Practical DevSecOps

Container Security 101
Secure Every Layer

Secure Your Containers and Safeguard
Your Code with Confidence!

www.practical-devsecops.com

Contents

Introduction to Container Security

Overview of Container Security

Today's computer programs often run in special packages called containers. Think of containers like sealed boxes that hold everything a program needs to run. This new way of running programs is great, but it needs special care to keep it safe.

Here's why container security matters: One program might use many containers, each with its parts from different places. Each piece needs to be checked and protected to keep the whole program safe.

This guide will help you:

- Understand how to keep containers safe
- Learn what problems to watch for
- Find out how to fix security issues quickly
- Keep your programs protected from bad guys

Whether you work with containers every day or are just learning about them, this guide will show you how to protect your programs better. You'll learn everything from basic safety steps to advanced ways to catch problems early.

The main goal is simple: help you build and run safer container-based programs that can stand up to today's security threats.

CHAPTER 1

Understanding Container Fundamentals and Security Implications

Computer containers are changing how we run programs. While they make things easier, we need new ways to keep them safe. The old ways of protecting computer systems don't work as well with containers.

Think of containers as sealed packages that hold programs. Each container has its walls to keep it separate from other containers. But these walls need special security rules to work right.

To keep containers safe, you need to:

- Check each piece that goes into the container
- Make sure containers can't interfere with each other
- Watch how containers behave when running
- Catch problems before they cause trouble

This guide will teach you these new security rules. You'll learn how to protect your container-based programs from the ground up. Whether you're new to containers or use them every day, you'll discover better ways to keep your programs safe.

The key is understanding that container security is different from old computer security. Once you know these differences, you can build stronger protection for your programs.

Container Architecture: The Building Blocks

Containers use special safety features built into Linux to keep programs separate and controlled. There are two main parts that work together to protect containers:

First are namespaces, which create private spaces for each container. These spaces control different things:

- Each container gets its own list of running programs
- Each has its own private network
- Each sees only its own files
- Each has its own users and permissions
- Each controls how its programs talk to each other
- Each can have its own computer name

Second are control groups (cgroups), which manage how much of the computer's power each container can use:

- They limit how much memory containers can use
- They control how much processing power containers get
- They make sure one container doesn't slow down others
- They watch how much power each container uses

These features work like walls and rules that keep containers separate and make sure they play nice with each other. They help stop containers from causing problems for other containers or the main computer system.

The Container Security Model

Virtual machines and containers protect programs differently. Virtual machines are like having separate computers inside your computer - each with its own complete system. This keeps programs very safe but uses lots of power.

Containers work more simply by sharing one system, making them faster but needing extra security rules. Since containers share more parts of the computer, we need to watch them carefully and set clear boundaries. Both ways work well - you just need to pick what's best for your needs.

Advantages:

1. Containers start faster because they don't need to boot an entire OS
2. They use fewer resources since they share the kernel
3. You can run more containers than VMs on the same hardware

Security Considerations:

1. Kernel vulnerabilities can affect all containers on the host
2. Resource isolation requires careful configuration
3. Container breakouts have a more direct path to the host

Security Boundaries and Their Implementation

Linux kernel security features create the boundaries between containers and the host system. These features include:

Capabilities

Instead of running containers as root, you should drop unnecessary capabilities and retain only those application needs. For example:

```
FROM ubuntu:20.04

# Create a non-root user
RUN useradd -ms /bin/bash appuser

# Switch to the non-root user
USER appuser

# ... rest of your Dockerfile instructions ...
```

Seccomp Profiles

Seccomp filters restrict which system calls your container can make. You should:

1. Create custom seccomp profiles for your applications
2. Block unnecessary system calls
3. Monitor system call patterns to detect anomalies

AppArmor and SELinux

These Mandatory Access Control systems provide additional security layers:

- AppArmor profiles define file access permissions
- SELinux policies control process interactions
- Both help prevent container breakouts

Resource Constraints and Security

Resource limitations serve both operational and security purposes. You must:

