy configuration for custom rules
policy:
  # Path to custom Rego policies
  policy-bundle: ".trivy/policies/"
  # Namespace for policies
  namespaces:
    - "custom.policies"

# Reporting settings
report:
  # Report format
  format: "sarif"  # SARIF format for GitHub Security
```

```yaml
  # Template for custom report formatting
  template: "@trivy/html.tpl"
  # Dependency tree in report
  dependency-tree: true
  # List all packages even without vulnerabilities
  list-all-pkgs: false

# Timeout settings
timeout:
  # Overall scan timeout
  timeout: 15m
  # Timeout for image pulling
  image-timeout: 10m

# Registry authentication (for private registries)
registry:
  # Credentials for private registries
  credentials:
    - registry: "myregistry.company.com"
      username: "scanner-service-account"
      # Use environment variable for password
      password: "${REGISTRY_PASSWORD}"

# Kubernetes scanning settings
kubernetes:
  # Components to scan in cluster
  components:
    - workload      # Deployments, StatefulSets, etc.
    - rbac          # Roles and RoleBindings
    - config        # ConfigMaps and Secrets

  # Namespaces to include/exclude
  include-namespaces:
    - production
    - staging

  exclude-namespaces:
    - kube-system
    - kube-public

# Exit code configuration
exit-code:
  # Exit code when vulnerabilities found
  found: 1
  # Exit code when errors occur
  error: 2
  # Severity threshold for exit code
  severity-threshold: HIGH
```

Custom Exception Management:

The `ignores` section in the configuration file demonstrates a critical capability: managing false positives and exceptions in a structured, auditable way. Each exception includes:

1. ***CVE Identifier:*** The specific vulnerability being ignored
2. ***Justification:*** A required human-readable explanation of why this exception is acceptable
3. ***Expiration Date:*** Forces periodic review to ensure exceptions don't become permanent

This approach provides auditability and ensures exceptions are reviewed regularly. The expiration date is critical—many organizations find that exceptions become permanent when there's no forcing function for review.

License Scanning Integration:

The license configuration demonstrates another important use case for Trivy beyond vulnerability scanning. Many organizations have restrictions on which software licenses they can use, particularly around copyleft licenses like GPL that might require releasing source code. Trivy's license scanning can automatically detect forbidden licenses and fail builds, ensuring compliance.

Policy-as-Code with Rego:

Trivy supports custom policies written in Rego (the policy language used by Open Policy Agent). This enables organizations to define custom security requirements beyond just vulnerability thresholds. Examples of custom policies might include:

- Requiring all images to be from approved registries
- Enforcing specific base image versions
- Requiring images to be signed
- Checking for required security labels
- Validating image age (preventing deployment of stale images)

Integration with Security Tools:

Trivy supports multiple output formats that enable integration with various security tools:

- SARIF: For GitHub Security integration
- JUnit: For CI/CD systems that understand JUnit format
- Template: For custom formatting using Go templates
- CycloneDX/SPDX: For SBOM management systems

2.4 CI/CD Integration Patterns

Integrating vulnerability scanning into CI/CD pipelines is essential for catching security issues early. The goal is to provide fast feedback to developers while preventing vulnerable images from reaching production. Effective integration requires balancing security rigor with development velocity.

GitLab CI Example:

```
# .gitlab-ci.yml - Comprehensive Security-Integrated CI/CD Pipeline
# This pipeline demonstrates production-ready security integration

variables:
  # Docker image for building
  DOCKER_BUILD_IMAGE: docker:24.0
  # Trivy scanner version (pinned for consistency)
  TRIVY_VERSION: "0.48.0"
  # Registry configuration
  REGISTRY: registry.company.com
  IMAGE_NAME: "${REGISTRY}/${CI_PROJECT_PATH}"
  # Security thresholds
  VULN_SEVERITY_THRESHOLD: "HIGH,CRITICAL"
  # Build configuration
  DOCKER_BUILDKIT: "1"  # Enable BuildKit for better performance

# Pipeline stages
stages:
  - build
  - security-scan
  - push
  - deploy

# ============================================================================
# BUILD STAGE
# ============================================================================
build:
  stage: build
  image: ${DOCKER_BUILD_IMAGE}
  services:
    - docker:24.0-dind
  before_script:
    # Login to registry
    - echo "$CI_REGISTRY_PASSWORD" | docker login -u "$CI_REGISTRY_USER"
--password-stdin "$REGISTRY"
  script:
    # Build with cache to speed up builds
    - |
      docker build \
        --build-arg BUILDKIT_INLINE_CACHE=1 \
        --cache-from "${IMAGE_NAME}:latest" \
        --tag "${IMAGE_NAME}:${CI_COMMIT_SHA}" \
        --tag "${IMAGE_NAME}:${CI_COMMIT_REF_SLUG}" \
```

```
        .

    # Save image as artifact for subsequent stages
    - docker save "${IMAGE_NAME}:${CI_COMMIT_SHA}" -o image.tar

  artifacts:
    # Pass image to next stages
    paths:
      - image.tar
    expire_in: 1 hour

  # Only run for merge requests and main branch
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
    - if: '$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH'

# ============================================================================
# SECURITY SCANNING STAGE
# ============================================================================

# Vulnerability Scanning Job
vulnerability-scan:
  stage: security-scan
  image: aquasec/trivy:${TRIVY_VERSION}
  before_script:
    # Load image from build stage
    - docker load -i image.tar
    # Update Trivy vulnerability database
    - trivy image --download-db-only

  script:
    # Scan image for vulnerabilities
    - |
      trivy image \
        --severity ${VULN_SEVERITY_THRESHOLD} \
        --ignore-unfixed \
        --no-progress \
        --exit-code 0 \
        --format json \
        --output vulnerability-scan.json \
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"

    # Generate HTML report for human review
    - |
      trivy image \
        --severity HIGH,CRITICAL,MEDIUM,LOW \
        --format template \
        --template "@trivy/html.tpl" \
        --output vulnerability-report.html \
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"
```

```
    # Check against security policy and fail if issues found
    - |
      trivy image \
        --severity ${VULN_SEVERITY_THRESHOLD} \
        --ignore-unfixed \
        --exit-code 1 \
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"

  artifacts:
    reports:
      # GitLab parses these reports and displays in MR UI
      dependency_scanning: vulnerability-scan.json
    paths:
      # HTML report for downloading
      - vulnerability-report.html
    expire_in: 30 days

  # Allow manual continuation if scan fails (for emergency deployments)
  # Remove this in production for strict enforcement
  allow_failure: false

# Configuration Scanning Job
config-scan:
  stage: security-scan
  image: aquasec/trivy:${TRIVY_VERSION}
  script:
    # Scan Dockerfile for misconfigurations
    - trivy config --exit-code 1 --severity HIGH,CRITICAL Dockerfile

    # Scan Kubernetes manifests if they exist
    - |
      if [ -d "k8s" ]; then
        trivy config --exit-code 1 --severity HIGH,CRITICAL k8s/
      fi

  # Run in parallel with vulnerability scan
  needs: []

# Secret Scanning Job
secret-scan:
  stage: security-scan
  image: aquasec/trivy:${TRIVY_VERSION}
  before_script:
    - docker load -i image.tar

  script:
    # Scan image for accidentally committed secrets
    - |
      trivy image \
        --scanners secret \
        --exit-code 1 \
```

```
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"

  # Fail pipeline if secrets found
  allow_failure: false

# SBOM Generation Job
sbom-generate:
  stage: security-scan
  image: aquasec/trivy:${TRIVY_VERSION}
  before_script:
    - docker load -i image.tar

  script:
    # Generate CycloneDX SBOM
    - |
      trivy image \
        --format cyclonedx \
        --output sbom-cyclonedx.json \
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"

