CKS Exam Quick Steps Reference

> **Exam**: 2 hours | 15-20 Questions | 67% Pass | Hands-on
> **This is your visual step-by-step cheat sheet**

CKS Domain Weights

Domain	Weight	Key Topics
1. Cluster Setup	15%	NetworkPolicy, CIS/kube-bench, Ingress TLS, Metadata Protection
2. Cluster Hardening	15%	RBAC, ServiceAccount, K8s Upgrade
3. System Hardening	10%	AppArmor, Seccomp, OS Hardening
4. Minimize Microservice Vulns	20%	PSA, SecurityContext, Secrets Encryption, RuntimeClass, Gatekeeper
5. Supply Chain Security	20%	Trivy, Kubesec, KubeLinter, SBOM, ImagePolicyWebhook
6. Monitoring & Runtime	20%	Falco, Audit Logs, Container Immutability

First Things First - Set Aliases!

```
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 - Default Deny

NetworkPolicies control pod-to-pod traffic. A default deny policy blocks ALL ingress and egress traffic unless explicitly allowed. Essential first step for zero-trust networking.

> **Step 1:** Create namespace

```
kubectl create ns <ns>
```

Step 2: Create NetworkPolicy YAML

- podSelector: {} (ALL pods)
- policyTypes: [Ingress, Egress]
- NO rules = DENY ALL

Step 3: Apply and verify

```
kubectl apply -f <file>
kubectl get netpol -n <ns>
```

Key YAML:

```
spec:
  podSelector: {}
  policyTypes: [Ingress, Egress]
  # No rules = deny all
```

---

2. NetworkPolicy - Allow Specific

After default deny, create allow policies for legitimate traffic. Use pod labels to select sources and destinations. Always include DNS egress (port 53) or pods can't resolve services.

Step 1: Identify source/dest pods (labels)

Step 2: Create policy with:

- podSelector: target pods
- ingress.from: source pods
- egress.to: dest pods
- ALWAYS add DNS (port 53 UDP/TCP)

Step 3: Apply and test

```
kubectl apply -f <file>
kubectl exec <pod> -- wget ...
```

DNS Egress (always add):

```
egress:
  - ports:
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53
```

---

3. CIS Benchmark / kube-bench

CIS benchmarks define security best practices for Kubernetes. kube-bench scans your cluster against these standards and reports failures. Fix findings in API server and kubelet configs.

Step 1: SSH and run kube-bench

```
ssh controlplane
kube-bench run --targets=master
```

Step 2: Fix API server

```
vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

- `--anonymous-auth=false`
- `--profiling=false`
- `--authorization-mode=Node,RBAC`

Step 3: Fix kubelet (on nodes)

```
vim /var/lib/kubelet/config.yaml
```

```
authentication:
  anonymous:
    enabled: false
authorization:
  mode: Webhook
readOnlyPort: 0
```

Step 4: Restart and verify

```
sudo systemctl restart kubelet
kube-bench run --targets=master
```

---

4. RBAC - Role & RoleBinding

RBAC controls who can do what in the cluster. Roles define permissions (verbs on resources), RoleBindings attach roles to users/ServiceAccounts. Namespace-scoped.

Step 1: Create ServiceAccount

```
kubectl create sa <sa> -n <ns>
```

Step 2: Create Role (namespace-scoped)

```
kubectl create role <role> \
  --verb=get,list,create \
  --resource=pods,deployments \
  -n <ns>
```

Step 3: Create RoleBinding

```
kubectl create rolebinding <rb> \
  --role=<role> \
  --serviceaccount=<ns>:<sa> \
  -n <ns>
```

Step 4: Test

```
kubectl auth can-i create pods \
  --as=system:serviceaccount:<ns>:<sa> -n <ns>
```

API Groups:

Group	Resources
""	pods, services, secrets, configmaps
apps	deployments, daemonsets, statefulsets
networking.k8s.io	networkpolicies, ingresses

5. RBAC - ClusterRole (cluster-wide)

ClusterRoles grant permissions across ALL namespaces or on cluster-scoped resources (nodes, PVs). Use ClusterRoleBinding to assign. Be careful - these are powerful.

