

CKS Exam Study Reminder Guide

Quick Reference for Certified Kubernetes Security Specialist (CKS) Exam

Exam Duration: 2 hours | **Questions:** 15-20 | **Passing Score:** 67%

Format: Performance-based (hands-on)

Table of Contents

1. [NetworkPolicy - Default Deny & Allow](#)
 2. [CIS Benchmark & kube-bench](#)
 3. [RBAC - Role, RoleBinding, ClusterRole](#)
 4. [ServiceAccount Security](#)
 5. [AppArmor Profiles](#)
 6. [Seccomp Profiles](#)
 7. [Pod Security Admission \(PSA\)](#)
 8. [Secrets Encryption at Rest](#)
 9. [SecurityContext Hardening](#)
 10. [Trivy Image Scanning](#)
 11. [Kubesecc Static Analysis](#)
 12. [Falco Runtime Security](#)
 13. [Kubernetes Audit Logs](#)
 14. [RuntimeClass & gVisor Sandbox](#)
 15. [ImagePolicyWebhook](#)
 16. [Binary Verification](#)
 17. [Node Metadata Protection](#)
 18. [Ingress TLS Configuration](#)
-

Exam Aliases - Set These First!

```
alias k=kubectl
alias kn='kubectl config set-context --current --namespace'
export do="--dry-run=client -o yaml"
source <(kubectl completion bash)
complete -o default -F __start_kubectl k
```

1. NetworkPolicy

Theory

NetworkPolicy is a Kubernetes resource that controls traffic flow between pods. By default, all pods can communicate with each other. NetworkPolicies implement a **zero-trust network model** by:

- Controlling **ingress** (incoming) traffic
- Controlling **egress** (outgoing) traffic
- Using **label selectors** to target pods
- Specifying allowed **ports** and **protocols**

Key Concepts:

- Empty **podSelector**: `{}` = selects ALL pods in namespace
 - Empty **policyTypes** array = only ingress if ingress rules exist
 - No ingress/egress rules = blocks that traffic type entirely
 - DNS typically uses port 53 UDP (and TCP)
-

Steps: Default Deny All Traffic

Scenario: Create a policy that blocks ALL ingress and egress traffic for all pods in a namespace.

Step 1: Create the namespace

```
kubectl create namespace isolated-ns
```

Step 2: Create the NetworkPolicy

```
# /opt/course/01/netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: isolated-ns
spec:
  podSelector: {}           # Applies to ALL pods
  policyTypes:
    - Ingress
    - Egress
  # No ingress/egress rules = deny all
```

Step 3: Apply and verify

```
kubectl apply -f /opt/course/01/netpol.yaml  
kubectl get netpol -n isolated-ns
```

Steps: Allow Specific Traffic (Multi-tier App)

Scenario: Allow frontend → API on port 8080, API → database on port 5432, DNS everywhere.

API Policy (allows ingress from frontend, egress to database)

```
# /opt/course/02/api-netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-policy
  namespace: microservices-ns
spec:
  podSelector:
    matchLabels:
      tier: api
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              tier: frontend
        ports:
          - protocol: TCP
            port: 8080
      - from:
          - namespaceSelector:
              matchLabels:
                name: monitoring-ns
            ports:
              - protocol: TCP
                port: 8080
  egress:
    - to:
        - podSelector:
            matchLabels:
              tier: database
        ports:
          - protocol: TCP
            port: 5432
    - ports:                                     # DNS egress
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

Database Policy (allows ingress only from API)

```
# /opt/course/02/db-netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-policy
  namespace: microservices-ns
spec:
  podSelector:
    matchLabels:
      tier: database
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              tier: api
        ports:
          - protocol: TCP
            port: 5432
  egress:
    - ports: # DNS only
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

Test connectivity

```
# From frontend pod - should fail to reach database directly
kubectl exec -n microservices-ns frontend-pod -- wget -q0- --timeout=2
database:5432

# From API pod - should reach database
kubectl exec -n microservices-ns api-pod -- wget -q0- --timeout=2
database:5432
```

Key Points to Remember

Element	Meaning
<code>podSelector: {}</code>	All pods in namespace
<code>namespaceSelector: {}</code>	All namespaces
<code>policyTypes: [Ingress, Egress]</code>	Control both directions
No rules under ingress/egress	Deny that traffic type
Port 53 UDP/TCP	DNS - almost always needed for egress

2. CIS Benchmark & kube-bench

Theory

The **CIS Kubernetes Benchmark** is a security configuration guide for Kubernetes clusters. **kube-bench** is a tool that checks cluster configuration against CIS benchmarks automatically.