    # Generate SPDX SBOM
    - |
      trivy image \
        --format spdx-json \
        --output sbom-spdx.json \
        "${IMAGE_NAME}:${CI_COMMIT_SHA}"

  artifacts:
    paths:
      - sbom-cyclonedx.json
      - sbom-spdx.json
    expire_in: 90 days

  # SBOM generation shouldn't fail pipeline
  allow_failure: true

# ============================================================================
# PUSH STAGE - Only if security scans pass
# ============================================================================
push:
  stage: push
  image: ${DOCKER_BUILD_IMAGE}
  services:
    - docker:24.0-dind
  before_script:
    - docker load -i image.tar
    - echo "$CI_REGISTRY_PASSWORD" | docker login -u "$CI_REGISTRY_USER"
--password-stdin "$REGISTRY"

  script:
    # Push image with commit SHA tag
```

```
    - docker push "${IMAGE_NAME}:${CI_COMMIT_SHA}"

    # Update latest tag on main branch
    - |
      if [ "$CI_COMMIT_BRANCH" == "$CI_DEFAULT_BRANCH" ]; then
        docker tag "${IMAGE_NAME}:${CI_COMMIT_SHA}" "${IMAGE_NAME}:latest"
        docker push "${IMAGE_NAME}:latest"
      fi

  # Only push if security scans pass
  needs:
    - build
    - vulnerability-scan
    - config-scan
    - secret-scan

  rules:
    - if: '$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH'

# ============================================================================
# DEPLOY STAGE - Production deployment
# ============================================================================
deploy-production:
  stage: deploy
  image: bitnami/kubectl:latest
  script:
    # Update Kubernetes deployment with new image
    - |
      kubectl set image deployment/myapp \
        myapp="${IMAGE_NAME}:${CI_COMMIT_SHA}" \
        --namespace=production \
        --record

    # Wait for rollout to complete
    - |
      kubectl rollout status deployment/myapp \
        --namespace=production \
        --timeout=5m

  # Manual approval required for production
  when: manual

  needs:
    - push

  only:
    - main

  environment:
    name: production
    url: https://app.company.com
```

CI/CD Integration Best Practices:

1. ***Fail Fast:*** Security scans should run early in the pipeline, before expensive deployment steps. If an image fails security checks, there's no point in pushing it to registries or attempting deployment. The example pipeline runs all security scans in parallel after the build, blocking the push stage if any scan fails.

2. ***Multiple Security Checks:*** The pipeline includes multiple types of security scanning:

- Vulnerability Scanning: Checks for known CVEs in dependencies
- Configuration Scanning: Validates Dockerfiles and manifests
- Secret Scanning: Detects accidentally committed credentials
- SBOM Generation: Creates software bill of materials for tracking

This defense-in-depth approach catches different categories of security issues.

3. ***Progressive Security:*** Not all security checks need to fail the pipeline. The example uses `allow_failure: true` for SBOM generation because SBOM creation failures shouldn't block deployments, but the generated SBOMs are still valuable for tracking. Teams can tune which checks are blocking vs. warning based on their risk tolerance.

4. ***Security Reports as Artifacts:*** Scan results are saved as pipeline artifacts and GitLab-native reports. This enables:

- Viewing vulnerability details in merge request UI
- Downloading detailed reports for investigation
- Historical tracking of vulnerabilities over time
- Compliance documentation

5. ***Emergency Bypass Procedures:*** While strong security gates are important, teams need escape hatches for emergency situations. The pipeline uses manual approval for production deployments, providing a checkpoint where security exceptions can be granted if absolutely necessary.

6. ***Automated Remediation Workflows:*** Some organizations extend these pipelines with automatic fix creation. When vulnerabilities are found, the pipeline can automatically create merge requests that update dependencies or rebuild from newer base images. This reduces the manual effort required for vulnerability remediation.

This comprehensive approach to CI/CD security integration ensures vulnerabilities are caught early, provides visibility into security posture, and prevents insecure images from reaching production while maintaining development velocity.

Chapter 3: Kubernetes Security Architecture and Pod Security Standards

Kubernetes has evolved from a simple container orchestrator into a comprehensive platform for managing cloud-native applications at scale. This evolution brought immense power and flexibility, but also introduced significant security complexity. Understanding Kubernetes security architecture is essential for security professionals, platform engineers, and developers working with container orchestration. This chapter explores Kubernetes security fundamentals, focusing on the control plane, Pod Security Standards, and admission control mechanisms.

3.1 Kubernetes Control Plane Security

The Kubernetes control plane consists of several critical components that manage the cluster state and orchestrate workload deployment. Securing these components is paramount because compromise of control plane components can lead to complete cluster takeover. The control plane includes the API server, etcd datastore, scheduler, controller manager, and cloud controller manager. Each component requires specific security configurations to prevent unauthorized access and maintain cluster integrity.

API Server Security - The Gateway to Kubernetes:

The Kubernetes API server (kube-apiserver) is the primary interface for all cluster operations. All kubectl commands, CI/CD integrations, and internal component communications pass through the API server. This central role makes it the most critical component to secure. API server security involves multiple layers:

Authentication determines who can access the API. Kubernetes supports multiple authentication methods including X.509 client certificates, bearer tokens, authentication proxies, and OpenID Connect (OIDC). Production clusters should use OIDC integration with corporate identity providers, enabling centralized authentication and single sign-on capabilities. Certificate-based authentication should be reserved for service accounts and infrastructure components.

Authorization determines what authenticated users can do. Kubernetes provides several authorization modes, with Role-Based Access Control (RBAC) being the most common and powerful. RBAC allows defining fine-grained permissions based on roles and bindings. Production clusters should enable only RBAC authorization mode and disable permissive modes like AlwaysAllow or ABAC.

Admission Control provides the last line of defense, intercepting requests after authentication and authorization but before objects are persisted to etcd. Admission controllers can validate, mutate, or reject requests based on policies. Critical admission controllers for security include PodSecurityAdmission, LimitRanger, ResourceQuota, and various policy engines.

Here's a comprehensive example of securing the API server through configuration and network controls:

```
# ================================================================================
# KUBERNETES API SERVER SECURE CONFIGURATION
# ================================================================================
# File: /etc/kubernetes/manifests/kube-apiserver.yaml
# This configuration demonstrates production-grade API server hardening

apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
  labels:
    component: kube-apiserver
    tier: control-plane
spec:
  containers:
  - name: kube-apiserver
    image: registry.k8s.io/kube-apiserver:v1.28.0

    command:
    - kube-apiserver

    # ============================================================================
    # AUTHENTICATION CONFIGURATION
    # ============================================================================

    # Client certificate authentication for service accounts
    - --client-ca-file=/etc/kubernetes/pki/ca.crt

    # TLS certificates for API server
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key

    # Enable OIDC authentication for users (integrate with corporate IdP)
```

```
    - --oidc-issuer-url=https://idp.company.com
    - --oidc-client-id=kubernetes
    - --oidc-ca-file=/etc/kubernetes/pki/oidc-ca.crt
    - --oidc-username-claim=email
    - --oidc-groups-claim=groups
    # OIDC username prefix prevents conflicts with service account names
    - --oidc-username-prefix=oidc:
    - --oidc-groups-prefix=oidc:

    # Service account token authentication
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-account-signing-key-file=/etc/kubernetes/pki/sa.key
    - --service-account-issuer=https://kubernetes.default.svc.cluster.local
    # Require bound service account tokens (more secure)
    - --service-account-extend-token-expiration=true

    # Anonymous authentication disabled (critical security setting)
    - --anonymous-auth=false

    # ================================================================================
    # AUTHORIZATION CONFIGURATION
    # ================================================================================