Step 1: Create ClusterRole

```
kubectl create clusterrole <cr> \
  --verb=get,list,watch \
  --resource=nodes,pods
```

Step 2: Create ClusterRoleBinding

```
kubectl create clusterrolebinding <crb> \
  --clusterrole=<cr> \
  --serviceaccount=<ns>:<sa>
```

Step 3: Test

```
kubectl auth can-i list nodes \
  --as=system:serviceaccount:<ns>:<sa>
```

6. ServiceAccount Security

ServiceAccounts provide pod identity. By default, tokens are auto-mounted giving pods API access. Disable auto-mount and use minimal RBAC for least privilege.

Step 1: Create SA with no auto-mount

```
automountServiceAccountToken: false
```

Step 2: Update Pod/Deployment spec

- serviceAccountName: `<sa>`
- automountServiceAccountToken: `false`

Step 3: Create minimal Role (least priv)

- NO secrets unless required

Step 4: Verify no token

```
kubectl exec <pod> -- ls /var/run/secrets/kubernetes.io/
# Should fail (no token mounted)
```

---

7. AppArmor Profiles

AppArmor is a Linux kernel security module that restricts program capabilities. Profiles must be loaded on each node before pods can use them. Kubernetes 1.30+ uses native field.

Step 1: SSH to node

```
ssh <node>
```

Step 2: Check profile loaded

```
sudo aa-status | grep <profile>
```

Step 3: Load if needed

```
sudo apparmor_parser -r /etc/apparmor.d/<profile>
```

Step 4: Add to Pod spec

```
containers:
- securityContext:
    appArmorProfile:
      type: Localhost
      localhostProfile: <profile>
```

Step 5: Apply and verify

```
kubectl apply -f <file>
```

Profile Types: `RuntimeDefault` | `Localhost` | `Unconfined`

---

8. Seccomp Profiles

Seccomp filters system calls a container can make. RuntimeDefault blocks dangerous syscalls. Custom profiles go in `/var/lib/kubelet/seccomp/`. Required for PSA restricted.

Option 1: RuntimeDefault (easiest)

```
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
```

Option 2: Custom Localhost profile

- Profile at: `/var/lib/kubelet/seccomp/<file>`

```
seccompProfile:
  type: Localhost
  localhostProfile: <file>.json
```

Step 3: Apply and verify running

```
kubectl apply -f <file>
```

---

9. Pod Security Admission (PSA)

PSA enforces security standards at namespace level via labels. Restricted level requires non-root, seccomp, dropped capabilities, and no privilege escalation. Replaces PodSecurityPolicy.

Step 1: Label namespace

```
kubectl label ns <ns> \
  pod-security.kubernetes.io/enforce=restricted
```

Step 2: Restricted Pod MUST have:

- ☑ `runAsNonRoot: true`
- ☑ `seccompProfile: RuntimeDefault`
- ☑ `allowPrivilegeEscalation: false`
- ☑ `capabilities.drop: ["ALL"]`
- ☑ No hostPath, hostNetwork, hostPID
- ☑ No privileged containers

Step 3: Best practices (add for nginx etc)

- `readOnlyRootFilesystem: true`
- emptyDir for `/tmp`, `/var/cache`, `/var/run`

Step 4: Test - run non-compliant pod

- Should be rejected

Levels: `privileged` | `baseline` | `restricted`
Modes: `enforce` | `warn` | `audit`

---

10. Secrets Encryption at Rest

By default, Secrets are stored base64-encoded (not encrypted) in etcd. Enable encryption at rest using aescbc provider. Encryption provider must come BEFORE identity in config.

Step 1: Generate key

```
head -c 32 /dev/urandom | base64
```

Step 2: Create encryption config

- Path: `/etc/kubernetes/encryption-config.yaml`
- **!! aescbc FIRST, identity LAST !!**

Step 3: Edit kube-apiserver.yaml

- `--encryption-provider-config=/etc/kubernetes/encryption-config.yaml`
- Add volumeMounts + volumes

Step 4: Wait for API restart

```
watch "crictl ps | grep apiserver"
```

Step 5: Re-encrypt existing secrets

```
kubectl get secrets -A -o json | kubectl replace -f -
```

Step 6: Verify in etcd (encrypted)

```
etcdctl get /registry/secrets/...
# Should start with k8s:enc:aescbc
```

---

11. SecurityContext Hardening

SecurityContext sets security settings at pod/container level. Key settings: runAsNonRoot, readOnlyRootFilesystem, drop ALL capabilities, and disable privilege escalation.