Key Areas Checked:

- API Server security settings
- Controller Manager configuration
- Scheduler configuration
- etcd security
- Kubelet configuration
- Worker node settings

Steps: Run kube-bench and Fix Issues

Step 1: SSH to control plane and run kube-bench

```
ssh controlplane

# Run kube-bench for master components
kube-bench run --targets=master > /opt/course/03/kube-bench-before.txt

# Or run specific checks
kube-bench run --targets=master --check=1.2.1,1.2.2
```

Step 2: Common fixes for API Server

Edit `/etc/kubernetes/manifests/kube-apiserver.yaml`:

```
spec:
  containers:
  - command:
    - kube-apiserver
    # Security fixes - add/modify these:
    - --anonymous-auth=false           # Disable anonymous access
    - --profiling=false                 # Disable profiling
    - --enable-admission-plugins=NodeRestriction,PodSecurity
    - --audit-log-path=/var/log/apiserver/audit.log
    - --audit-log-maxage=30
    - --audit-log-maxbackup=10
    - --audit-log-maxsize=100
    - --authorization-mode=Node,RBAC    # Never use AlwaysAllow
    # REMOVE if present:
    - --insecure-port=8080              # Must be 0 or removed
```

Step 3: Fix kubelet issues on worker nodes

```
ssh node01

# Edit kubelet config
sudo vim /var/lib/kubelet/config.yaml
```

Common kubelet fixes:

```
# /var/lib/kubelet/config.yaml
authentication:
  anonymous:
    enabled: false           # Disable anonymous auth
  webhook:
    enabled: true
authorization:
  mode: Webhook              # Not AlwaysAllow
readOnlyPort: 0              # Disable read-only port
protectKernelDefaults: true
```

Step 4: Restart kubelet and verify

```
sudo systemctl restart kubelet
kube-bench run --targets=node > /opt/course/03/kube-bench-after.txt
```

Common kube-bench Failures & Fixes

Check	Issue	Fix
1.2.1	Anonymous auth enabled	<code>--anonymous-auth=false</code>
1.2.18	Profiling enabled	<code>--profiling=false</code>
1.2.6	Insecure port open	Remove <code>--insecure-port</code>
4.2.1	Kubelet anonymous auth	<code>anonymous.enabled: false</code>
4.2.2	Kubelet authorization	<code>authorization.mode: Webhook</code>

3. RBAC

Theory

Role-Based Access Control (RBAC) restricts access to Kubernetes resources based on roles assigned to users or service accounts.

Four RBAC Resources:

Resource	Scope	Binds To
Role	Namespace	RoleBinding
ClusterRole	Cluster-wide	ClusterRoleBinding or RoleBinding
RoleBinding	Namespace	Role or ClusterRole
ClusterRoleBinding	Cluster-wide	ClusterRole

Key API Groups:

- `""` (core) = pods, services, secrets, configmaps, namespaces
- `apps` = deployments, daemonsets, replicaset, statefulsets
- `networking.k8s.io` = networkpolicies, ingresses
- `rbac.authorization.k8s.io` = roles, rolebindings

Steps: Create Role & RoleBinding

Step 1: Create ServiceAccount

```
kubectl create namespace cicd-ns
kubectl create serviceaccount deploy-sa -n cicd-ns
```

Or with YAML:

```
# /opt/course/04/sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: deploy-sa
  namespace: cicd-ns
```

Step 2: Create Role

```
# /opt/course/04/role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: deployment-manager
  namespace: cicd-ns
rules:
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["services", "configmaps"]
    verbs: ["get", "list"]
# NOTE: NO secrets access - critical for least privilege
```

Step 3: Create RoleBinding

```
# /opt/course/04/rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: deploy-sa-binding
  namespace: cicd-ns
subjects:
  - kind: ServiceAccount
    name: deploy-sa
    namespace: cicd-ns
roleRef:
  kind: Role
  name: deployment-manager
  apiGroup: rbac.authorization.k8s.io
```

Step 4: Test permissions

```
# Check what the SA can do
kubectl auth can-i create deployments -n cicd-ns --
as=system:serviceaccount:cicd-ns:deploy-sa
# yes

kubectl auth can-i get secrets -n cicd-ns --as=system:serviceaccount:cicd-
ns:deploy-sa
# no
```

Steps: Create ClusterRole & ClusterRoleBinding (Cluster-wide Read-Only)

```
# /opt/course/20/clusterrole.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-monitor
rules:
  - apiGroups: [""]
    resources: ["pods", "nodes", "namespaces", "endpoints", "services",
"events"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["pods/log"]
    verbs: ["get"]
# NO write verbs, NO secrets, NO exec
```

```
# /opt/course/20/clusterrolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-monitor-binding
subjects:
  - kind: ServiceAccount
    name: monitor-sa
    namespace: monitoring
roleRef:
  kind: ClusterRole
  name: cluster-monitor
  apiGroup: rbac.authorization.k8s.io
```

Quick RBAC Commands

```
# Create role imperatively
kubectl create role pod-reader --verb=get,list,watch --resource=pods -n
myns

# Create rolebinding imperatively
kubectl create rolebinding pod-reader-binding --role=pod-reader --
serviceaccount=myns:mysa -n myns

# Create clusterrole
kubectl create clusterrole node-reader --verb=get,list,watch --
resource=nodes

# Create clusterrolebinding
kubectl create clusterrolebinding node-reader-binding --clusterrole=node-
reader --serviceaccount=myns:mysa

# Test permissions
kubectl auth can-i list pods --as=system:serviceaccount:myns:mysa -n myns
kubectl auth can-i --list --as=system:serviceaccount:myns:mysa -n myns
```

4. ServiceAccount Security

Theory

ServiceAccounts provide an identity for pods. By default, Kubernetes:

- Creates a **default** ServiceAccount in each namespace
- Automatically mounts a token into pods at
`/var/run/secrets/kubernetes.io/serviceaccount/`

Security Best Practices:

- Set **automountServiceAccountToken: false** on ServiceAccount or Pod
- Use dedicated ServiceAccounts with minimal permissions
- Never use the **default** ServiceAccount for applications

Steps: Secure ServiceAccount Configuration

Step 1: Create ServiceAccount with no auto-mount

```
# /opt/course/05/sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: restricted-sa
  namespace: secure-ns
automountServiceAccountToken: false
```

Step 2: Create minimal Role and RoleBinding

```
# /opt/course/05/role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: secure-ns
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list"]
```

```
# /opt/course/05/rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: restricted-sa-binding
  namespace: secure-ns
subjects:
- kind: ServiceAccount
  name: restricted-sa
  namespace: secure-ns
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Step 3: Update deployment to use restricted SA

```
# /opt/course/05/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: insecure-app
  namespace: secure-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: insecure-app
  template:
    metadata:
      labels:
        app: insecure-app
    spec:
      serviceAccountName: restricted-sa
      automountServiceAccountToken: false # Also set at pod level
      containers:
        - name: app
          image: nginx:alpine
```

Step 4: Verify no token mounted

```
kubectl exec -n secure-ns <pod-name> -- ls
/var/run/secrets/kubernetes.io/serviceaccount/
# Should return error - directory doesn't exist
```

5. AppArmor Profiles

Theory

AppArmor (Application Armor) is a Linux Security Module (LSM) that provides Mandatory Access Control (MAC). It restricts what applications can do, including:

- File access (read/write/execute)
- Network operations
- Capability usage

Kubernetes Integration:

- Profiles must be loaded on each node where the pod might run
- Uses annotation (legacy) or securityContext (K8s 1.30+)
- Profiles are node-specific, not cluster-wide

Steps: Apply AppArmor Profile to Pod

Step 1: Check/load profile on node

```
ssh node01

# Check loaded profiles
sudo aa-status | grep k8s-deny-write

# If not loaded, load it
sudo apparmor_parser -r /etc/apparmor.d/k8s-deny-write

# Verify profile is in enforce mode
sudo aa-status
```

Step 2: Create pod with AppArmor profile

```
# /opt/course/06/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
  namespace: apparmor-ns
spec:
  containers:
    - name: secured-container
      image: busybox:1.36
      command: ["sh", "-c", "echo 'AppArmor secured!' && sleep 1h"]
      securityContext:
        appArmorProfile:
          type: Localhost
          localhostProfile: k8s-deny-write
```

Step 3: Apply and verify

```
kubectl apply -f /opt/course/06/pod.yaml

# Test - should fail to write
kubectl exec -n apparmor-ns secured-pod -- touch /tmp/test
# Permission denied
```

AppArmor Profile Types

Type	Description
RuntimeDefault	Container runtime's default profile
Localhost	Custom profile loaded on node
Unconfined	No AppArmor restrictions

6. Seccomp Profiles

Theory

Seccomp (Secure Computing Mode) filters system calls a process can make to the kernel. It reduces the attack surface by:

- Allowing only necessary syscalls
- Blocking dangerous syscalls (e.g., `ptrace`, `mount`)
- Logging blocked syscall attempts

Profile Locations:

- Custom profiles: `/var/lib/kubelet/seccomp/`
- Default: Defined by container runtime

Steps: Apply Seccomp Profiles

Step 1: Create pod with RuntimeDefault seccomp

```
# /opt/course/07/runtime-default-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: runtime-default-pod
  namespace: seccomp-ns
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: nginx
      image: nginx:alpine
```

Step 2: Create pod with custom Localhost seccomp profile

```
# /opt/course/07/custom-seccomp-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: custom-seccomp-pod
  namespace: seccomp-ns
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: audit-log.json # Path relative to
/var/lib/kubelet/seccomp/
  containers:
  - name: nginx
    image: nginx:alpine
```

Step 3: Verify seccomp is applied

```
kubectl apply -f /opt/course/07/runtime-default-pod.yaml
kubectl apply -f /opt/course/07/custom-seccomp-pod.yaml

# Check pod is running
kubectl get pods -n seccomp-ns

# Verify via crictl on the node
ssh node01
sudo crictl inspect <container-id> | grep -i seccomp
```

Seccomp Profile Types

Type	Description
RuntimeDefault	Runtime's default profile (recommended)
Localhost	Custom profile at <code>/var/lib/kubelet/seccomp/<profile></code>
Unconfined	No seccomp restrictions

7. Pod Security Admission (PSA)

Theory

Pod Security Admission enforces Pod Security Standards at the namespace level. It replaced PodSecurityPolicy (deprecated).

Three Modes:

Mode	Action
<code>enforce</code>	Reject violating pods
<code>warn</code>	Allow but warn
<code>audit</code>	Log to audit log

Three Levels:

Level	Description
<code>privileged</code>	Unrestricted (default)
<code>baseline</code>	Minimally restrictive, prevents known privilege escalations
<code>restricted</code>	Heavily restricted, security best practices

Steps: Configure PSA Restricted Namespace

Step 1: Create namespace with PSA labels