    # Enable RBAC only (disable permissive modes)
    - --authorization-mode=Node,RBAC

    # Webhook authorization for custom policies (optional)
    # -
--authorization-webhook-config-file=/etc/kubernetes/webhooks/authz-webhook.yaml
    # - --authorization-webhook-cache-authorized-ttl=5m
    # - --authorization-webhook-cache-unauthorized-ttl=30s

    # ================================================================================
    # ADMISSION CONTROL CONFIGURATION
    # ================================================================================

    # Critical admission controllers for security
    # Order matters - controllers run in sequence
    - --enable-admission-plugins=
        NodeRestriction,           # Prevents nodes from modifying other nodes
        PodSecurityAdmission,      # Enforces Pod Security Standards
        LimitRanger,               # Enforces resource limits
        ResourceQuota,             # Enforces quota constraints
        ServiceAccount,            # Ensures service accounts are properly
configured
        DefaultStorageClass,       # Sets default storage class
        DefaultTolerationSeconds,  # Sets default tolerations
        MutatingAdmissionWebhook,  # Enables mutating webhooks
        ValidatingAdmissionWebhook # Enables validating webhooks

    # Disable dangerous admission controllers
```

```
    - --disable-admission-plugins=AlwaysAdmit,NamespaceAutoProvision

    # Pod Security Admission configuration
    -
--admission-control-config-file=/etc/kubernetes/admission/admission-config.yaml

    # ================================================================================
    # AUDIT LOGGING CONFIGURATION
    # ================================================================================

    # Enable comprehensive audit logging
    - --audit-policy-file=/etc/kubernetes/audit/audit-policy.yaml
    - --audit-log-path=/var/log/kubernetes/audit/audit.log
    - --audit-log-maxage=30        # Retain for 30 days
    - --audit-log-maxbackup=10     # Keep 10 backup files
    - --audit-log-maxsize=100      # Max 100MB per file
    # Audit log format (json for parsing)
    - --audit-log-format=json
    # Send audit events to webhook (SIEM integration)
    - --audit-webhook-config-file=/etc/kubernetes/audit/webhook-config.yaml
    - --audit-webhook-batch-max-wait=5s

    # ================================================================================
    # ENCRYPTION AT REST (etcd encryption)
    # ================================================================================

    # Encrypt secrets and other sensitive data in etcd
    -
--encryption-provider-config=/etc/kubernetes/encryption/encryption-config.yaml

    # ================================================================================
    # NETWORK AND TLS CONFIGURATION
    # ================================================================================

    # Bind to specific IP (not 0.0.0.0 for better security)
    - --bind-address=10.0.1.10
    # Secure port for all communications
    - --secure-port=6443
    # Disable insecure port completely
    - --insecure-port=0

    # TLS version restrictions (only TLS 1.2+)
    - --tls-min-version=VersionTLS12
    # Strong cipher suites only
    - --tls-cipher-suites=
        TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
        TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
        TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
        TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

    # ================================================================================
```

```
    # ETCD CONNECTION SECURITY
    # ================================================================================

    # Connect to etcd securely
    - --etcd-servers=https://127.0.0.1:2379
    - --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

    # ================================================================================
    # ADDITIONAL SECURITY SETTINGS
    # ================================================================================

    # Enable profiling only on localhost (debug information leakage)
    - --profiling=false

    # Request timeout to prevent DoS
    - --request-timeout=60s

    # Maximum requests in flight to prevent resource exhaustion
    - --max-requests-inflight=400
    - --max-mutating-requests-inflight=200

    # Kubelet certificate validation
    - --kubelet-certificate-authority=/etc/kubernetes/pki/ca.crt
    -
--kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    # CRITICAL: Verify kubelet serving certificates
    - --kubelet-https=true

    # Feature gates (enable/disable features)
    - --feature-gates=
        RotateKubeletServerCertificate=true,  # Auto-rotate kubelet certs
        ServiceAccountIssuerDiscovery=true     # OIDC discovery for SAs

    # ================================================================================
    # RESOURCE LIMITS FOR API SERVER CONTAINER
    # ================================================================================

    resources:
      requests:
        cpu: "250m"
        memory: "512Mi"
      limits:
        cpu: "1000m"
        memory: "2Gi"

    # ================================================================================
    # VOLUME MOUNTS FOR CONFIGURATION AND PKI
    # ================================================================================
```

```
    volumeMounts:
    - mountPath: /etc/kubernetes/pki
      name: k8s-certs
      readOnly: true
    - mountPath: /etc/kubernetes/admission
      name: admission-config
      readOnly: true
    - mountPath: /etc/kubernetes/audit
      name: audit-config
      readOnly: true
    - mountPath: /etc/kubernetes/encryption
      name: encryption-config
      readOnly: true
    - mountPath: /var/log/kubernetes/audit
      name: audit-logs

  # ================================================================================
  # HOST CONFIGURATION
  # ================================================================================

  hostNetwork: true
  priorityClassName: system-node-critical

  volumes:
  - hostPath:
      path: /etc/kubernetes/pki
      type: DirectoryOrCreate
    name: k8s-certs
  - hostPath:
      path: /etc/kubernetes/admission
      type: DirectoryOrCreate
    name: admission-config
  - hostPath:
      path: /etc/kubernetes/audit
      type: DirectoryOrCreate
    name: audit-config
  - hostPath:
      path: /etc/kubernetes/encryption
      type: DirectoryOrCreate
    name: encryption-config
  - hostPath:
      path: /var/log/kubernetes/audit
      type: DirectoryOrCreate
    name: audit-logs
```

Comprehensive Analysis of API Server Security Configuration:

This configuration demonstrates production-grade API server hardening addressing multiple attack vectors:

Authentication Architecture: The configuration implements multi-method authentication suited for different use cases. OIDC integration with corporate identity providers enables human users to authenticate using their standard corporate credentials, providing centralized authentication management, automated account lifecycle management, and integration with existing security tools. Service accounts use certificate-based authentication, which provides strong cryptographic identity for workloads. The critical setting `--anonymous-auth=false` prevents unauthenticated requests entirely, eliminating a common attack vector where anonymous users probe the API for information disclosure.

Authorization Hardening: Setting `--authorization-mode=Node,RBAC` enables only secure authorization modes. The Node authorization mode allows kubelets to access only resources for pods scheduled on their nodes, implementing least-privilege principles for node-to-API-server communication. RBAC authorization requires explicit permission grants for all operations, replacing permissive legacy modes. Notably absent from this configuration are `AlwaysAllow` (grants all requests), `ABAC` (deprecated attribute-based access control), and `Webhook` authorization modes that might introduce security risks if misconfigured.

Admission Control Defense-in-Depth: The admission controller configuration creates multiple security checkpoints for API requests. `NodeRestriction` prevents nodes from modifying Node objects they don't own, limiting the blast radius of compromised nodes. `PodSecurityAdmission` enforces Pod Security Standards cluster-wide, preventing deployment of insecure pod configurations. `LimitRanger` and `ResourceQuota` prevent resource exhaustion attacks by enforcing limits on resource consumption. The webhook admission controllers enable custom policy enforcement through tools like OPA Gatekeeper or Kyverno, allowing organizations to implement business-specific security policies.

Audit Logging Configuration: Comprehensive audit logging configured through `--audit-policy-file` and related flags creates an immutable record of all API operations for security monitoring, compliance, and forensics. The configuration sends audit logs to both local files (for immediate access) and webhooks (for SIEM integration), ensuring logs survive cluster failures and cannot be easily tampered with by attackers. Audit logs capture who did what, when, and from where, enabling security teams to detect suspicious activity, investigate incidents, and demonstrate compliance with regulatory requirements.

Encryption at Rest: The `--encryption-provider-config` flag enables encryption of sensitive data in etcd using provider plugins. This ensures secrets, ConfigMaps, and other potentially sensitive data are encrypted when stored in the etcd database. Without encryption at rest, anyone with etcd access

(including etcd backups) can read all cluster secrets in plaintext. The encryption configuration supports multiple providers including aescbc, aesgcm, secretbox, and KMS (Key Management Service) integration with cloud providers or HashiCorp Vault for enterprise key management.

Network Security Hardening: Several network-related settings improve security posture. Binding to a specific IP address (`--bind-address`) rather than 0.0.0.0 limits which network interfaces can receive API requests, reducing attack surface. Setting `--insecure-port=0` completely disables the insecure HTTP port, ensuring all communications use TLS. The TLS configuration specifies minimum version (TLS 1.2) and restricts cipher suites to strong algorithms, preventing downgrade attacks and ensuring forward secrecy.

Resource Limit Protection: Request throttling through `--max-requests-inflight` and `--max-mutating-requests-inflight` prevents API server resource exhaustion during DoS attacks or accidental request storms. These limits ensure the API server remains responsive to critical operations even under heavy load. The request timeout (`--request-timeout`) prevents long-running requests from tying up API server resources indefinitely.

Kubelet Security: The kubelet-related settings establish secure communication between the API server and kubelets. Most critically, `--kubelet-https=true` combined with certificate verification settings ensures the API server validates kubelet identity before sending sensitive workload information. This prevents man-in-the-middle attacks where malicious actors impersonate kubelets to receive pod specifications containing secrets.

3.2 Pod Security Standards (PSS) - Modern Pod Security

Pod Security Standards represent Kubernetes' modern approach to pod security, replacing the deprecated PodSecurityPolicy (PSP) admission controller. PSS defines three security profiles—Privileged, Baseline, and Restricted—that represent different levels of security hardening. Unlike PSP's complex policy definitions, PSS provides predefined, well-documented security postures that organizations can apply cluster-wide or per-namespace.

Understanding the Three Security Profiles:

Privileged Profile: The Privileged profile is completely unrestricted, allowing any pod configuration including those with full host access, privileged containers, and dangerous capabilities. This profile should be used sparingly, typically only for system-level workloads like CNI plugins, CSI drivers, or privileged monitoring agents that genuinely require host access. Using the Privileged profile for application workloads represents a significant security risk.

Baseline Profile: The Baseline profile prevents the most dangerous pod configurations while remaining broadly compatible with existing workloads. It blocks privileged containers, host namespace access, and other configurations known to enable easy container escape or host compromise. The Baseline profile serves as a reasonable starting point for organizations beginning their pod security journey, providing meaningful security improvements without requiring extensive application modifications.

Restricted Profile: The Restricted profile implements defense-in-depth security controls aligned with current pod hardening best practices. It enforces running as non-root, dropping all capabilities, using read-only root filesystems, and preventing privilege escalation. This profile represents the gold standard for pod security and should be the goal for all application workloads in production environments. However, achieving Restricted compliance often requires application modifications, particularly for legacy applications not designed with container security in mind.

Here's a comprehensive example of implementing Pod Security Standards across a cluster:

```
# ================================================================================
# POD SECURITY ADMISSION CONFIGURATION
# ================================================================================
# File: /etc/kubernetes/admission/admission-config.yaml
# This configuration enforces Pod Security Standards cluster-wide

apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration

    # Default security profiles for namespaces without labels
    defaults:
      # Enforce Baseline by default (good starting point)
      enforce: "baseline"
      enforce-version: "latest"

      # Warn on Restricted violations (encourage best practices)
      warn: "restricted"
      warn-version: "latest"

      # Audit Restricted violations for compliance tracking
      audit: "restricted"
      audit-version: "latest"
```

```yaml
    # Exemptions - specific workloads or namespaces exempt from policies
    exemptions:
      # Exempt specific usernames (typically admins)
      usernames:
        - "system:serviceaccount:kube-system:privileged-sa"

      # Exempt specific runtime classes
      runtimeClasses: []

      # Exempt specific namespaces (system namespaces)
      namespaces:
        - kube-system      # System components
        - kube-node-lease  # Node lease system
        - kube-public      # Public cluster info
---
# ================================================================================
# NAMESPACE SECURITY PROFILE CONFIGURATION
# ================================================================================
# Apply security profiles at namespace level using labels

# Example 1: Production namespace with RESTRICTED profile
apiVersion: v1
kind: Namespace
metadata:
  name: production
  labels:
    # Enforce restricted profile - highest security
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest

    # Warn and audit on the same level
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: latest
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: latest

    # Additional organizational labels
    environment: production
    security-tier: high
---
# Example 2: Development namespace with BASELINE profile
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    # Enforce baseline profile - balance security and flexibility
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: latest

    # Warn on restricted violations (encourage best practices)
```

```yaml
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: latest
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: latest

    environment: development
    security-tier: medium
---
# Example 3: System namespace with PRIVILEGED profile (use sparingly)
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring-system
  labels:
    # Allow privileged workloads (monitoring agents need host access)
    pod-security.kubernetes.io/enforce: privileged
    pod-security.kubernetes.io/enforce-version: latest

    # Still warn and audit on violations
    pod-security.kubernetes.io/warn: baseline
    pod-security.kubernetes.io/warn-version: latest
    pod-security.kubernetes.io/audit: baseline
    pod-security.kubernetes.io/audit-version: latest

    environment: production
    workload-type: infrastructure
---
# ==============================================================================
# EXAMPLE: RESTRICTED POD SPECIFICATION
# ==============================================================================
# This pod meets all Restricted profile requirements

apiVersion: v1
kind: Pod
metadata:
  name: secure-application
  namespace: production
  labels:
    app: secure-app
spec:
  # ============================================================================
  # POD-LEVEL SECURITY CONTEXT
  # ============================================================================
  securityContext:
    # Run as non-root user (required by Restricted)
    runAsNonRoot: true

    # Specify user ID explicitly (best practice)
    runAsUser: 10001
    runAsGroup: 10001
```

```yaml
    # Set filesystem group for volume permissions
    fsGroup: 10001

    # Ensure fsGroup ownership applied efficiently
    fsGroupChangePolicy: OnRootMismatch

    # Apply seccomp profile (required by Restricted)
    seccompProfile:
      type: RuntimeDefault

    # Supplemental groups (if needed for file access)
    supplementalGroups: [10002]

  containers:
  - name: app
    image: myapp:1.0.0

    # ============================================================================
    # CONTAINER-LEVEL SECURITY CONTEXT
    # ============================================================================
    securityContext:
      # Prevent privilege escalation (required by Restricted)
      allowPrivilegeEscalation: false

      # Run as non-root (required by Restricted)
      runAsNonRoot: true
      runAsUser: 10001
      runAsGroup: 10001

      # Drop all capabilities (required by Restricted)
      capabilities:
        drop:
          - ALL

      # Read-only root filesystem (required by Restricted)
      readOnlyRootFilesystem: true

      # Explicitly deny privileged mode (required by Restricted)
      privileged: false

    # Resource limits (recommended for Restricted)
    resources:
      requests:
        memory: "128Mi"
        cpu: "100m"
      limits:
        memory: "256Mi"
        cpu: "200m"

    # Volume mounts for writable directories
    volumeMounts:
```

```yaml
    - name: tmp
      mountPath: /tmp
    - name: cache
      mountPath: /app/cache

    # Health checks
    livenessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10

    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5

  # ============================================================================
  # VOLUMES (using emptyDir for writable directories)
  # ============================================================================
  volumes:
  - name: tmp
    emptyDir:
      medium: Memory
      sizeLimit: 100Mi
  - name: cache
    emptyDir:
      sizeLimit: 500Mi

  # ============================================================================
  # ADDITIONAL SECURITY SETTINGS
  # ============================================================================

