Securing Containers and Kubernetes

Modern applications run inside containers and orchestrators like Kubernetes.
Misconfigurations in containers or Kubernetes clusters can lead to privilege escalation, data breaches, and compromised infrastructure.

1.1 Security Risks in Containers

- Using outdated or vulnerable base images
- Containers running as root
- Exposed ports without proper firewall rules
- Hardcoded secrets inside images
- Images pulled from untrusted public registries
- Insecure Dockerfiles
- Sensitive files copied into images

1.2 Securing Docker Containers

1. Use Minimal Base Images

Prefer:

- alpine
- scratch
- distroless

This reduces attack surface.

2. Avoid Running Containers as Root

Use:

USER 1000

or create a dedicated user.

3. Implement Multi-Stage Builds

Removes unnecessary tools and dependencies.

4. Do Not Store Secrets in Images

Use:

- Vault
- AWS Secrets Manager
- Kubernetes Secrets

5. Use Read-only File Systems

docker run --read-only

6. Limit Container Capabilities

Drop privileged capabilities using:

--cap-drop ALL

7. Sign and Verify Images

Use:

- Cosign
- Notary v2

1.3 Securing Kubernetes Clusters

1. RBAC (Role-Based Access Control)

Assign minimal permissions to:

- users
- service accounts
- namespaces

2. Use Network Policies

Restrict pod-to-pod communication:

- deny-by-default
- allow only required traffic

3. Use Pod Security Standards (PSS)

Avoid:

- privileged pods
- host network access
- host PID/IPC

4. Enable Audit Logging

Maintain logs for:

- API activity
- cluster changes

5. Secure Kubelet and API Server

Disable insecure ports and enforce TLS everywhere.

6. Protect ETCD

Encrypt ETCD and restrict access to admin nodes only.

7. Use Secrets Safely

Store secrets in:

- Kubernetes Secrets with encryption-at-rest
- External secret managers (Vault, AWS/GCP/Azure Key Vault)

8. ImagePullPolicy

Always pull updated secure images using:

imagePullPolicy: Always

9. Limit Resource Usage

Use:

- Requests
- Limits
  To avoid DoS attacks inside the cluster.

---

2. Scanning Container Images for Vulnerabilities

Container images often include:

- OS-level packages
- Dependencies
- Framework libraries

These may contain publicly known vulnerabilities (CVEs).

2.1 Why Container Scanning is Needed

- Vulnerabilities in base images (Ubuntu, Alpine)
- Outdated libraries inside the image
- Minimal visibility into transitive components
- Compliance and audit requirements

2.2 Common Container Scanning Tools

Open-Source

- Trivy
- Clair

- Grype

Commercial

- Aqua Trivy Enterprise
- Prisma Cloud
- Anchore Enterprise
- Tenable.io

2.3 Container Scanning Workflow

1. Build Image

docker build -t myapp:latest .

2. Scan Image

Using Trivy:

trivy image myapp:latest

Output includes:

- List of CVEs
- Severity: Critical/High/Medium/Low
- Package details
- Recommended fixes

3. Integrate into Pipeline

Example GitHub Action:

```
- name: Scan container
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: 'myapp:latest'
```

4. Fail Build When Vulnerabilities Found

Use thresholds:

- Fail when high or critical vulnerabilities detected

5. Fix and Rebuild

Update:

- base images
- dependency versions

- OS packages

2.4 Secure Image Best Practices

- Use FROM alpine or distroless images
- Pin image tags (avoid latest)
- Run vulnerability scans after every build
- Use private container registries
- Remove build-time dependencies in final images

---

3. Cloud Security Basics

Cloud environments introduce new challenges and require shared responsibility among cloud providers and users.

3.1 Shared Responsibility Model

Cloud Provider (AWS, Azure, GCP) is responsible for:

- Physical infrastructure
- Hardware security
- Network backbone
- Virtualization layer

Customer is responsible for:

- Application configuration
- Data protection
- Identity and access management
- Network configurations
- OS-level security
- Container and Kubernetes security

3.2 Key Cloud Security Concepts

1. Identity and Access Management (IAM)

- Follow least privilege
- Use roles instead of long-lived users
- Apply MFA

- Rotate access keys

2. Network Security

- Use VPCs, subnets, security groups
- Restrict public access
- Apply NACLs
- Use firewalls and WAFs

3. Data Security

- Enable encryption in transit (TLS)
- Enable encryption at rest (KMS)
- Use secure storage options
- Do not expose buckets/blob containers publicly

4. Logging and Monitoring

Use:

- CloudTrail (AWS)
- Monitor (Azure)
- Stackdriver (GCP)
- GuardDuty / Security Center
- Centralized SIEM

5. Key Management

Use:

- AWS KMS
- Azure Key Vault
- GCP KMS
  Avoid storing secrets in code or containers.