Add to Pod/Container spec:

```
spec:
  securityContext:           # Pod-level
    runAsNonRoot: true
    runAsUser: 1000
    fsGroup: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
  - securityContext:         # Container
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop: ["ALL"]
```

Add emptyDir for writable paths

---

12. Trivy Image Scanning

Trivy scans container images for known vulnerabilities (CVEs). Focus on HIGH and CRITICAL severity. Compare images to choose the one with fewer vulnerabilities.

Step 1: Scan for HIGH/CRITICAL

```
trivy image --severity HIGH,CRITICAL <image>:<tag>
```

Step 2: Compare images

```
trivy image nginx:1.19 > old.txt
trivy image nginx:alpine > new.txt
```

Step 3: Choose image with fewer vulns

Step 4: Update deployment

```
kubectl set image deploy/<name> <container>=<safer-image>
```

Quick flags: `--severity` | `-q` (quiet) | `--ignore-unfixed`

---

13. Kubesec Analysis

Kubesec analyzes Kubernetes manifests and assigns a security score. Higher score = more secure. Add security features like runAsNonRoot and resource limits to improve score.

Step 1: Scan manifest

```
kubesec scan <file>.yaml
```

Step 2: Check score (target: 8+)

Step 3: Add security features

- +1 `runAsNonRoot: true`
- +1 `readOnlyRootFilesystem: true`
- +1 `capabilities.drop: ALL`
- +1 `resources.limits`
- +1 `automountServiceAccountToken: false`

Step 4: Rescan and verify score >= 8

---

14. Falco Runtime Security

Falco detects abnormal behavior at runtime using kernel-level monitoring. Custom rules go in `/etc/falco/rules.d/`. Alerts on shell spawns, file access, network connections, etc.

Step 1: SSH to node where Falco runs

```
ssh <node>
```

Step 2: Create rule file

- Path: `/etc/falco/rules.d/<name>.yaml`

Step 3: Rule structure

```
- rule: <name>
  desc: <description>
  condition: <expression>
  output: <message with %fields>
  priority: WARNING|ALERT|etc
```

Step 4: Restart Falco

```
sudo systemctl restart falco-modern-bpf
```

Step 5: Trigger & check logs

```
kubectl exec <pod> -- /bin/sh
journalctl -u falco-modern-bpf -f
```

Common macros:

```
- macro: spawned_process
  condition: evt.type in (execve, execveat)
- macro: container
  condition: container.id != host
```

Output fields: `%proc.name` | `%container.name` | `%k8s.pod.name` | `%user.name`

---

15. Audit Logs

Audit logs record all API requests for security analysis and compliance. Policy defines what to log (resources, verbs) and at what level (Metadata, Request, RequestResponse).

> **Step 1:** Create audit policy
>
> - Path: `/etc/kubernetes/audit-policy.yaml`
> - level: `None` | `Metadata` | `Request` | `RequestResponse`
> - resources: [secrets, pods, etc]
> - verbs: [create, delete, etc]

> **Step 2:** Edit kube-apiserver.yaml
>
> - `--audit-policy-file=<path>`
> - `--audit-log-path=<log-path>`
> - `--audit-log-maxage=30`
> - Add volumeMounts + volumes

> **Step 3:** Create log directory

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

> **Step 4:** Wait for API restart

> **Step 5:** Test & find entry

```
kubectl create secret ...
grep <secret> <audit-log>
```

Audit Levels: `None` -> `Metadata` -> `Request` -> `RequestResponse`

---

16. RuntimeClass / gVisor

RuntimeClass selects different container runtimes. gVisor provides kernel-level isolation by intercepting syscalls. Adds security layer between container and host kernel.

> **Step 1:** Verify RuntimeClass exists

```
kubectl get runtimeclass gvisor
```

Step 2: Add to Pod spec

```
spec:
  runtimeClassName: gvisor
```

Step 3: Apply and verify running

```
kubectl apply -f <file>
```

Step 4: Verify gVisor

```
kubectl exec <pod> -- dmesg | head
# Should show gVisor kernel
```

---

17. ImagePolicyWebhook

ImagePolicyWebhook is an admission controller that validates images before pod creation. External webhook decides allow/deny. Set defaultAllow=false to deny if webhook is down.