  # Use specific service account (not default)
  serviceAccountName: app-service-account

  # Disable service account token auto-mount if not needed
  automountServiceAccountToken: false

  # Node affinity for security zones (optional)
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: security-zone
            operator: In
            values:
```

```yaml
            - restricted

  # Toleration for dedicated security nodes
  tolerations:
  - key: security
    operator: Equal
    value: restricted
    effect: NoSchedule
---
# ============================================================================
# BASELINE POD EXAMPLE (less restrictive but still secure)
# ============================================================================

apiVersion: v1
kind: Pod
metadata:
  name: baseline-app
  namespace: development
spec:
  securityContext:
    # Can run as any non-root user
    runAsNonRoot: true
    # Seccomp not strictly required but recommended
    seccompProfile:
      type: RuntimeDefault

  containers:
  - name: app
    image: myapp:dev

    securityContext:
      # Must prevent privilege escalation
      allowPrivilegeEscalation: false

      # Can have some capabilities (not ALL)
      capabilities:
        drop:
          - ALL
        add:
          - NET_BIND_SERVICE  # Allow binding to ports < 1024

      # Read-only filesystem not required
      readOnlyRootFilesystem: false

      # Must not be privileged
      privileged: false

    # Can write to any path (no read-only root filesystem)
    # But still follow security best practices
---
# ============================================================================
```

```yaml
# SERVICE ACCOUNT WITH RESTRICTED PERMISSIONS
# ================================================================================

apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: production
automountServiceAccountToken: false  # Only mount when explicitly needed
---
# RBAC role for service account (least privilege)
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: production
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get", "list"]
  # Only specific ConfigMaps, not all
  resourceNames: ["app-config"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-role-binding
  namespace: production
subjects:
- kind: ServiceAccount
  name: app-service-account
  namespace: production
roleRef:
  kind: Role
  name: app-role
  apiGroup: rbac.authorization.k8s.io
```

Detailed Analysis of Pod Security Standard Implementation:

Multi-Level Enforcement Strategy: The configuration demonstrates a sophisticated approach to PSS enforcement using multiple enforcement modes. The `enforce` mode strictly blocks pods that violate policies, while `warn` mode allows deployment but generates warnings, and `audit` mode silently logs violations. This multi-level approach enables gradual security tightening—organizations can start with baseline enforcement while warning on restricted violations, giving teams time to remediate applications before enforcing the stricter restricted profile.

Namespace-Based Policy Application: Applying security policies at the namespace level provides granular control appropriate for different workload types and environments. Production namespaces enforce restricted profiles for maximum security, development namespaces use baseline profiles to balance security with development velocity, and specialized infrastructure namespaces might use privileged profiles for system components that genuinely require host access. This namespace-level granularity enables defense-in-depth while accommodating legitimate operational requirements.

Restricted Profile Deep Dive: The restricted pod example demonstrates all requirements for the strictest security profile. Running as non-root (`runAsNonRoot: true` with explicit UIDs) prevents numerous privilege escalation attacks. Dropping all Linux capabilities (`capabilities.drop: [ALL]`) removes dangerous privileges even from non-root processes. Using read-only root filesystems (`readOnlyRootFilesystem: true`) prevents malware installation and persistence mechanisms. Preventing privilege escalation (`allowPrivilegeEscalation: false`) blocks exploitation of setuid binaries. Together, these controls dramatically reduce the impact of application compromises.

Read-Only Root Filesystem Pattern: The read-only root filesystem requirement of the restricted profile prevents writing to the container filesystem, forcing explicit declaration of writable paths through volume mounts. This pattern significantly improves security by preventing attackers from modifying system binaries, installing backdoors, or creating persistent malware. The example uses emptyDir volumes for `/tmp` and application cache directories, providing writable storage where needed while maintaining immutability elsewhere.

Service Account Security: The example demonstrates service account security best practices. Creating dedicated service accounts for each application (rather than using the default service account) enables fine-grained RBAC policies. Setting `automountServiceAccountToken: false` prevents automatic mounting of service account credentials into pods that don't need Kubernetes API access, reducing the impact of container compromises. The associated RBAC role grants only the minimum necessary permissions—reading specific ConfigMaps rather than all resources.

Resource Limits in Context: While not strictly required by PSS, resource limits are critical for defense in depth. They prevent resource exhaustion attacks where compromised containers attempt to consume all cluster resources. The restricted profile example includes both requests (guaranteed resources) and limits (maximum allowed), ensuring predictable behavior and preventing noisy neighbor problems.

Progressive Security Migration: Organizations with existing workloads can use the baseline profile as a stepping stone toward restricted. Start by enforcing baseline cluster-wide, which blocks the most dangerous configurations with minimal application changes required. Configure warning and audit for restricted violations to identify applications needing remediation. Work through applications systematically, updating them to meet restricted requirements. Once all applications comply, switch enforcement to restricted. This gradual approach prevents security improvements from disrupting operations.

This comprehensive approach to Pod Security Standards provides strong baseline security while accommodating the operational realities of running diverse workloads in Kubernetes clusters. The key is balancing security with usability, using technology controls (PSS enforcement) combined with process controls (security reviews, remediation plans) to progressively improve security posture.

Chapter 4: Kubernetes Network Policies - Microsegmentation and Defense in Depth

Network security in Kubernetes environments presents unique challenges due to the dynamic nature of pod scheduling, the flat network model where all pods can communicate by default, and the east-west traffic patterns of microservices architectures. Traditional network security controls designed for north-south perimeter defense are insufficient for securing container workloads. Kubernetes Network Policies provide declarative microsegmentation capabilities, enabling security teams to implement zero-trust networking principles where every connection must be explicitly authorized.

4.1 Network Policy Fundamentals and Default-Deny Strategy

Kubernetes Network Policies define rules controlling network traffic to and from pods. Without Network Policies, all pods in a cluster can communicate with all other pods a flat network topology that provides no isolation between different applications or tenants. This default-allow posture creates significant security risks where compromised pods can lateral movement to attack other workloads.

Understanding Network Policy Architecture:

Network Policies are implemented by CNI (Container Network Interface) plugins, not by Kubernetes itself. This means your cluster must use a CNI plugin that supports Network Policies—such as Calico, Cilium, Weave Net, or others.

Popular CNI plugins like Flannel do not support Network Policies in their default configuration. Attempting to apply Network Policies in clusters without a supporting CNI plugin has no effect, creating a dangerous false sense of security.

Network Policies are namespace-scoped resources that select pods using label selectors, similar to how Services select pods. Selected pods have their network traffic restricted according to the policy rules. Policies can control both ingress (incoming) and egress (outgoing) traffic, with separate rule sets for each direction. Multiple policies can apply to the same pod, with the combination of all matching policies determining the effective rules (union of all allow rules).

The Default-Deny Pattern:

The most important network security pattern for Kubernetes is implementing default-deny Network Policies that block all traffic except explicitly allowed connections. This zero-trust approach ensures that only documented, intended communication paths are permitted, dramatically reducing the blast radius of security incidents.

Here's a comprehensive example demonstrating default-deny network policies and progressive microsegmentation:

```
# ============================================================================
# DEFAULT-DENY NETWORK POLICIES
# ============================================================================
# These policies should be applied first, establishing a secure baseline

---
# Policy 1: Default deny all ingress traffic to all pods
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: production
spec:
  # Select ALL pods in namespace (empty selector matches everything)
  podSelector: {}

  # Define policy types this policy applies to
  policyTypes:
  - Ingress

  # Empty ingress rules = deny all ingress
  # (presence of Ingress in policyTypes with no rules = deny)

---
# Policy 2: Default deny all egress traffic from all pods
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
  namespace: production
spec:
  podSelector: {}
  policyTypes:
  - Egress

  # Empty egress rules = deny all egress

# ============================================================================
# With default-deny in place, pods cannot communicate at all.
# Now we add specific allow rules for required communication paths.
# ============================================================================