1. Set appropriate memory limits to prevent DoS attacks
2. Configure CPU quotas to ensure fair resource sharing
3. Implement disk I/O controls to maintain system stability
4. Monitor resource usage for potential security incidents

Consider this example of secure resource configuration:

```
# Kubernetes pod specification with security
constraints
apiVersion: v1
kind: Pod
metadata:
 name: secure-pod
spec:
 securityContext:
   runAsUser: 1000  # Run as non-root user
   runAsGroup: 3000 # Run as non-root group
   fsGroup: 2000    # Set file system group
 containers:
 - name: app
   image: nginx
   resources:
     limits:
       memory: "256Mi"  # Limit memory usage to 256
MiB
       cpu: "500m"      # Limit CPU usage to 500
milliCPU
     requests:
       memory: "128Mi"  # Request 128 MiB of memory
       cpu: "250m"      # Request 250 milliCPU
   securityContext:
     readOnlyRootFilesystem: true  # Ensure the root
filesystem is read-only
     allowPrivilegeEscalation: false  # Prevent
privilege escalation
     capabilities:
       drop:
       - ALL  # Drop all Linux capabilities
```

Practical Implementation Steps

To implement these security concepts effectively:

Start with the principle of least privilege:

- Run containers as non-root users
- Remove unnecessary capabilities
- Implement read-only root filesystems

Configure resource limits:

- Set appropriate memory constraints
- Define CPU quotas
- Implement disk I/O controls

Enable security features:

- Activate SELinux or AppArmor
- Implement seccomp profiles
- Configure network policies

Monitor and audit:

- Track resource usage
- Monitor system calls
- Log security events

Container security works like layers of protection that fit together. You need to get the basics right first, then add more security as you go. Here's what matters most:

Each safety feature we talked about does something important:

- It sets clear boundaries between containers
- It controls what each container can do
- It monitors how containers use resources

When you use all these features together, your containers stay safer. Once you understand these basics, you can learn more advanced ways to protect your containers.

Next, we'll look at how to keep the stuff inside containers safe. You'll learn how to check that everything in your containers is secure before you use them.

Remember: Good container security starts with these basics. Get them right, and you'll build stronger, safer programs.

CHAPTER 2

Container Image Security Best Practices

Container images are like recipes for your programs. If the recipe has a problem, every program made from it will be unsafe. That's why we need to check our container images carefully. We should use trusted sources, check all parts, and remove anything not needed. This keeps all our programs safer from the start.

Selecting and Verifying Base Images

Start your container security journey with the foundation - your base image. Think of base images like the foundation of a house; you need to trust the materials you're building upon.

When selecting base images:

1. Choose official images from trusted repositories. Docker Official Images and distro-maintained images provide a reliable starting point. For example:

```
# Prefer official images over unknown sources
FROM ubuntu:20.04    # Good: Official Ubuntu image
# Instead of: FROM random-user/ubuntu  # Risky:
Unknown source
```

Use specific version tags rather than 'latest' to ensure consistency:

```
# Be specific with versions
FROM node:16.14.2-alpine3.15  # Good: Specific version
# Instead of: FROM node:latest # Risky: Unpredictable
content
```

Implement image verification using content trust:

```
# Enable Docker Content Trust
export DOCKER_CONTENT_TRUST=1
docker pull ubuntu:20.04
```

Building Secure Images

Let's create images that minimize attack surface and maximize security. Here's how:

1. Minimize Image Size

Remove unnecessary components to reduce attack surface:

```
dockerfile
FROM alpine:3.15
RUN apk add --no-cache python3 && \
    rm -rf /var/cache/apk/*
```

2. Implement Multi-stage Builds

Use multi-stage builds to separate build dependencies from runtime requirements:

```
dockerfile
# Build stage
FROM node:16.14.2 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Production stage
FROM node:16.14.2-alpine
COPY --from=builder /app/dist /app
USER node
CMD ["node", "app/server.js"]
```