Step 1: Create webhook kubeconfig

- Path: `/etc/kubernetes/admission/image-policy-kubeconfig.yaml`

Step 2: Create admission config

- Path: `/etc/kubernetes/admission/admission-config.yaml`
- `defaultAllow: false` (DENY if webhook down)

Step 3: Edit kube-apiserver.yaml

- `--enable-admission-plugins=NodeRestriction,ImagePolicyWebhook`
- `--admission-control-config-file=<admission-config-path>`
- Add volumeMounts + volumes

Step 4: Wait & test allowed/denied images

---

18. Binary Verification

Verify Kubernetes binaries haven't been tampered with by comparing SHA checksums against official releases. Mismatch indicates compromised binary - potential supply chain attack.

Step 1: Get cluster version

```
kubectl version
```

Step 2: Download official checksum

```
curl -LO
https://dl.k8s.io/release/<version>/bin/linux/amd64/kubectl.sha512
```

Step 3: Calculate local checksum

```
sha512sum $(which kubectl)
```

Step 4: Compare

- MATCH -> GENUINE
- NO MATCH -> TAMPERED

Step 5: Save conclusion to file

---

19. Node Metadata Protection

Cloud metadata service (169.254.169.254) exposes sensitive info like IAM credentials. Block pod access using NetworkPolicy egress rules to prevent SSRF attacks.

Step 1: Create NetworkPolicy to block `169.254.169.254/32`

Step 2: Policy structure

```
spec:
  podSelector: {}
  policyTypes: [Egress]
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
        - 169.254.169.254/32
  - ports: [UDP/TCP 53]  # DNS
```

Step 3: Test metadata access - should fail

```
wget http://169.254.169.254/...
```

---

20. Ingress TLS

TLS encrypts traffic between clients and the Ingress controller. Store certificate and key in a TLS Secret, reference it in Ingress spec. HTTPS prevents traffic sniffing.

Step 1: Generate cert

```
openssl req -x509 -nodes -days 365 \
  -newkey rsa:2048 \
  -keyout tls.key -out tls.crt \
  -subj "/CN=<domain>"
```

Step 2: Create TLS secret

```
kubectl create secret tls <name> \
  --cert=tls.crt --key=tls.key \
  -n <ns>
```

Step 3: Create Ingress with TLS

```
spec:
  tls:
  - hosts: [<domain>]
    secretName: <tls-secret>
  rules: ...
```

Step 4: Apply and verify

```
kubectl apply -f <file>
```

---

21. OPA Gatekeeper (Policy Enforcement)

Gatekeeper uses OPA (Open Policy Agent) to enforce custom policies via admission control. ConstraintTemplates define policy logic in Rego, Constraints apply them to resources.

Step 1: Verify Gatekeeper installed

```
kubectl get pods -n gatekeeper-system
```

Step 2: Create ConstraintTemplate (policy)

```
apiVersion: templates.gatekeeper.sh
kind: ConstraintTemplate
spec.targets[].rego: <policy-logic>
```

Step 3: Create Constraint (apply policy)

```
apiVersion: constraints.gatekeeper.sh
kind: <TemplateName>
spec.match.kinds: [Pod, Deployment]
spec.parameters: <values>
```

Step 4: Apply Template FIRST, then Constraint

Step 5: Test - create violating resource

- Should be rejected

Common Use Cases:

- Restrict allowed image registries
- Require resource limits on pods
- Enforce required labels

---

22. SBOM (Software Bill of Materials)

SBOM lists all components/dependencies in a container image. Enables vulnerability tracking and license compliance. Generate with Trivy in CycloneDX or SPDX format.

Step 1: Generate SBOM with Trivy

```
trivy image --format cyclonedx -o sbom.json <image>
```

Step 2: Or generate SPDX format

```
trivy image --format spdx-json -o sbom.spdx.json <image>
```

Step 3: Generate with bom tool

```
bom generate --image <image> --format spdx -o sbom.spdx
```

Step 4: Scan existing SBOM for vulns

```
trivy sbom sbom.json
```

Formats: `CycloneDX` (OWASP) | `SPDX` (ISO standard)

---

23. KubeLinter (Static Analysis)

KubeLinter statically analyzes Kubernetes manifests for security misconfigurations and best practices. Catches issues before deployment. Non-zero exit code for CI/CD integration.