---
# Policy 3: Allow DNS queries (required for service discovery)
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
  namespace: production
spec:
  # Apply to ALL pods (they all need DNS)
  podSelector: {}

  policyTypes:
  - Egress

  egress:
  # Allow DNS queries to kube-dns/CoreDNS
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53

---
# ============================================================================
# APPLICATION-SPECIFIC NETWORK POLICIES
# ============================================================================
# Define explicit allow rules for each microservice communication path

---
# Policy 4: Frontend service ingress rules
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-ingress
  namespace: production
spec:
  # Select frontend pods
  podSelector:
    matchLabels:
      app: frontend
      tier: web

  policyTypes:
  - Ingress

  ingress:
  # Rule 1: Accept HTTPS from ingress controller
  - from:
    - namespaceSelector:
        matchLabels:
          name: ingress-nginx
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: ingress-nginx
    ports:
    - protocol: TCP
      port: 8080
      # Optional: Specify endpoint port name instead of number
      # port: http

  # Rule 2: Accept traffic from monitoring system (Prometheus)
  - from:
    - namespaceSelector:
        matchLabels:
          name: monitoring
    - podSelector:
        matchLabels:
          app: prometheus
    ports:
    - protocol: TCP
      port: 9090  # Metrics endpoint

---
# Policy 5: Frontend service egress rules
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-egress
  namespace: production
spec:
  podSelector:
```

```yaml
    matchLabels:
      app: frontend
      tier: web

  policyTypes:
  - Egress

  egress:
  # Rule 1: Allow connections to backend API service
  - to:
    - podSelector:
        matchLabels:
          app: api
          tier: backend
    ports:
    - protocol: TCP
      port: 8080

  # Rule 2: Allow connections to Redis cache
  - to:
    - podSelector:
        matchLabels:
          app: redis
          tier: cache
    ports:
    - protocol: TCP
      port: 6379

  # Rule 3: Allow DNS (covered by allow-dns-access policy)
  # Rule 4: Allow external API calls (specific external services)
  - to:
    - namespaceSelector: {}  # Any namespace
      podSelector: {}
    # Use CIDR to allow external IPs
    ports:
    - protocol: TCP
      port: 443  # HTTPS only
    # Limit to specific external IPs
  - to:
    - ipBlock:
        cidr: 203.0.113.0/24  # External API service IP range
        except:
        - 203.0.113.1/32  # Exclude specific problematic IP

---
# Policy 6: Backend API service policies
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-network-policy
  namespace: production
```

```yaml
spec:
  podSelector:
    matchLabels:
      app: api
      tier: backend

  policyTypes:
  - Ingress
  - Egress

  ingress:
  # Only accept from frontend service
  - from:
    - podSelector:
        matchLabels:
          app: frontend
          tier: web
    ports:
    - protocol: TCP
      port: 8080

  egress:
  # Allow connections to database
  - to:
    - podSelector:
        matchLabels:
          app: postgresql
          tier: database
    ports:
    - protocol: TCP
      port: 5432

  # Allow connections to Redis
  - to:
    - podSelector:
        matchLabels:
          app: redis
          tier: cache
    ports:
    - protocol: TCP
      port: 6379

  # Allow DNS
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
    - protocol: UDP
      port: 53
```

```yaml
---
# Policy 7: Database ingress (most restrictive)
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-network-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: postgresql
      tier: database

  policyTypes:
  - Ingress
  - Egress

  ingress:
  # ONLY allow connections from API backend
  # No other service should directly access the database
  - from:
    - podSelector:
        matchLabels:
          app: api
          tier: backend
    ports:
    - protocol: TCP
      port: 5432

  # Also allow from backup service
  - from:
    - podSelector:
        matchLabels:
          app: backup
          tier: ops
    ports:
    - protocol: TCP
      port: 5432

  egress:
  # Database needs minimal egress
  # Allow DNS for potential external connections (like S3 for backups)
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
    ports:
    - protocol: UDP
      port: 53

  # Allow S3 access for backups (if using AWS S3)
```

```yaml
  - to:
    - ipBlock:
        cidr: 52.216.0.0/15  # S3 IP range (example)
    ports:
    - protocol: TCP
      port: 443

---
# ============================================================================
# CROSS-NAMESPACE COMMUNICATION POLICY
# ============================================================================
# Controlling traffic between namespaces (multi-tenancy)

---
# Policy 8: Allow specific cross-namespace communication
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-staging
  namespace: production
spec:
  # Select specific pods in production namespace
  podSelector:
    matchLabels:
      app: shared-service
      access-level: shared

  policyTypes:
  - Ingress

  ingress:
  # Allow access from staging namespace
  - from:
    - namespaceSelector:
        matchLabels:
          name: staging
      podSelector:
        matchLabels:
          allowed: "true"
    ports:
    - protocol: TCP
      port: 8080

---
# ============================================================================
# ADVANCED: NAMED PORT USAGE
# ============================================================================
# Using named ports for better maintainability

---
apiVersion: v1
```

```yaml
kind: Service
metadata:
  name: frontend
  namespace: production
spec:
  selector:
    app: frontend
  ports:
  - name: http
    port: 80
    targetPort: http  # References named port in pod
  - name: metrics
    port: 9090
    targetPort: metrics
---
# Network policy using named ports
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-named-ports
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: frontend

  policyTypes:
  - Ingress

  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: ingress-nginx
    ports:
    - protocol: TCP
      port: http  # References named port (more maintainable)

  - from:
    - namespaceSelector:
        matchLabels:
          name: monitoring
    ports:
    - protocol: TCP
      port: metrics

---
# ================================================================================
# TROUBLESHOOTING HELPER: TEMPORARY ALLOW-ALL POLICY
# ================================================================================
# Use this temporarily to test if Network Policies are blocking traffic
```

```yaml
# NEVER use this in production except for debugging!

---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: temporary-allow-all-debug
  namespace: production
  annotations:
    # Document why this exists and when it should be removed
    purpose: "Temporary debugging - DELETE BY 2024-12-31"
    ticket: "JIRA-12345"
spec:
  podSelector:
    matchLabels:
      debug: "true"  # Only pods with this label