```
apiVersion: v1
kind: Namespace
metadata:
  name: psa-restricted
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/audit: restricted
```

Or imperatively:

```
kubectl create namespace psa-restricted
kubectl label namespace psa-restricted \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/warn=restricted \
  pod-security.kubernetes.io/audit=restricted
```

Step 2: Create PSA-compliant pod

```
# /opt/course/08/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
  namespace: psa-restricted
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: nginx
      image: nginx:alpine
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        runAsNonRoot: true
        runAsUser: 1000
        capabilities:
          drop:
            - ALL
      volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: cache
          mountPath: /var/cache/nginx
        - name: run
          mountPath: /var/run
  volumes:
    - name: tmp
      emptyDir: {}
    - name: cache
      emptyDir: {}
    - name: run
      emptyDir: {}
```

Step 3: Test that non-compliant pods are rejected

```
# This will fail
kubectl run test --image=nginx -n psa-restricted
# Error: violates "restricted" policy

# Save the error message
kubectl run test --image=nginx -n psa-restricted 2>&1 >
/opt/course/08/rejected-error.txt
```

PSA Restricted Requirements

A **restricted** pod must:

- Run as non-root (`runAsNonRoot: true`)
- Have seccomp profile (`RuntimeDefault` or `Localhost`)
- Drop all capabilities (`capabilities.drop: ["ALL"]`)
- Disable privilege escalation (`allowPrivilegeEscalation: false`)
- Not use hostNetwork, hostPID, hostIPC
- Not use hostPath volumes
- Not run privileged containers

8. Secrets Encryption at Rest

Theory

By default, Kubernetes stores Secrets in etcd as **base64-encoded plaintext**. Encryption at rest protects Secrets by encrypting them before writing to etcd.

Encryption Providers:

Provider	Description
<code>identity</code>	No encryption (default)
<code>aescbc</code>	AES-CBC encryption (recommended)
<code>aesgcm</code>	AES-GCM encryption
<code>secretbox</code>	XSalsa20 + Poly1305
<code>kms</code>	External KMS provider

Steps: Enable Secrets Encryption

Step 1: Generate encryption key

```
# Generate a 32-byte key and base64 encode it
head -c 32 /dev/urandom | base64
```

Step 2: Create EncryptionConfiguration

```
# /etc/kubernetes/encryption-config.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
  providers:
    - aescbc:
        keys:
          - name: key1
            secret: <base64-encoded-32-byte-key>
    - identity: {} # Fallback for reading unencrypted secrets
```

Step 3: Configure API server

Edit `/etc/kubernetes/manifests/kube-apiserver.yaml`:

```
spec:
  containers:
    - command:
        - kube-apiserver
        - --encryption-provider-config=/etc/kubernetes/encryption-config.yaml
        # ... other flags
      volumeMounts:
        - name: encryption-config
          mountPath: /etc/kubernetes/encryption-config.yaml
          readOnly: true
      volumes:
        - name: encryption-config
          hostPath:
            path: /etc/kubernetes/encryption-config.yaml
            type: File
```

Step 4: Wait for API server restart

```
# Watch for API server to restart
watch crictl ps | grep kube-apiserver

# Or check pods
kubectl get pods -n kube-system | grep api
```

Step 5: Re-encrypt existing secrets

```
# Re-encrypt all secrets
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Step 6: Verify encryption in etcd

```
# Check secret in etcd (should be encrypted, not plaintext)
ETCDCTL_API=3 etcdctl \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/secrets-ns/my-secret | hexdump -C

# Encrypted secrets start with "k8s:enc:aescbc:v1:"
```

9. SecurityContext Hardening

Theory

SecurityContext defines privilege and access control settings for pods and containers. It's the primary way to harden container security.

Pod-level vs Container-level:

- Pod-level: Applies to all containers
- Container-level: Overrides pod-level for specific container

Steps: Create Fully Hardened Pod

```
# /opt/course/10/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: hardened-pod
  namespace: hardened-ns
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
    fsGroup: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: nginx
      image: nginx:alpine
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        capabilities:
          drop:
            - ALL
        # Add back only what's needed (usually none for nginx)
      volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: cache
          mountPath: /var/cache/nginx
        - name: run
          mountPath: /var/run
  volumes:
    - name: tmp
      emptyDir: {}
    - name: cache
      emptyDir: {}
    - name: run
      emptyDir: {}
```

SecurityContext Quick Reference

Field	Purpose	Recommended Value
<code>runAsNonRoot</code>	Prevent root execution	<code>true</code>
<code>runAsUser</code>	Specific UID	<code>1000</code> or higher
<code>readOnlyRootFilesystem</code>	Prevent writes	<code>true</code>
<code>allowPrivilegeEscalation</code>	Prevent sudo/setuid	<code>false</code>
<code>capabilities.drop</code>	Remove Linux capabilities	<code>["ALL"]</code>
<code>seccompProfile.type</code>	Syscall filtering	<code>RuntimeDefault</code>
<code>privileged</code>	Full host access	<code>false</code> (or omit)

10. Trivy Image Scanning

Theory

Trivy is a vulnerability scanner for container images, file systems, and git repositories. It detects:

- OS package vulnerabilities (CVEs)
- Application dependency vulnerabilities
- Misconfigurations
- Secrets in images