Step 1: Scan manifest

```
kube-linter lint <file>.yaml
```

Step 2: Scan directory

```
kube-linter lint ./manifests/
```

Step 3: Scan Helm chart

```
kube-linter lint ./my-chart/
```

Step 4: List available checks

```
kube-linter checks list
```

Step 5: Run specific checks only

```
kube-linter lint --include "run-as-non-root,no-read-only-root-fs"
<file>.yaml
```

Step 6: Fix issues and rescan

Note: Non-zero exit code on findings (CI/CD friendly)

---

24. Kubernetes Version Upgrade

Regular upgrades patch security vulnerabilities. Upgrade one minor version at a time (never skip). Order: kubeadm first, then kubelet/kubectl. Drain nodes before upgrading.

Step 1: Drain control plane node

```
kubectl drain <node> --ignore-daemonsets
```

Step 2: Upgrade kubeadm FIRST

```
apt-get update
apt-get install -y kubeadm=1.XX.0-*
```

Step 3: Plan and apply upgrade

```
kubeadm upgrade plan
kubeadm upgrade apply v1.XX.0
```

Step 4: Upgrade kubelet & kubectl

```
apt-get install -y kubelet=1.XX.0-* kubectl=1.XX.0-*
```

Step 5: Restart kubelet

```
systemctl daemon-reload
systemctl restart kubelet
```

Step 6: Uncordon node

```
kubectl uncordon <node>
```

Rule: NEVER skip minor versions (1.32->1.33->1.34)

---

25. mTLS / Pod-to-Pod Encryption

mTLS encrypts traffic between pods with mutual authentication. Service meshes like Istio or CNIs like Cilium provide this. STRICT mode enforces mTLS-only communication.

ISTIO mTLS

Step 1: Label ns for sidecar injection

```
kubectl label ns <ns> istio-injection=enabled
```

Step 2: Create PeerAuthentication

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
spec.mtls.mode: STRICT
```

Step 3: Verify

```
istioctl x describe pod <pod>
```

CILIUM WireGuard

Step 1: Enable during install

```
helm install cilium --set \
  encryption.enabled=true \
  encryption.type=wireguard
```

> **Step 2:** Verify
>
> ```
> cilium encrypt status
> ```

Modes: `STRICT` (mTLS only) | `PERMISSIVE` (both)

Critical File Paths

Path	Purpose
`/etc/kubernetes/manifests/`	Static pod manifests (API, etcd, etc)
`/var/lib/kubelet/config.yaml`	Kubelet configuration
`/var/lib/kubelet/seccomp/`	Seccomp profiles
`/etc/apparmor.d/`	AppArmor profiles
`/etc/falco/rules.d/`	Custom Falco rules
`/etc/kubernetes/pki/`	Cluster certificates
`/etc/kubernetes/audit/`	Audit policy location
`/var/log/kubernetes/audit/`	Audit log files

Quick Commands Cheat Sheet

```
# RBAC testing
kubectl auth can-i <verb> <resource> --as=system:serviceaccount:<ns>:<sa>
-n <ns>
kubectl auth can-i --list --as=<user> -n <ns>

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

# Watch API server restart
watch "crictl ps | grep kube-apiserver"

# etcd access
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 <key>

# Falco logs
journalctl -u falco-modern-bpf -f

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

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

Common Mistakes to AVOID

Mistake	Fix
Forgot `-n <namespace>`	ALWAYS specify namespace
Didn't wait for API restart	`watch crictl ps | grep api`
Wrong output file path	Double-check question paths
Missing DNS in NetworkPolicy	Add port 53 UDP/TCP egress
Missing seccomp for PSA	Add `seccompProfile.type: RuntimeDefault`
Missing `capabilities.drop: ALL`	Required for PSA restricted
Put `identity: {}` first in encryption	Encryption provider MUST be first
Forgot to re-encrypt secrets	`kubectl get secrets -A -o json | kubectl replace -f -`

Exam Day Flow

```
1. Set aliases FIRST
2. Read question FULLY (note ns, paths, names)
3. Use imperative commands when possible
4. VERIFY after each step
5. Flag hard questions -> skip -> return later
6. Check output paths match exactly
7. Watch for restart requirements
```

Good luck on your CKS exam!