Understand Application Security

Elle Krout
Principal Course Author, Pluralsight

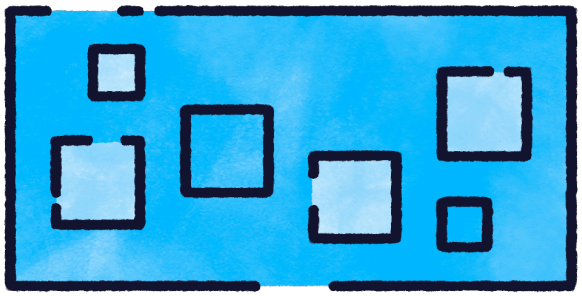
Kubernetes Security Contexts

Security contexts define privilege and access control settings for a pod or container

Security Context Privilege and Access Control Settings

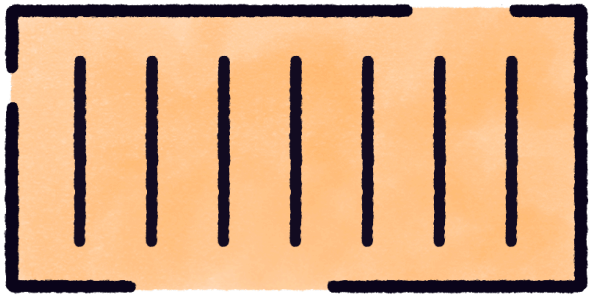
- UID/GID settings
- Privilege escalation configuration
- Filesystem settings
- AppArmor/SELinux settings
- Seccomp settings
- Linux capabilities options

Security Contexts Can Be Set at Two Levels



Pod

Settings apply to all containers in the pod; ideal for settings that needs to be consistent

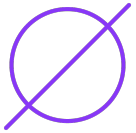


Container

Settings apply to only a single container; ideal for individualized settings

Use Cases

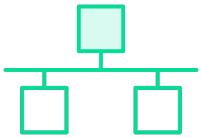
Run as non-root to prevent privilege escalation



Reduce access to unneeded kernel calls with seccomp

Reduce potential issues by providing read-only access

Limit pod communication with SELinux



Use Linux capabilities to bing containers to a port

... and more!

Security Contexts

Allow you to reduce privileges, enforce consistency, and apply kernel-level protections to your pods and containers

Using Kubernetes Security Contexts

Assign Security Context to Container

secure-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "sleep 3600”]
    securityContext:
      allowPrivilegeEscalation: false
      runAsGroup: 3000
      runAsUser: 1000
```

Assign Security Context to Pods

secure-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsGroup: 3000
    runAsUser: 1000
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "sleep 3600”]
```

Security Context Parameters

Scope	Matches
`runAsUser`	Define user ID for container process
`runAsGroup`	Define group ID for container process
`readOnlyRootFilesystem`	Force container to have a read-only filesystem
`procMount`	Define proc mount for container
`allowPrivilegeEscalation`	Set whether container process can gain more privilege than its parent process
`privileged`	Boolean to set container to what is essentially root on the host

Security Context Parameters Continued

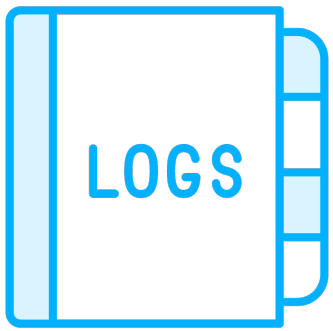
Scope	Matches
`runAsNonRoot`	Boolean indicating a container should run as a non-root user
`appArmorProfile`	Define AppArmor profile and settings
`capabilities`	Add or drop Linux capabilities
`seLinuxOptions`	Define SELinux context
`seccompProfile`	Define seccomp profile and settings
`windowsOptions`	Security options for Windows containers

View Security Contexts

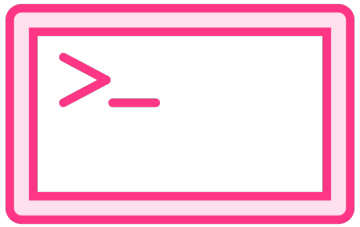
```
> kubectl get pod <pod-name> -o yaml
```

```
...
spec:
  containers:
  - command:
    - sh
    - -c
    - sleep 3600
    image: busybox
    imagePullPolicy: Always
    name: demo
    resources: {}
    securityContext:
      allowPrivilegeEscalation: false
      runAsGroup: 3000
      runAsUser: 1000
```