Severity Levels: CRITICAL, HIGH, MEDIUM, LOW, UNKNOWN

Steps: Scan and Compare Images

Step 1: Scan images for HIGH and CRITICAL vulnerabilities

```
# Scan nginx images
trivy image --severity HIGH,CRITICAL nginx:1.19 > /opt/course/11/nginx-1.19-scan.txt
trivy image --severity HIGH,CRITICAL nginx:1.25-alpine > /opt/course/11/nginx-1.25-alpine-scan.txt

# Scan python images
trivy image --severity HIGH,CRITICAL python:3.8 > /opt/course/11/python-3.8-scan.txt
trivy image --severity HIGH,CRITICAL python:3.12-alpine > /opt/course/11/python-3.12-alpine-scan.txt
```

Step 2: Count vulnerabilities

```
# Quick count
trivy image --severity HIGH,CRITICAL -q nginx:1.19 | grep "Total:"
```

Step 3: Document recommendations

```
# /opt/course/11/recommendations.txt
# Recommended nginx: nginx:1.25-alpine (fewer vulns, smaller image)
# Recommended python: python:3.12-alpine (newer version, fewer vulns)
```

Step 4: Update deployment with safer images

```
kubectl set image deployment/web-app nginx=nginx:1.25-alpine -n trivy-test
```

Trivy Quick Commands

```
# Basic scan
trivy image nginx:latest

# Only HIGH and CRITICAL
trivy image --severity HIGH,CRITICAL nginx:latest

# JSON output
trivy image -f json nginx:latest > scan.json

# Scan and fail if vulns found (for CI/CD)
trivy image --exit-code 1 --severity CRITICAL nginx:latest

# Ignore unfixed vulnerabilities
trivy image --ignore-unfixed nginx:latest
```

11. Kubesec Static Analysis

Theory

Kubesec analyzes Kubernetes manifests for security risks and provides a score. Higher scores indicate better security posture.

Score System:

- Positive points for security features (e.g., `runAsNonRoot`)
 - Critical issues can result in negative scores
 - Target score: 8+ for production workloads
-

Steps: Analyze and Fix Deployment

Step 1: Scan insecure deployment

```
kubesecc scan /opt/course/12/insecure-deploy.yaml > /opt/course/12/kubesecc-report.json
```

```
# Or use the online API
```

```
curl -sSX POST --data-binary @/opt/course/12/insecure-deploy.yaml \
  https://v2.kubesecc.io/scan
```

Step 2: Review and fix issues

Common fixes to improve score:

```
# /opt/course/12/secure-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-app
  namespace: kubesecc-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: secure-app
  template:
    metadata:
      labels:
        app: secure-app
    spec:
      automountServiceAccountToken: false # +1 point
      containers:
        - name: app
          image: nginx:alpine
          securityContext:
            runAsNonRoot: true # +1 point
            runAsUser: 1000 # +1 point
            readOnlyRootFilesystem: true # +1 point
            allowPrivilegeEscalation: false # +1 point
            capabilities:
              drop:
                - ALL # +1 point
          resources: # +1 point
            limits:
              memory: "128Mi"
              cpu: "500m"
            requests:
              memory: "64Mi"
              cpu: "250m"
```

Step 3: Verify improved score

```
kubesecc scan /opt/course/12/secure-deploy.yaml > /opt/course/12/kubesecc-
fixed.json
# Score should be >= 8
```

Kubesecc Scoring Items

Feature	Points	Description
<code>runAsNonRoot: true</code>	+1	Non-root user
<code>runAsUser > 10000</code>	+1	High UID
<code>readOnlyRootFilesystem: true</code>	+1	Read-only FS
<code>capabilities.drop: ALL</code>	+1	Drop caps
<code>resources.limits</code>	+1	Resource limits
<code>automountServiceAccountToken: false</code>	+1	No SA token
<code>serviceAccountName</code>	+3	Custom SA
<code>privileged: true</code>	Critical!	Never use

12. Falco Runtime Security

Theory

Falco is a runtime security tool that detects anomalous activity in containers and hosts. It uses rules to identify:

- Shell spawned in container
- Sensitive file access
- Unexpected network connections
- Privilege escalation attempts

Rule Components:

- **rule**: Name of the rule
- **desc**: Description
- **condition**: When to trigger (uses Sysdig syntax)
- **output**: What to log
- **priority**: Severity level

Steps: Create Custom Falco Rule

Step 1: SSH to node with Falco

```
ssh node01
```

Step 2: Create custom rule file

```
# /etc/falco/rules.d/shell-detect.yaml
- rule: Shell Spawned in Container
  desc: Detect shell processes spawned inside containers
  condition: >
    spawned_process and
    container and
    proc.name in (bash, sh, ash, dash, zsh)
  output: >
    Shell spawned in container
    (user=%user.name command=%proc.cmdline container=%container.name
    pod=%k8s.pod.name ns=%k8s.ns.name)
  priority: WARNING
  tags: [container, shell, mitre_execution]
```

Step 3: Restart Falco

```
sudo systemctl restart falco
```

Step 4: Trigger the rule

```
# From control plane, exec into a pod
kubectl exec -it <pod-name> -- /bin/sh
```