  policyTypes:
  - Ingress
  - Egress

  ingress:
  - {}  # Allow all ingress

  egress:
  - {}  # Allow all egress
```

Comprehensive Analysis of Network Policy Patterns:

Default-Deny Foundation: The first two policies (`default-deny-ingress` and `default-deny-egress`) establish a zero-trust foundation by blocking all traffic. The empty `podSelector: {}` means these policies apply to all pods in the namespace. The presence of `Ingress` or `Egress` in `policyTypes` with no matching rules creates an implicit deny-all effect. This is counterintuitive—you might expect that no rules means no restrictions, but in Network Policy semantics, selecting a policy type with no rules means "deny all traffic of this type." This default-deny approach is critical for security because it inverts the security model from "block known-bad" to "allow known-good."

DNS Allowance - Essential for Function: After implementing default-deny, the first exception needed is DNS access. Without DNS, pods cannot resolve service names or external hostnames, breaking normal application functionality. The `allow-dns-access` policy permits DNS queries to kube-dns/CoreDNS pods in the kube-system namespace. This policy uses both `namespaceSelector` (selecting the kube-system namespace) and opens ports 53 for both UDP and TCP. While DNS primarily uses UDP, some queries fall back to TCP, so both protocols should be allowed.

Layered Security Through Multiple Policies: The frontend service has both ingress and egress policies defined in separate Network Policy resources. This separation improves maintainability and follows the principle of separation of concerns. The ingress policy controls what can connect to the frontend (ingress controller and monitoring), while the egress policy controls what the frontend can connect to (backend API, Redis, external services). Multiple policies affecting the same pods create additive allow rules—the effective policy is the union of all matching policies.

Microsegmentation Pattern: The policies implement strict microsegmentation where each service can only communicate with its direct dependencies. The frontend can connect to the backend API and Redis, but not directly to the database. The backend API can connect to the database and Redis. The database accepts connections only from the backend API and backup service. This layered approach limits the blast radius of compromises—an attacker who compromises the frontend cannot directly access the database; they must first compromise the backend API, significantly increasing attack complexity.

External Traffic Control: The frontend egress policy demonstrates controlling external API access using `ipBlock` with CIDR notation. This allows specifying external IP ranges that pods can connect to, preventing arbitrary internet access. The `except` clause within `ipBlock` enables excluding specific IPs from otherwise-allowed ranges, useful for blacklisting known-malicious IPs within generally-trusted IP blocks. For production environments, restricting external access to specific endpoints reduces the risk of data exfiltration and command-and-control communication.

Cross-Namespace Communication: The `allow-from-staging` policy shows how to control cross-namespace traffic in multi-tenant clusters. It uses both `namespaceSelector` (selecting the staging namespace) and `podSelector` (selecting pods with `allowed: "true"` label) in combination. Both selectors must match for traffic to be allowed, implementing both namespace-level and pod-level access control. This pattern is essential for multi-tenant Kubernetes clusters where different teams or applications share the cluster but should have limited interaction.

Named Ports for Maintainability: The `frontend-named-ports` policy demonstrates using named ports instead of port numbers. Named ports reference port names defined in pod specifications, making policies more maintainable when port numbers change. Instead of updating all Network Policies when a port number changes, you only update the pod specification. This is particularly valuable in large environments with many Network Policies.

Database Security - Defense in Depth: The database Network Policy is the most restrictive, reflecting the critical nature of data stores. It allows ingress only

from the backend API and backup service—no other services should directly access the database. The egress policy is similarly restrictive, allowing only DNS and S3 access (for backups). This tight restriction implements the principle of least privilege at the network level, ensuring databases are protected even if application-level access controls fail.

Troubleshooting Considerations: Network Policies can be challenging to debug when things don't work as expected. The temporary allow-all policy demonstrates a debugging technique—apply a permissive policy to specific pods (selected by the `debug: "true"` label) to determine if Network Policies are causing connectivity issues. This policy should never exist in production except during active debugging and should be immediately removed after troubleshooting. The annotations document why it exists and when it should be removed, preventing temporary debugging policies from becoming permanent.

Operational Best Practices:

1. Apply Default-Deny First: Always start by applying default-deny policies to establish a secure baseline, then add specific allow rules. Attempting to add deny rules to an allow-by-default model inevitably leaves gaps.
2. Document Communication Flows: Maintain architecture diagrams showing all service-to-service communication paths. Network Policies should directly reflect documented architecture, making it easy to verify that policies match intentions.
3. Test in Non-Production First: Network Policy misconfigurations can break applications in hard-to-debug ways. Always test policies in development/staging environments before applying to production.
4. Use Namespace Labels Consistently: Establish naming conventions for namespace labels used in Network Policies. Inconsistent labeling makes policies fragile and hard to maintain.
5. Monitor Policy Violations: Use tools like Cilium Hubble or Calico Enterprise to monitor denied connections. These telemetry tools show which connections are being blocked, helping identify both legitimate traffic that needs policy updates and malicious connection attempts that policies are correctly blocking.

This comprehensive approach to Network Policies creates defense-in-depth network security suitable for zero-trust architectures, dramatically reducing attack surface and limiting the impact of security incidents.

Frequently Asked Questions

What's the difference between Docker security and Kubernetes security?

Docker security focuses on securing individual containers and the Docker daemon, including image security, container runtime configuration, and host-level protections. Kubernetes security operates at a higher abstraction level, addressing orchestration concerns like pod security, service-to-service authentication, network microsegmentation, and secrets management across distributed workloads. While Docker provides the foundation, Kubernetes adds layers of security controls specific to managing containerized applications at scale. Both are essential—poor Docker security undermines Kubernetes security controls, while strong Docker security without Kubernetes-level controls leaves gaps in distributed systems security.

How do I migrate from PodSecurityPolicy to Pod Security Standards?

PodSecurityPolicy (PSP) was deprecated in Kubernetes 1.21 and removed in 1.25, replaced by Pod Security Standards (PSS). Migration involves: (1) Analyzing existing PSP usage to understand current security posture, (2) Mapping PSP configurations to equivalent PSS profiles (Privileged, Baseline, or Restricted), (3) Applying PSS labels to namespaces starting with permissive profiles and warning/audit modes, (4) Gradually tightening profiles as applications are updated to meet requirements, (5) Removing PSPs once PSS enforcement is in place. The key difference is that PSS uses predefined profiles rather than custom policies, simplifying security but requiring application modifications to meet stricter profiles. Organizations should use admission controllers like OPA Gatekeeper or Kyverno for custom policies that PSS doesn't address.

What's the best tool for container vulnerability scanning?

The "best" scanner depends on specific requirements, but popular choices include:

- Trivy: Open-source, comprehensive coverage of OS packages and application dependencies, fast scanning, easy CI/CD integration. Excellent default choice for most organizations.
- Grype: Open-source from Anchore, good accuracy, integrates with Syft for SBOM generation. Strong choice for supply chain security focused organizations.

- Clair: Open-source from Red Hat, focuses on OS vulnerabilities, works well with Quay registry. Good for organizations heavily invested in Red Hat ecosystem.
- Snyk: Commercial tool with excellent developer experience, IDE integration, and automated fix suggestions. Best for organizations prioritizing developer productivity.
- Aqua/Prisma/Sysdig: Enterprise commercial platforms with vulnerability scanning plus runtime protection, compliance, and other security features. Best for enterprises needing comprehensive security platforms.

Most organizations benefit from using multiple scanners to catch vulnerabilities that individual tools might miss, aggregating results in a central vulnerability management platform.

How do I secure secrets in Kubernetes without using external tools like Vault?

While external secret managers like HashiCorp Vault provide the best security, you can improve native Kubernetes Secrets security through:

1. Enable Encryption at Rest: Configure the kube-apiserver with `--encryption-provider-config` to encrypt secrets in etcd using aescbc, aesgcm, or KMS providers.
2. RBAC Restrictions: Use strict RBAC to limit who can read secrets. Create roles that grant access only to specific secrets needed, not all secrets in a namespace.
3. Sealed Secrets: Use Bitnami Sealed Secrets to encrypt secrets before committing to Git, enabling GitOps workflows while protecting sensitive data.
4. External Secrets Operator: Use External Secrets Operator as a middle ground—it pulls secrets from various sources (AWS Secrets Manager, Azure Key Vault, GCP Secret Manager) without running a separate Vault cluster.
5. Avoid Environment Variables: Mount secrets as volumes rather than environment variables, as environment variables are easier to expose through process listings and error messages.

