# Configuring Pod Security Contexts

37m 51s left

**Open Cloud Environment**
100% Setup completed
Average setup time: 2m 28s

**Open Code Environment**
100% Setup completed

## Credentials

Account ID
63274379 Copy

Username
student Copy

Password
Ca1_eIUh Copy

Region
US West 2 Copy

PEM Download
PPK Download

Bastion Public Ip
54.245.18 Copy

Cluster SSH
ssh ubunt

## Introduction

A security context allows you to set access control for Pods, as well as containers and volumes in Pods, when applicable. Examples of access controls that can be set with security contexts include:

- The user ID and group IDs of the first process running in a container
- The group ID of volumes
- If a container's root file system is read-only
- Security-Enhanced Linux (SELinux) options
- The privileged status of containers, which allows the container to do almost everything root can do on the host if enabled
- Whether or not privilege escalation, where child processes can have more privileges than their parent, is allowed

When a security context field is configured for a Pod and one of the Pod's containers, the container's setting takes precedence. Configuring the security context of Pods and containers can greatly reduce the security risk posed by using third-party images.

In this lab step, you will review the available options for configuring security contexts. You will then create multiple Pods with differing security contexts to observe the effects of each.

## Instructions

1. Explain the available Pod-level security context fields:

Copy code

```
kubectl explain pod.spec.securityContext | more
```

## Lab Steps

1. Connecting to the Kubernetes Cluster
2. **Configuring Pod Security Contexts**

Need help? Contact our support team

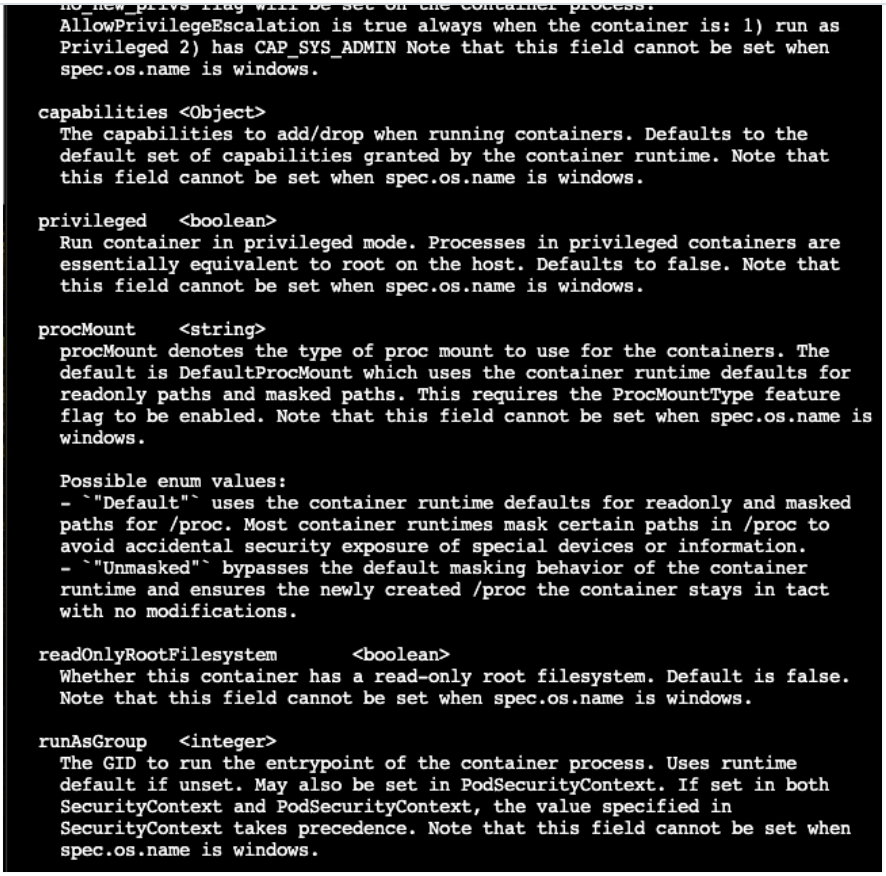
```
owned by the pod:

  1. The owning GID will be the FSGroup 2. The setgid bit is set (new files
  created in the volume will be owned by FSGroup) 3. The permission bits are
  OR'd with rw-rw----

  If unset, the Kubelet will not modify the ownership and permissions of any
  volume. Note that this field cannot be set when spec.os.name is windows.

fsGroupChangePolicy  <string>
  fsGroupChangePolicy defines behavior of changing ownership and permission
  of the volume before being exposed inside Pod. This field will only apply
  to volume types which support fsGroup based ownership(and permissions). It
  will have no effect on ephemeral volume types such as: secret, configmaps
  and emptydir. Valid values are "OnRootMismatch" and "Always". If not
  specified, "Always" is used. Note that this field cannot be set when
  spec.os.name is windows.

  Possible enum values:
  - `"Always"` indicates that volume's ownership and permissions should
  always be changed whenever volume is mounted inside a Pod. This the default
  behavior.
  - `"OnRootMismatch"` indicates that volume's ownership and permissions will
  be changed only when permission and ownership of root directory does not
  match with expected permissions on the volume. This can help shorten the
  time it takes to change ownership and permissions of a volume.

runAsGroup   <integer>
  The GID to run the entrypoint of the container process. Uses runtime
  default if unset. May also be set in SecurityContext. If set in both
  SecurityContext and PodSecurityContext, the value specified in
  SecurityContext takes precedence for that container. Note that this field
  cannot be set when spec.os.name is windows.
```