Step 5: Check Falco logs

```
# On node01
sudo journalctl -u falco | grep "Shell Spawned"

# Or tail the log
sudo tail -f /var/log/falco.log | grep -i shell
```

Step 6: Get container ID

```
# Use crictl to find container
sudo crictl ps | grep <pod-name>

# Save container ID
echo "<container-id>" > /opt/course/13/container-id.txt
```

Falco Macros Quick Reference

Macro	Meaning
<code>spawned_process</code>	New process created
<code>container</code>	Event from container (not host)
<code>proc.name</code>	Process name
<code>proc.cmdline</code>	Full command line
<code>user.name</code>	User that ran the command
<code>container.name</code>	Container name
<code>k8s.pod.name</code>	Kubernetes pod name
<code>k8s.ns.name</code>	Kubernetes namespace

Falco Priority Levels

Priority	Use Case
EMERGENCY	System unusable
ALERT	Immediate action needed
CRITICAL	Critical condition
ERROR	Error condition
WARNING	Warning condition
NOTICE	Normal but significant
INFO	Informational
DEBUG	Debug messages

13. Kubernetes Audit Logs

Theory

Audit logging records all requests to the Kubernetes API server. It captures:

- Who made the request (user, service account)
- What was requested (verb, resource)
- When it happened (timestamp)
- Whether it succeeded

Audit Levels:

Level	Data Recorded
None	Don't log
Metadata	Request metadata only
Request	Metadata + request body
RequestResponse	Metadata + request + response body

Steps: Configure Audit Logging

Step 1: Create audit policy

```
# /etc/kubernetes/audit-policy.yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  # Log all secrets at RequestResponse level
  - level: RequestResponse
    resources:
      - group: ""
        resources: ["secrets"]

  # Log pod operations at Metadata level
  - level: Metadata
    resources:
      - group: ""
        resources: ["pods", "pods/log"]

  # Don't log specific configmaps
  - level: None
    resources:
      - group: ""
        resources: ["configmaps"]
        resourceNames: ["controller-leader"]

  # Don't log kube-proxy watch requests
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
      - group: ""
        resources: ["endpoints", "services"]

  # Log kube-system configmap/secret changes
  - level: Request
    namespaces: ["kube-system"]
    resources:
      - group: ""
        resources: ["configmaps", "secrets"]
    verbs: ["create", "update", "patch", "delete"]

  # Catch-all: log everything else at Metadata
  - level: Metadata
    omitStages:
      - RequestReceived
```

Step 2: Configure API server

Edit `/etc/kubernetes/manifests/kube-apiserver.yaml`:

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --audit-policy-file=/etc/kubernetes/audit-policy.yaml
    - --audit-log-path=/var/log/kubernetes/audit/audit.log
    - --audit-log-maxage=8
    - --audit-log-maxbackup=3
    - --audit-log-maxsize=9
    # ... other flags
  volumeMounts:
  - name: audit-policy
    mountPath: /etc/kubernetes/audit-policy.yaml
    readOnly: true
  - name: audit-log
    mountPath: /var/log/kubernetes/audit
  volumes:
  - name: audit-policy
    hostPath:
      path: /etc/kubernetes/audit-policy.yaml
      type: File
  - name: audit-log
    hostPath:
      path: /var/log/kubernetes/audit
      type: DirectoryOrCreate
```

Step 3: Create directory and wait for restart

```
mkdir -p /var/log/kubernetes/audit

# Wait for API server to restart
watch crictl ps | grep kube-apiserver
```

Step 4: Test and find audit entry

```
# Create a secret
kubectl create namespace audit-test
kubectl create secret generic test-secret --from-literal=key=value -n
audit-test

# Find the audit log entry
grep "test-secret" /var/log/kubernetes/audit/audit.log | tail -1 >
/opt/course/14/secret-audit.log
```

Audit Policy Tips

- **Order matters:** First matching rule wins
 - Use `omitStages: ["RequestReceived"]` to reduce noise
 - `None` level for high-frequency, low-value events
 - `RequestResponse` for secrets (to see what was accessed)
 - Always have a catch-all rule at the end
-

14. RuntimeClass & gVisor Sandbox

Theory

RuntimeClass allows different container runtimes for different workloads. **gVisor** is a container sandbox that provides an additional layer of isolation by intercepting system calls.

When to use gVisor:

- Untrusted workloads
 - Multi-tenant environments
 - Defense in depth for sensitive applications
-

Steps: Use gVisor RuntimeClass

Step 1: Verify RuntimeClass exists

```
kubectl get runtimeclass gvisor
```

Step 2: Create pod with gVisor runtime

```
# /opt/course/15/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sandboxed-pod
  namespace: sandbox-ns
spec:
  runtimeClassName: gvisor      # Use gVisor runtime
  containers:
    - name: nginx
      image: nginx
```

Step 3: Apply and verify

```
kubectl apply -f /opt/course/15/pod.yaml

# Verify pod is running
kubectl get pod sandboxed-pod -n sandbox-ns

# Check runtime class is applied
kubectl get pod sandboxed-pod -n sandbox-ns -o
jsonpath='{.spec.runtimeClassName}'
```

Step 4: Verify gVisor is working