However, these approaches still have limitations compared to dedicated secret managers, particularly around dynamic secret generation, automated rotation, and comprehensive audit logging.

What's the difference between Network Policies and Service Mesh for security?

Network Policies and Service Meshes provide complementary network security capabilities:

Network Policies:

- Layer 3/4 network-level traffic control (IP addresses and ports)
- Implemented by CNI plugins at the network layer
- Simpler to understand and implement
- Lower overhead and better performance
- Limited to allow/deny decisions based on labels and namespaces
- No encryption, authentication, or Layer 7 controls

Service Mesh (Istio, Linkerd):

- Layer 7 application-level traffic control (HTTP paths, methods, headers)
- Implemented by sidecar proxies in each pod
- More complex setup and operation
- Higher resource overhead (sidecar containers)
- Provides mutual TLS encryption between services automatically
- Offers rich traffic management (retries, timeouts, circuit breaking)
- Enables advanced security policies (JWT validation, authorization policies)
- Provides detailed observability (distributed tracing, metrics)

Organizations typically use both: Network Policies for baseline network segmentation and Service Mesh for application-level security and traffic management. Start with Network Policies for their simplicity, add Service Mesh when you need its advanced features.

How often should I update container base images?

Update frequency depends on balancing security with operational stability:

Critical Vulnerabilities: Update immediately when critical CVEs affecting your images are disclosed, especially if they're remotely exploitable. Don't wait for scheduled maintenance windows.

Regular Updates: Establish a regular update cadence:

- Production: Monthly updates work for most organizations, providing frequent security patches while maintaining stability
- Development/Staging: Weekly or bi-weekly updates catch issues earlier
- Infrastructure Images: Update when base image releases are published (often monthly)

Automated Scanning: Run vulnerability scans daily or on every build, even if not updating immediately. This provides visibility into your security posture and enables quick response when critical vulnerabilities emerge.

Update Process: Successful update programs include:

1. Automated scanning detecting base image updates
2. Automated PR creation rebuilding images with new base images
3. Automated testing verifying functionality
4. Manual review and approval for production deployments
5. Phased rollout limiting blast radius

Organizations with mature DevOps practices can update more frequently (weekly or even daily), while those with limited automation should focus on monthly updates with emergency processes for critical vulnerabilities.

What runtime security monitoring tools should I use?

Runtime security monitoring tools detect suspicious behavior after containers are deployed:

Falco: Open-source CNCF project, uses eBPF or kernel module to monitor system calls, highly customizable rules, excellent community support. Best for organizations wanting flexibility and community-driven development.

Tetragon: eBPF-based security observability from Isovalent (Cilium creators), deep kernel visibility, strong Kubernetes integration. Excellent for organizations already using Cilium networking.

Sysdig Secure: Commercial platform built on Falco, adds enterprise features like incident response workflows, compliance reporting, and commercial support. Good for enterprises wanting turnkey solutions.

Aqua Security: Comprehensive commercial platform including runtime protection, vulnerability management, and compliance. Best for enterprises needing all-in-one security platforms.

Prisma Cloud (Palo Alto): Enterprise security platform with runtime defense, covers multiple cloud providers and environments. Strong choice for organizations standardized on Palo Alto security tools.

Open-Source Combination: Many organizations use Falco for runtime monitoring, combined with Prometheus/Grafana for metrics and alerting, providing powerful capabilities without licensing costs. This requires more operational expertise but offers maximum flexibility.

The choice depends on budget, existing tool investments, required features, and operational capabilities. Start with open-source tools like Falco to understand requirements, then evaluate commercial tools if you need enterprise features, support, or integrated platforms.

Related Articles

- CIS Kubernetes Benchmark - Industry-standard security configuration guidelines
- NSA Kubernetes Hardening Guide - Government security guidance for Kubernetes
- NIST Application Container Security Guide - Comprehensive container security framework
- Kubernetes Security Documentation - Official Kubernetes security documentation
- Pod Security Standards - Official PSS documentation
- Docker Security Best Practices - Official Docker security documentation
- CNCF Security TAG - Cloud Native Computing Foundation security resources
- Falco Rules Repository - Community-maintained runtime security rules
- Kubernetes CVE List - Official Kubernetes vulnerability tracking
- OPA Gatekeeper Policy Library - Pre-built admission control policies

---

Conclusion and Security Maturity Assessment

Container security represents a continuous journey rather than a destination. Organizations must assess their current security posture, identify gaps, prioritize improvements, and systematically implement controls across the security stack. This comprehensive guide has covered the essential technical domains for securing containerized workloads—from Docker image hardening and vulnerability scanning to Kubernetes network policies and Pod Security Standards.

Container Security Maturity Model

Organizations typically progress through several maturity stages in their container security journey:

Level 1 - Basic (Ad-Hoc):

- No systematic vulnerability scanning
- Running containers as root
- No network policies or segmentation
- Secrets in environment variables or ConfigMaps
- Manual security reviews (if any)

Level 2 - Developing (Reactive):

- Basic vulnerability scanning in CI/CD
- Some containers run as non-root
- Network policies in selected namespaces
- Kubernetes secrets used (but not encrypted)
- Security reviews for critical applications

Level 3 - Established (Proactive):

- Comprehensive vulnerability scanning with blocking thresholds
- Baseline Pod Security Standards enforced
- Default-deny network policies cluster-wide
- External secret management (Vault/cloud providers)
- Regular security audits and compliance checking
- Runtime security monitoring deployed

Level 4 - Advanced (Optimized):

- Automated security testing in all pipelines
- Restricted Pod Security Standards enforced
- Network segmentation with service mesh
- Dynamic secret generation and rotation
- Advanced runtime threat detection with automated response
- Supply chain security with SBOMs and image signing
- Regular penetration testing and red team exercises

Level 5 - Leading (Innovative):

- Zero-trust architecture implementation
- Policy-as-code for all security controls
- AI/ML-powered threat detection
- Automated remediation workflows
- Security chaos engineering
- Contribution to open-source security projects

Most organizations operate between levels 2 and 3, with significant variation between teams and applications. Moving up the maturity ladder requires investment in tools, training, and process improvement, but delivers substantial risk reduction and improved security posture.

Implementation Roadmap

For organizations beginning their container security journey, we recommend this implementation roadmap:

Phase 1 (Months 1-3): Foundation

1. Implement basic vulnerability scanning in CI/CD pipelines
2. Start building non-root container images
3. Enable audit logging on API server
4. Document current architecture and communication flows
5. Train development teams on container security basics

Phase 2 (Months 4-6): Core Controls

1. Deploy Baseline Pod Security Standards with warnings
2. Implement default-deny Network Policies in non-production namespaces
3. Move to external secret management for new applications
4. Establish vulnerability remediation SLAs
5. Implement runtime security monitoring (Falco) in monitoring mode

Phase 3 (Months 7-9): Enhancement

1. Enforce Baseline Pod Security Standards cluster-wide
2. Expand Network Policies to production namespaces
3. Implement SBOM generation for all images
4. Configure runtime security alerts and response procedures
5. Conduct first security assessment or penetration test

Phase 4 (Months 10-12): Optimization

1. Work toward Restricted Pod Security Standards
2. Implement service mesh for advanced security
3. Deploy image signing and admission control
4. Establish continuous compliance monitoring
5. Document security architecture and runbooks

This roadmap provides a structured approach while remaining flexible enough to adapt to organizational priorities and constraints. The key is maintaining momentum while avoiding security improvements that disrupt operations.

Final Recommendations

Container security requires commitment from multiple stakeholders—security teams, platform engineers, developers, and leadership. Success demands:

Technical Excellence: Implement comprehensive security controls across all layers of the stack using the patterns and configurations detailed in this guide.

Process Discipline: Establish clear policies, procedures, and responsibilities for container security. Document security requirements, review processes, and escalation procedures.

Continuous Learning: Container security evolves rapidly. Stay current with new vulnerabilities, attack techniques, and security tools through ongoing training and community participation.

Culture of Security: Build security into development workflows through shift-left practices, security champions programs, and blameless postmortems that treat security incidents as learning opportunities.

Measurement and Improvement: Track security metrics (vulnerabilities by severity, policy violations, mean-time-to-remediate) and use data to drive continuous improvement.

Container security is challenging, but the patterns and practices in this guide provide a solid foundation for building secure, scalable containerized applications. Organizations that systematically implement these controls dramatically reduce their risk while enabling the agility and efficiency that containers promise.

Author: Okan YILDIZ
Email: okan@securedebug.com

Last Updated: December 2025
Version: 1.0
Document Classification: Technical Guide - Public