Briefly read through each of the fields to get an understanding of what each can be used for.

2. Explain the available container-level security context fields:

Copy code

```
kubectl explain pod.spec.containers.securityContext | more
```




Briefly read through each of the fields to get an understanding of what each can be used for.

3. Create the following Pod manifest file:

Copy code

```
cat << EOF > pod-no-security-context.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-test-1
spec:
  containers:
  - image: busybox:1.30.1
    name: busybox
    args:
    - sleep
    - "3600"
EOF
```

The pod simply runs a container that sleeps.

4. Create the Pod and use exec to list the available devices in the container:



*seconds before running the command again*

```
ubuntu@ip-10-0-128-5:~$
core
fd
full
mqueue
null
ptmx
pts
random
shm
stderr
stdin
stdout
termination-log
tty
urandom
zero
```

There are only a minimal number of devices available in the container and none that can do any harm. For the sake of what you will do next, notice there are no block devices. In particular, there is no nvme0n1p1 device that is the host's file system disk.

5. Delete the previous Pod and create a similar Pod that has a privileged container:

Copy code

```
kubectl delete -f pod-no-security-context.yaml
cat > pod-privileged.yaml <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: security-context-test-2
spec:
  containers:
  - image: busybox:1.30.1
    name: busybox
    args:
    - sleep
    - "3600"
    securityContext:
      privileged: true
EOF
kubectl create -f pod-privileged.yaml
```

Note the `securityContext` field included in the `spec`.

6. List the devices available in the container:

Copy code

```
kubectl exec security-context-test-2 -- ls /dev
```

```
cpu_dma_latency
cuse
dma_heap
ecryptfs
fd
full
fuse
hpet
hwrng
input
kmsg
loop-control
loop0
loop1
loop2
loop3
loop4
loop5
loop6
loop7
mapper
mcelog
```

All of the host devices are available including the host file system disk **nvme0n1p1**. This could be a major security breach and shows the importance of carefully considering if you should ever use a privileged container.

8. Create another pod that includes a Pod security context as well as a container security context:

Copy code

```
kubectl delete -f pod-privileged.yaml
cat << EOF > pod-runas.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-test-3
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
  containers:
  - image: busybox:1.30.1
    name: busybox
    args:
    - sleep
    - "3600"
    securityContext:
      runAsUser: 2000
      readOnlyRootFilesystem: true
EOF
kubectl create -f pod-runas.yaml
```

The Pod security context enforces that container processes do not run as root (`runAsNonRoot`) and sets the user ID of the container process to `1000`. The container `securityContext` sets the container process' user ID to `2000` and sets the root file system to read-only.

```
kubectl exec security-context-test-3 -it -- /bin/sh
```

```
/ $
```

Notice that the shell prompt is **$** and not # indicating that you are not the root user.

10. List the running processes in the container:

Copy code

```
ps
```

```
PID   USER     TIME  COMMAND
    1 2000      0:00 sleep 3600
    5 2000      0:00 /bin/sh
   26 2000      0:00 ps
```

The **USER** ID is 2000 illustrating that it is not root and that the container security context overrides the setting in the Pod security context when both security contexts include the same field. Whenever possible you should not run as root.

11. Attempt to create a file in the /tmp directory:

Copy code

```
touch /tmp/test-file
exit
```

```
touch: /tmp/test-file: Read-only file system
```

The attempt fails due to the **Read-only file system**. When possible, it is best to use read-only root file systems to harden your container environments. A best practice is to use volumes to mount any files that require modification, allowing the root file system to be read-only.

12. Delete the pod resource:

Copy code

In this lab step, you understood the use and capabilities of Pod and container security contexts. You also learned about potential risks and how to protect your cluster from them.

Did you like this step?

End Lab Back Submit