```
# Exec into pod and check kernel (should be gVisor, not host)
kubectl exec -n sandbox-ns sandboxed-pod -- dmesg | head
# Should show gVisor kernel, not Linux
```

RuntimeClass YAML

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc    # The CRI handler name
```

15. ImagePolicyWebhook

Theory

ImagePolicyWebhook is an admission controller that validates images against an external webhook service before allowing pod creation. Used to enforce:

- Allowed registries (e.g., only internal registry)
- Image signing verification
- Vulnerability scan requirements

Steps: Configure ImagePolicyWebhook

Step 1: Create webhook kubeconfig

```
# /etc/kubernetes/admission/image-policy-kubeconfig.yaml
apiVersion: v1
kind: Config
clusters:
  - name: image-policy-webhook
    cluster:
      certificate-authority: /etc/kubernetes/pki/image-policy/ca.crt
      server: https://image-policy-webhook.image-policy.svc:443
contexts:
  - name: image-policy-webhook
    context:
      cluster: image-policy-webhook
      user: image-policy-webhook
current-context: image-policy-webhook
users:
  - name: image-policy-webhook
    user: {}
```

Step 2: Create admission configuration

```
# /etc/kubernetes/admission/admission-config.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
  - name: ImagePolicyWebhook
    configuration:
      imagePolicy:
        kubeConfigFile: /etc/kubernetes/admission/image-policy-
kubeconfig.yaml
        allowTTL: 50
        denyTTL: 50
        retryBackoff: 500
        defaultAllow: false # DENY if webhook unavailable
```

Step 3: Configure API server

Edit `/etc/kubernetes/manifests/kube-apiserver.yaml`:

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --enable-admission-plugins=NodeRestriction,ImagePolicyWebhook
    - --admission-control-config-file=/etc/kubernetes/admission/admission-
config.yaml
    # ... other flags
    volumeMounts:
    - name: admission-config
      mountPath: /etc/kubernetes/admission
      readOnly: true
    - name: image-policy-certs
      mountPath: /etc/kubernetes/pki/image-policy
      readOnly: true
  volumes:
  - name: admission-config
    hostPath:
      path: /etc/kubernetes/admission
      type: Directory
  - name: image-policy-certs
    hostPath:
      path: /etc/kubernetes/pki/image-policy
      type: Directory
```

Step 4: Test the webhook

```
# Wait for API server to restart
watch crictl ps | grep kube-apiserver

# Test allowed image (depends on webhook policy)
kubectl run test --image=myregistry.io/nginx

# Test denied image
kubectl run test2 --image=docker.io/nginx
# Should be denied by webhook
```

16. Binary Verification

Theory

Binary verification ensures Kubernetes binaries haven't been tampered with. Verify by comparing SHA512 checksums of installed binaries against official release checksums.

Steps: Verify Kubernetes Binaries

Step 1: Identify cluster version

```
kubectl version --short  
# or  
kubectl get nodes -o wide
```

Step 2: Download official checksums

```
VERSION=$(kubectl version -o json | jq -r '.serverVersion.gitVersion')  
  
# Download official checksum file  
curl -LO "https://dl.k8s.io/${VERSION}/bin/linux/amd64/kubectl.sha256"  
# or SHA512  
curl -LO  
"https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl.sha512"
```

Step 3: Calculate local binary checksum

```
# Calculate SHA512 of local kubectl  
sha512sum $(which kubectl) > /opt/course/17/kubectl-local.sha512  
  
# Compare with official  
cat kubectl.sha512  
cat /opt/course/17/kubectl-local.sha512
```

Step 4: Verify suspicious binary

```
# Calculate checksum of suspicious binary  
sha512sum /tmp/kubelet-suspicious > /opt/course/17/kubelet-suspicious.sha512  
  
# Download official kubelet checksum  
curl -LO  
"https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubelet.sha512"  
  
# Compare  
cat kubelet.sha512  
cat /opt/course/17/kubelet-suspicious.sha512
```

Step 5: Document conclusion

```
# If checksums match
echo "GENUINE" > /opt/course/17/conclusion.txt

# If checksums don't match
echo "TAMPERED" > /opt/course/17/conclusion.txt
```

Quick Verification Commands

```
# SHA512 checksum
sha512sum /usr/bin/kubectl

# SHA256 checksum
sha256sum /usr/bin/kubectl

# Verify in one command
echo "$(cat kubectl.sha512) /usr/bin/kubectl" | sha512sum -c
# kubectl: OK
```

17. Node Metadata Protection

Theory

Cloud providers expose instance metadata at **169.254.169.254**. This endpoint can leak sensitive information:

- Instance credentials (AWS IAM, GCP service accounts)
- Instance identity tokens
- User data scripts (may contain secrets)

Protection: Block pod access to metadata endpoint using NetworkPolicy.

Steps: Block Metadata Endpoint

Step 1: Create namespace

```
kubectl create namespace protected-ns
```

Step 2: Create NetworkPolicy to block metadata

```
# /opt/course/18/metadata-netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: block-metadata
  namespace: protected-ns
spec:
  podSelector: {}          # Apply to all pods
  policyTypes:
    - Egress
  egress:
    # Allow all egress EXCEPT metadata endpoint
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0
            except:
              - 169.254.169.254/32
    # Allow DNS
    - ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

Step 3: Apply and test

```
kubectl apply -f /opt/course/18/metadata-netpol.yaml

# Create test pod
kubectl run test -n protected-ns --image=busybox --command -- sleep 3600

# Test metadata access (should timeout/fail)
kubectl exec -n protected-ns test -- wget -q0- --timeout=2
http://169.254.169.254/latest/meta-data/
# Connection timed out

# Test other egress (should work)
kubectl exec -n protected-ns test -- wget -q0- --timeout=2
http://google.com
# Works
```

18. Ingress TLS Configuration

Theory

Ingress TLS terminates HTTPS at the ingress controller, encrypting traffic between clients and the cluster.