3. Implement Security Scanning

Integrate vulnerability scanning into your build process. Here's how to use tools like Trivy:

```
bash
# Scan image for vulnerabilities
trivy image your-image:<Version># Fail builds on high
severity findings
trivy image --exit-code 1 --severity HIGH, CRITICAL
your-image:<Version>
```

Managing Image Security

Securing your images doesn't stop at build time. You need ongoing management:

3. Implement Security Scanning

1. Implement Image Signing

Sign your images to verify their authenticity:

```
bash
# Sign an image
docker trust sign myregistry.azurecr.io/myapp:1.0

# Verify signed image
docker trust inspect myregistry.azurecr.io/myapp:1.0
```

2. Set Up Access Controls

Implement strict access controls for your container registry:

```
yaml
# Example Azure Container Registry role assignment
az role assignment create \
    --role AcrPull \
    --assignee user@example.com \
    --scope /subscriptions/{subscription-id}/
resourceGroups/{resource-group}/providers/
Microsoft.ContainerRegistry/registries/{registry-name}
```

3. Automate Security Testing

Integrate security testing into your CI/CD pipeline:

```
yaml
# Example GitHub Actions workflow
name: Container Security
on: [push]
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Build image
        run: docker build -t myapp:${{ github.sha }} .
      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: myapp:${{ github.sha }}
          format: 'table'
          exit-code: '1'
          ignore-unfixed: true
          severity: 'CRITICAL,HIGH'
```

Supply Chain Security

Protect your entire container supply chain:

Implement Software Bill of Materials (SBOM):

```
# Specify a specific version of Alpine
syft alpine:3.15.0 -o json > sbom.json

# Analyze SBOM for vulnerabilities
grype sbom:sbom.json
```

Set up continuous monitoring:

- Monitor base images for new vulnerabilities
- Track dependency updates
- Implement automated rebuilds when security updates are available

Implement Software Bill of Materials (SBOM):

```
aws ecr put-image-scanning-configuration \
    --repository-name myapp \
    --image-scanning-configuration scanOnPush=true
```

Best Practices for Ongoing Security

Here's a simple way to keep container images safe:

- Update base images automatically
- Check for security problems regularly
- Rebuild images when fixes come out

Track Everything

- Write down what's in each image
- Keep notes about security choices
- List known problems and fixes

Be Ready for Problems

- Write down what's in each image
- Keep notes about security choices
- List known problems and fixes

That's all you need to remember to keep your container images safe and up to date.

CHAPTER 3

Runtime Security and Container Orchestration

Protecting running containers is more than just good setup - you need to watch them carefully while they work. Like guarding a busy building, you must watch for unusual behavior, catch problems quickly, and control what each container can do. Good security means paying attention all the time, not just during setup. This helps stop small problems from becoming big ones.

Runtime Protection: The First Line of Defense

Your containers need protection while they run, just like your body needs to stay healthy. Good protection means watching for problems and fixing them fast, but making sure your programs still work well. When you catch problems early, you can fix them before they cause big trouble.

Container Hardening

Start with these essential hardening measures:

Container Hardening

Start with these essential hardening measures:

```
#Dockerfile
FROM alpine:3.15

# Create a non-root user and group
RUN addgroup -S appgroup && adduser -S appuser -G
appgroup

# Set the working directory
WORKDIR /app

# Copy application files and set ownership
COPY --chown=appuser:appgroup ./app .

# Switch to the non-root user
USER appuser

# Set restrictive file permissions
RUN chmod -R 500 /app/*

# Define the entry point
ENTRYPOINT ["./app"]
```

I've implemented several crucial security controls here:

1. Created a non-root user
2. Set appropriate file permissions
3. Restricted the working directory
4. Removed unnecessary privileges