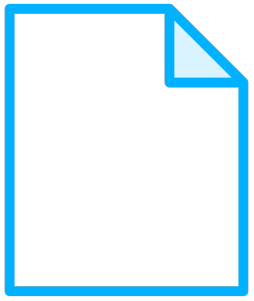
Troubleshoot Security Contexts



Check events and logs

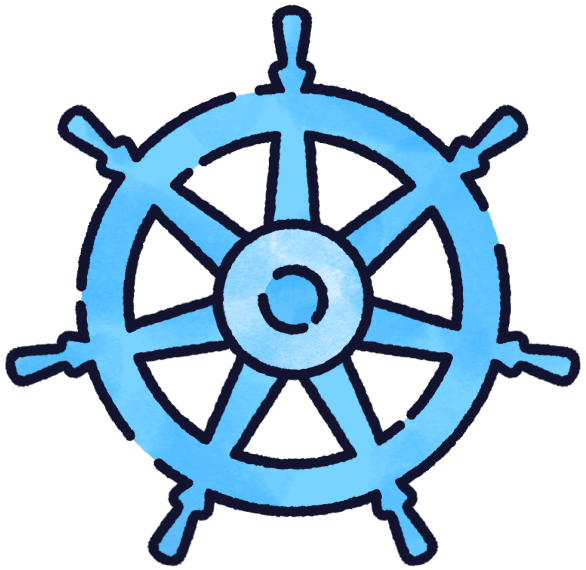


Exec into container



Validate the filesystem

Verify Linux capabilities



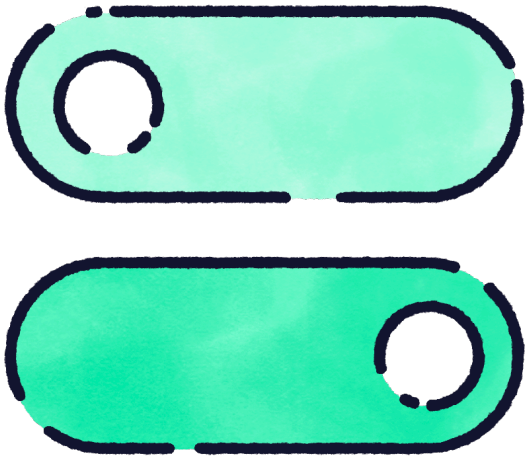
Security Contexts...

Are an essential security tool for managing your pod and container privileges and access

Kubernetes Capabilities

Linux capabilities are enhanced features available on all Linux-based hosts

Linux Capabilities



A way of breaking down root privileges into smaller permissions

Containers do not start with full privileges by default

Settings determined by runtime

View Capability Options

```
> man capabilities

CAPABILITIES(7)          Linux Programmer's Manual          CAPABILITIES(7)

NAME
       capabilities - overview of Linux capabilities

DESCRIPTION
       For  the  purpose of performing permission checks, traditional UNIX
       implementations distinguish two categories of processes: privileged
       processes  (whose  effective user ID is 0, referred to as superuser
       or root), and unprivileged processes (whose effective UID  is  non-
       zero).   Privileged  processes bypass all kernel permission checks,
       while unprivileged processes are subject to full permission  check-
       ing based on the process's credentials (usually: effective UID, ef-
       fective GID, and supplementary group list).
```

Add or Drop Capabilities

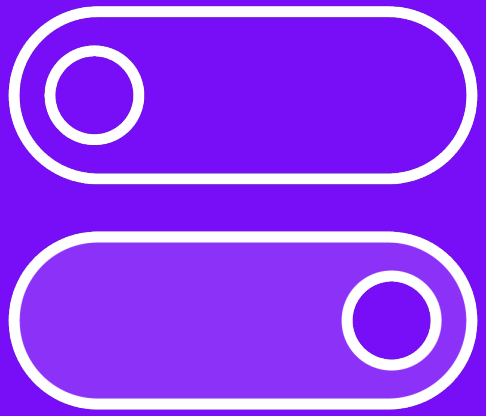
secure-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "sleep 3600”]
    securityContext:
      capabilities:
        drop: ["ALL"]
        add: ["NET_BIND_SERVICE"]
```

Verify Capabilities are Set

```
> kubectl exec -it <pod> -- sh
> cat /proc/1/status | grep Cap

CapInh:  0000000000000000
CapPrm:  00000000a80425fb
CapEff:  00000000a80425fb
CapBnd:  00000000a80425fb
CapAmb:  0000000000000000
```



Linux Capabilities

Are a powerful way to fine-tune your container privileges beyond simply letting them run as root

Demo: Applying Secure Setting in Kubernetes

Exam Scenario

Apply sensible pod-level security default to the `monitoring-pod`, ensuring it does not allow privilege escalation, and uses the `RunTimeDefault` seccomp profile

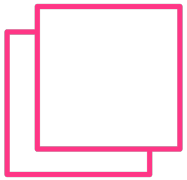
The two containers in the pod, `sidecar` and `logger` also need additional security settings. `sidecar` should run as UID 1000 and drop `NET_RAW` capabilities. `logger` should to set to run as both user and group 2000. Both containers should disable privilege escalation

Kubernetes Security Admission

The PodSecurityAdmission controller enables security standards on all pods

PodSecurityAdmission Permission Levels

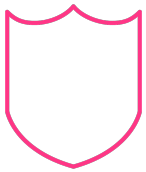
Privileged: Most permissive; allows workloads broad access to host



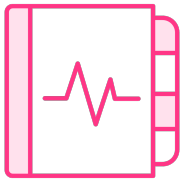
Baseline: Minimally restrictive; blocks know privilege escalations, allows for common container patterns

Restricted: Most restrictive; enforces current best practice security suggestions

PodSecurityAdmission Permission Modes



Enforce: Enforces set level; rejects pods in violation



Audit: Records issues in audit log, but lets pods deploy

Warn: Outputs violations to screen but lets pods deploy

Apply Level and Mode to Namespace

```
> kubectl label namespace test pod-security.kubernetes.io/warn=baseline


> kubectl apply -f violation-pod.yaml

Warning: would violate PodSecurity "baseline:latest": host namespaces
(hostNetwork=true)
```

View PodSecurityAdmission Labels for Namespace

```
> kubectl get ns <namespace> --show-labels

NAME   STATUS   AGE   LABELS
test   Active   46h   kubernetes.io/metadata.name=test,pod-security.kubernetes.io/
warn=baseline
```

Demo: Working with the Pod Security Admission

Exam Scenario

Create a namespace, `production-test`. This namespace should have the highest level of restriction for it's pod security, fully enforced. That said, before this is fully enabled, you should test the `bad-pod.yaml` pod. Have any issues output to the screen after you deployment, as well as tracked in the audit logs

Remove the pod, then apply full enforcement to the namespace. Attempt to deploy the pod again