6. Patching and Updates

Regularly update:

- EC2 VMs
- Containers
- OS packages
- Serverless dependencies

3.3 Cloud Misconfigurations to Avoid

- Public S3 buckets
- Open security groups
- Unrestricted RDP/SSH access
- Over-permissive IAM roles
- No logging/audit trails
- No encryption enabled
- Default VPC usage
- Exposed databases

Automating Compliance Checks (e.g., GDPR)

Compliance automation is a key DevSecOps capability that ensures applications, infrastructure, and processes continuously meet regulatory and organizational standards.
Instead of performing manual audits, compliance checks are integrated directly into CI/CD pipelines and cloud environments.

1.1 What Compliance Means in DevSecOps

Compliance refers to adhering to standards, regulations, and frameworks such as:

- GDPR (General Data Protection Regulation)
- PCI-DSS (Payment Card Industry Data Security Standard)
- HIPAA (Health Insurance Portability and Accountability Act)
- ISO 27001
- NIST 800-53
- SOC2
- CIS Benchmarks
- Internal enterprise policies

In DevSecOps, compliance becomes:

- Automated
- Continuous
- Transparent
- Enforced by code

1.2 Why Automate Compliance?

1. Manual audits are slow and prone to error
2. Cloud environments change frequently
3. Continuous deployment demands continuous compliance
4. Automated checks detect deviations immediately
5. Enforces security and data protection at scale
6. Makes organizations "audit-ready" at all times

1.3 Areas Where Compliance Checks Are Automated

A. Infrastructure Compliance

Tools scan:

- AWS, Azure, GCP resources
- S3 bucket exposure
- Encryption settings
- IAM policies
- Security groups
- Logging and monitoring

B. Application Compliance

Automated checks for:

- Sensitive data handling
- Data encryption in transit and at rest
- Data masking/redaction
- Secure session management

Example: GDPR Article 32 (security of processing)

C. CI/CD Pipeline Compliance

Pipelines enforce controls such as:

- SAST, SCA & DAST
- Secrets detection
- Security gates
- Policy-as-code validation

D. Data Protection Compliance

Checks ensure:

- No unauthorized data movement
- Logs do not leak personal data
- Backups are encrypted
- Retention policies are enforced

E. Container & Kubernetes Compliance

Standards like:

- CIS Kubernetes Benchmark
- NIST container guidelines

Automated tools detect misconfigurations.

1.4 Tools for Automated Compliance

Policy-as-Code Tools

- OPA (Open Policy Agent)
- Conftest
- HashiCorp Sentinel

Cloud Compliance Tools

- AWS Config
- AWS Security Hub
- AWS Macie (for GDPR/PII detection)
- Azure Policy
- GCP Security Command Center

Commercial Platforms

- Prisma Cloud
- Wiz
- Lacework
- Tenable
- Qualys

Open Source

- kube-bench
- Cloud Custodian

- Checkov (with compliance rules built-in)

1.5 Compliance in CI/CD

Compliance rules run as automated steps:

Commit → Build → Compliance Scan → Deploy

If violations found:

- Build fails
- Deployment blocked
- Issue sent to developers and security team

---

2. Introduction to Threat Modeling

Threat modeling is the structured process of identifying potential threats, vulnerabilities, and risks to an application or system **before** they occur.

It is a core shift-left practice in DevSecOps.

2.1 Purpose of Threat Modeling

1. Identify what can go wrong in the design
2. Understand how attackers may compromise the system
3. Prioritize high-risk components
4. Build secure-by-design architecture
5. Reduce future vulnerabilities

Threat modeling helps teams build **proactive** security, not reactive.

2.2 When to Perform Threat Modeling

- During design/architecture phase
- Before writing code
- For major new features
- When handling sensitive data
- When integrating external systems
- When moving to cloud or containers

2.3 Approaches to Threat Modeling

1. STRIDE Model (Most Common)

STRIDE stands for:

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

Each system component is evaluated for these risks.

---

2. DREAD Model

Used for scoring threats:

- Damage
- Reproducibility
- Exploitability
- Affected users
- Discoverability

---

3. Attack Trees

Represents how an attacker could exploit a system using a tree structure.

---

4. Data Flow Diagrams (DFDs)

Visualizes:

- Data inputs/outputs
- Processes
- Trust boundaries
- External systems

Used with STRIDE for accurate assessment.

2.4 Threat Modeling Workflow

1. **Define the System**
   Architecture, components, data flows.
2. **Identify Threats**
   Using STRIDE, attack trees, or MITRE ATT&CK.
3. **Identify Vulnerabilities**
   Design weaknesses, insecure APIs, misconfigurations.
4. **Assign Severity**
   Using risk matrix or DREAD scoring.
5. **Propose Mitigations**
   Controls, secure patterns, redesign.
6. **Validate and Iterate**
   Re-model after major changes.

2.5 Tools for Threat Modeling

Open-Source

- OWASP Threat Dragon
- Draw.io with templates

Commercial

- Microsoft Threat Modeling Tool
- IriusRisk

2.6 Practical Example

A web application handling customer data:

Threats:

- SQL injection (Tampering)
- Session hijacking (Spoofing)
- Sensitive data leakage (Information Disclosure)
- DDoS attacks (Denial of Service)

Mitigations:

- Parameterized queries
- Secure session tokens
- Encryption at rest
- Rate limiting