Resource Isolation

Implement strict resource controls to prevent container breakout attempts:

```
apiVersion: v1
kind: Pod
metadata:
 name: secure-app
spec:
 containers:
 - name: app
   image: your-image-repo/your-image-name:your-tag  # Specify your image here
   securityContext:
     runAsNonRoot: true
     allowPrivilegeEscalation: false
     capabilities:
       drop:
         - ALL
     readOnlyRootFilesystem: true  # Optional: Set the filesystem to read-only
   resources:
     limits:
       memory: "256Mi"
       cpu: "500m"
     requests:
       memory: "128Mi"
       cpu: "250m"
```

This configuration:

- Prevents privilege escalation
- Drops unnecessary capabilities
- Sets resource limits
- Enforces non-root execution

Kubernetes Security: Orchestrating Secure Deployments

When managing containers at scale, Kubernetes becomes your command center. Let's secure it properly.

Pod Security Policies

It's important to note that as of Kubernetes 1.21, PodSecurityPolicy is deprecated and will be removed in future releases. It's recommended to transition to using Pod Security Admission or other security mechanisms like Open Policy Agent (OPA) with Gatekeeper for future-proofing your security policies.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false  # Prevents privileged containers
  seLinux:
    rule: RunAsAny  # Allows any SELinux context
  runAsUser:
    rule: MustRunAsNonRoot  # Requires non-root
execution
  fsGroup:
    rule: RunAsAny  # Allows any fsGroup
  runAsGroup:
    rule: MustRunAs  # Optional: Specify a non-root
group
    ranges:
      - min: 1000
        max: 65535
```

```
  supplementalGroups:
      rule: RunAsAny  # Allows any supplemental groups
    readOnlyRootFilesystem: true  # Optional: Enforce
  read-only root filesystem
    volumes:
      - configMap
      - emptyDir
      - projected
      - secret
      - downwardAPI
      - persistentVolumeClaim
    allowPrivilegeEscalation: false  # Prevents
  privilege escalation
    defaultAllowPrivilegeEscalation: false
  # Default setting for privilege escalation.
```

This policy:

- Prevents privileged containers
- Requires non-root execution
- Restricts volume types
- Enforces SELinux controls

RBAC Configuration

Implement Role-Based Access Control to limit permissions:

```
yaml
# Example RBAC configuration
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: production
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: production
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Secrets Management

Protect sensitive information using Kubernetes secrets:

```
yaml
# Create encrypted secrets
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  API_KEY: <base64-encoded-value>
---
# Mount secrets securely
apiVersion: v1
kind: Pod
metadata:
  name: secure-app
spec:
  containers:
  - name: app
    volumeMounts:
    - name: secrets
      mountPath: /etc/secrets
      readOnly: true
  volumes:
  - name: secrets
    secret:
      secretName: app-secrets
```

Runtime Monitoring and Detection

Implement comprehensive monitoring to detect security events:

```
yaml
# Example Falco rule for runtime security
- rule: Terminal Shell in Container
  desc: A shell was spawned by a container with an
attached terminal
  condition: >
    container.id != host and
    proc.name = bash and
    evt.type = execve and
    evt.dir = < and
    container.tty = true
  output: >
    Shell spawned in a container with terminal
(user=%user.name
    container_id=%container.id
container_name=%container.name)
  priority: WARNING
```

Container Lifecycle Management

Implement secure container lifecycle policies:

Container Lifecycle Management

Implement secure container lifecycle policies:

1. **Automatic container updates:**

```
#apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-app
spec:
  replicas: 3  # Specify the number of replicas
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%  # Up to 25% of the pods can
be unavailable during the update
      maxSurge: 25%        # Up to 25% more pods than
the desired number can be created during the update
  template:
    metadata:
      labels:
        app: secure-app
    spec:
      containers:
      - name: app
        image: your-image-repo/your-image-name:your-
tag  # Specify your image here
        ports:
        - containerPort: 80
```

Health monitoring:

```
yaml
spec:
  containers:
  - name: app
    image: your-image-repo/image:your-tag  # Specify
your image here
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Best Practices for Runtime Security

Implement secure container lifecycle policies:

```
yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: your-namespace  # Specify the namespace
spec:
  podSelector: {}  # Selects all pods in the namespace
  policyTypes:
  - Ingress
  - Egress
```

Enable Audit Logging:

```
yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: RequestResponse
  resources:
  - group: ""
    resources: ["pods"]
```

Remember: Runtime security requires constant vigilance. Monitor your containers continuously, respond to security events promptly, and regularly update your security policies.

CHAPTER 4

Network Security in Containerized Environments

Container networks need special protection because programs are split into pieces that talk to each other. We must make sure containers only connect with the right partners and keep their messages safe. This helps stop problems before they start.

Understanding Container Network Architecture

Container networks connect program parts that need to work together. Like controlling traffic on roads, we need rules about which containers can connect and how they share information. This keeps programs safe while letting them work together.

Implementing Network Isolation

Let's start with a practical example of network isolation in Kubernetes:

```
yaml
# Network policy to restrict pod communication
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-access
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: web-app
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          environment: production
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: database
    ports:
    - protocol: TCP
      port: 5432
```

This policy creates a secure communication channel by:

- Allowing incoming traffic only from production namespace
- Restricting outbound traffic to database pods
- Specifying exact ports for communication

Service Mesh Security

Service mesh tools like Istio add extra security to container networks. They check containers before letting them connect and keep their messages private. This makes sure only the right containers can talk to each other.

```
yaml
# Istio Authentication Policy
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: prod
spec:
  mtls:
    mode: STRICT
```

The benefits include:

- Encrypted communication between services
- Automatic certificate rotation
- Traffic monitoring and control
- Enhanced access control

Load Balancer Security

Secure your load balancers to protect incoming traffic:

```
yaml
# Secure Ingress Configuration
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: secure-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect:
"true"
spec:
  tls:
  - hosts:
    - secure-app.example.com
    secretName: tls-secret
  rules:
  - host: secure-app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 443
```

This configuration:

- Forces HTTPS connections
- Implements TLS termination
- Specifies allowed hosts
- Defines secure routing rules

Implementing Network Controls

Let's build comprehensive network security controls:

Microsegmentation

Create granular network policies for each application component:

```
yaml
# Fine-grained network policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-network
  namespace: your-namespace  # Specify the namespace
spec:
  podSelector:
    matchLabels:
      app: api
  policyTypes:
  - Ingress
  - Egress
  ingress:
```

```
- from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 8080
```

API Security

Protect your APIs with authentication and rate limiting:

```
yaml
# API Gateway configuration
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: api-gateway
spec:
  hosts:
  - api.example.com
  http:
  - route:
    - destination:
        host: api-service
```

```
fault:
      delay:
        percentage:
          value: 100
        fixedDelay: 5s
    match:
    - headers:
        api-key:
          exact: valid-key
```

Monitoring Network Traffic

Implement comprehensive network monitoring:

```
yaml
# Network monitoring configuration
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: network-monitor
spec:
  selector:
    matchLabels:
      app: network-metrics
  endpoints:
  - port: metrics
    interval: 15s
```

This setup allows you to:

- Track network patterns
- Detect anomalies
- Monitor bandwidth usage
- Alert on suspicious activity

Network Security Best Practices

Default Deny Policies: Start with zero trust and explicitly allow required communication:

```
yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: your-namespace  # Specify the namespace
spec:
  podSelector: {}  # Selects all pods in the namespace
  policyTypes:
  - Ingress
  - Egress
```

Regular Security Audits:

Implement automated network security scanning:bash

```
# Example network security scan
kubectl-network-policy-advisor analyze \
  --namespace production \
  --report-file network-audit.json
```

Encryption Everywhere:

- Ensure all network traffic is encrypted:
- Use TLS for external communication
- Implement mutual TLS between services
- Encrypt sensitive data at rest

Remember: Network security in containerized environments requires constant attention and updates. Monitor your network traffic patterns, regularly review security policies, and stay informed about new security threats and mitigation strategies.

CHAPTER 5

Advanced Container Security Techniques and Tools

Advanced Security Controls

Runtime Application Self-Protection (RASP)