Requires:

- TLS certificate and private key
 - Kubernetes Secret of type `kubernetes.io/tls`
 - Ingress resource with `tls` configuration
-

Steps: Configure TLS Ingress

Step 1: Generate self-signed certificate

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \  
-keyout tls.key \  
-out tls.crt \  
-subj "/CN=secure.example.com/O=my-org"
```

Step 2: Create TLS Secret

```
kubectl create secret tls web-tls-secret \  
--cert=tls.crt \  
--key=tls.key \  
-n web-ns  
  
# Save command for reference  
echo "kubectl create secret tls web-tls-secret --cert=tls.crt --  
key=tls.key -n web-ns" > /opt/course/19/secret-create.txt
```

Step 3: Create Ingress with TLS

```
# /opt/course/19/ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: secure-ingress
  namespace: web-ns
spec:
  tls:
    - hosts:
        - secure.example.com
      secretName: web-tls-secret
  rules:
    - host: secure.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web-svc
                port:
                  number: 80
```

Step 4: Apply and verify

```
kubectl apply -f /opt/course/19/ingress.yaml

# Verify TLS is configured
kubectl get ingress secure-ingress -n web-ns -o yaml | grep -A5 tls

# Test TLS (if ingress controller has external IP)
curl -k https://secure.example.com
```

TLS Secret YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: web-tls-secret
  namespace: web-ns
type: kubernetes.io/tls
data:
  tls.crt: <base64-encoded-cert>
  tls.key: <base64-encoded-key>
```

Quick Reference Tables

API Groups

Group	Resources
"" (core)	pods, services, secrets, configmaps, namespaces, nodes, persistentvolumeclaims
apps	deployments, daemonsets, replicaset, statefulsets
networking.k8s.io	networkpolicies, ingresses
rbac.authorization.k8s.io	roles, rolebindings, clusterroles, clusterrolebindings
policy	poddisruptionbudgets
node.k8s.io	runtimeclasses

Important File Paths

File	Purpose
/etc/kubernetes/manifests/kube-apiserver.yaml	API server static pod
/etc/kubernetes/manifests/kube-controller-manager.yaml	Controller manager
/etc/kubernetes/manifests/kube-scheduler.yaml	Scheduler
/etc/kubernetes/manifests/etcd.yaml	etcd
/var/lib/kubelet/config.yaml	Kubelet configuration
/etc/kubernetes/pki/	Cluster PKI certificates
/var/lib/kubelet/seccomp/	Seccomp profiles
/etc/apparmor.d/	AppArmor profiles
/etc/falco/	Falco configuration
/etc/falco/rules.d/	Custom Falco rules

Essential Commands

```
# RBAC Testing
kubectl auth can-i create pods --as=system:serviceaccount:ns:sa
kubectl auth can-i --list --as=user

# Get resources in all namespaces
kubectl get pods -A
kubectl get secrets -A
kubectl get networkpolicies -A

# Debug pods
kubectl describe pod <pod> -n <ns>
kubectl logs <pod> -n <ns>
kubectl exec -it <pod> -n <ns> -- /bin/sh

# Check API server
kubectl get pods -n kube-system | grep api
crictl ps | grep kube-apiserver

# etcd operations
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  get /registry/secrets/<namespace>/<secret-name>

# Falco logs
journalctl -u falco
tail -f /var/log/falco.log

# AppArmor
aa-status
apparmor_parser -r /etc/apparmor.d/<profile>

# Container inspection
crictl ps
crictl inspect <container-id>
```

Exam Day Checklist

1. **Set aliases first** - `alias k=kubectl`, enable completion
 2. **Read questions carefully** - note namespace, resource names, output paths
 3. **Use imperative commands** when possible - faster than YAML
 4. **Verify after each step** - don't assume it worked
 5. **Flag and skip** hard questions - come back later
 6. **Check output paths** - exact paths matter for scoring
 7. **Watch for restarts** - API server changes need restart time
-

Domain Weights

Domain	Weight	Focus Areas
Cluster Setup	15%	NetworkPolicy, CIS Benchmark, TLS
Cluster Hardening	15%	RBAC, ServiceAccount, Metadata Protection
System Hardening	10%	AppArmor, Seccomp, RuntimeClass
Microservice Vulnerabilities	20%	PSA, Secrets Encryption, SecurityContext
Supply Chain Security	20%	Trivy, Kubesec, ImagePolicyWebhook
Monitoring & Runtime	20%	Falco, Audit Logs

Good luck on your CKS exam!