```
yaml
# Example Falco custom rule
- rule: Unexpected Child Process
  desc: Container spawned unexpected process
  condition: >
    container and
    proc.name != "app" and
    proc.name != "node"
  output: "Unexpected process spawned (proc=%proc.name
%proc.cmdline)"
  priority: WARNING
```

Behavioral Analysis Implementation

```
Python
# Example behavioral monitoring
from kubernetes import client, config, watch

def analyze_pod_metrics(pod):
    # Implement your logic to analyze the pod's
behavior
    print(f"Analyzing pod: {pod.metadata.name} in
namespace: {pod.metadata.namespace}")

def monitor_pod_behavior():
    # Load the Kubernetes configuration
    config.load_kube_config()  # Use
config.load_incluster_config() if running inside a
cluster

    v1 = client.CoreV1Api()
    w = watch.Watch()

    try:
        for event in
w.stream(v1.list_pod_for_all_namespaces):
            pod = event['object']
            analyze_pod_metrics(pod)
    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        w.stop()

if __name__ == "__main__":
    monitor_pod_behavior()
```

Automated Response Systems

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: security-autoscaler
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: web-app
 minReplicas: 1  # Optional: Specify minimum number of
replicas
 maxReplicas: 5  # Optional: Specify maximum number of
replicas
 metrics:
 - type: Pods
   pods:
     metric:
       name: security_score
     target:
       type: AverageValue
       averageValue: 0.8
```

Machine Learning Integration

Default Deny Policies: Start with zero trust and explicitly allow required communication:

```
Python
# Example ML-based anomaly detection
from sklearn.ensemble import IsolationForest

def detect_anomalies(container_metrics):
    # Initialize the Isolation Forest model
    model = IsolationForest(contamination=0.1)

    # Fit the model and predict anomalies
    predictions = model.fit_predict(container_metrics)

    return predictions
```

Threat Pattern Recognition

```yaml
# ML-enhanced security policy
apiVersion: security.example.com/v1
kind: MLSecurityPolicy
metadata:
  name: ml-threat-detection
spec:
  modelPath: "models/threat-detection"
  thresholds:
    confidence: 0.95
    falsePositiveRate: 0.01
  actions:
    - isolate
    - log
    - alert
```

Blockchain Integration

```yaml
# Container integrity verification using blockchain
apiVersion: blockchain.security/v1
kind: IntegrityCheck
metadata:
  name: container-verify
spec:
  type: ethereum
  smart_contract:
    address: "0x123..."
    method: "verifyIntegrity"
  verification:
    images:
      - repository: production/app
        tag: 3.12
```

Real-time Security Monitoring

```yaml
# Prometheus monitoring configuration
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: security-monitor
spec:
  endpoints:
  - interval: 10s
    path: /metrics
    port: metrics
  selector:
    matchLabels:
      security-monitor: "true"
```

Key Takeaways

- Containers need protection at every step. Think of it like a chain - each part needs to be strong to keep everything safe.
- When building containers, start with safe pieces. Keep checking them as they run, just like you'd check a car during a long trip.
- When lots of containers work together, we need special rules to keep them safe. It's like making sure a big team follows safety rules.
- Container networks need special protection because they share information. We need to make sure only the right messages get through.
- Check your containers often for problems. Fix little issues before they become big ones. Regular checks help catch trouble early.
- Let smart tools help keep containers safe. These tools can spot problems faster than people can, working day and night to protect things.
- Only let trusted people use and change containers. Keep track of who can do what, like having special keys for different rooms.
- Use special safety tools made just for containers. These tools know how to find and fix container problems quickly.
- Learn how containers really work to protect them better. The more you understand them, the better you can keep them safe.
- New problems show up all the time, so keep learning new ways to protect containers. What works today might not work tomorrow.

Practical DevSecOps

Become a Certified Container Security Expert

Get started >

Demand is high, and spots are limited! Secure your place today!

www.practical-devsecops.com