Configure a Security Context for a Pod or Container

Archived: 2026-04-05 17:45:59 UTC

A security context defines privilege and access control settings for a Pod or Container. Security context settings include, but are not limited to:

- Discretionary Access Control: Permission to access an object, like a file, is based on user ID (UID) and group ID (GID).
- Security Enhanced Linux (SELinux): Objects are assigned security labels.
- Running as privileged or unprivileged.
- Linux Capabilities: Give a process some privileges, but not all the privileges of the root user.
- AppArmor: Use program profiles to restrict the capabilities of individual programs.
- Seccomp: Filter a process's system calls.
- `allowPrivilegeEscalation`: Controls whether a process can gain more privileges than its parent process. This bool directly controls whether the `no_new_privs` flag gets set on the container process. `allowPrivilegeEscalation` is always true when the container:
  - is run as privileged, or
  - has `CAP_SYS_ADMIN`
- `readOnlyRootFilesystem`: Mounts the container's root filesystem as read-only.

The above bullets are not a complete set of security context settings -- please see SecurityContext for a comprehensive list.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud

To check the version, enter `kubectl version`.

Set the security context for a Pod

To specify security settings for a Pod, include the `securityContext` field in the Pod specification. The `securityContext` field is a PodSecurityContext object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a `securityContext` and an `emptyDir` volume:

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
    supplementalGroups: [4000]
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox:1.28
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, all processes run with user ID 1000. The `runAsGroup` field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when `runAsGroup` is specified. Since `fsGroup` field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume `/data/demo` and any files created in that volume will be Group ID 2000. Additionally, when the `supplementalGroups` field is specified, all processes of the container are also part of the specified groups. If this field is omitted, it means empty.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

In your shell, list the running processes:

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

```
PID   USER     TIME  COMMAND
   1 1000      0:00 sleep 1h
   6 1000      0:00 sh
...
```

In your shell, navigate to `/data`, and list the one directory:

The output shows that the `/data/demo` directory has group ID 2000, which is the value of `fsGroup`.

```
drwxrwsrwx 2 root 2000 4096 Jun  6 20:08 demo
```

In your shell, navigate to `/data/demo`, and create a file:

```
cd demo
echo hello > testfile
```

List the file in the `/data/demo` directory:

The output shows that `testfile` has group ID 2000, which is the value of `fsGroup`.

```
-rw-r--r-- 1 1000 2000 6 Jun  6 20:08 testfile
```

Run the following command:

The output is similar to this:

```
uid=1000 gid=3000 groups=2000,3000,4000
```

From the output, you can see that `gid` is 3000 which is same as the `runAsGroup` field. If the `runAsGroup` was omitted, the `gid` would remain as 0 (root) and the process will be able to interact with files that are owned by the root(0) group and groups that have the required group permissions for the root (0) group. You can also see that `groups` contains the group IDs which are specified by `fsGroup` and `supplementalGroups`, in addition to `gid`.

Exit your shell:

Implicit group memberships defined in `/etc/group` in the container image

By default, kubernetes merges group information from the Pod with information defined in `/etc/group` in the container image.

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    supplementalGroups: [4000]
  containers:
  - name: sec-ctx-demo
    image: registry.k8s.io/e2e-test-images/agnhost:2.45
    command: [ "sh", "-c", "sleep 1h" ]
    securityContext:
      allowPrivilegeEscalation: false
```

This Pod security context contains `runAsUser`, `runAsGroup` and `supplementalGroups`. However, you can see that the actual supplementary groups attached to the container process will include group IDs which come from `/etc/group` in the container image.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-5.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

Check the process identity:

The output is similar to this:

```
uid=1000 gid=3000 groups=3000,4000,50000
```

You can see that `groups` includes group ID `50000`. This is because the user (`uid=1000`), which is defined in the image, belongs to the group (`gid=50000`), which is defined in `/etc/group` inside the container image.

Check the `/etc/group` in the container image:

You can see that uid `1000` belongs to group `50000`.

```
 ...
user-defined-in-image:x:1000:
group-defined-in-image:x:50000:user-defined-in-image
```

Exit your shell:

Note:

Implicitly merged supplementary groups may cause security problems particularly when accessing the volumes (see kubernetes/kubernetes#112879 for details). If you want to avoid this. Please see the below section.

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default)

This feature can be enabled by setting the `SupplementalGroupsPolicy` feature gate for kubelet and kube-apiserver, and setting the `.spec.securityContext.supplementalGroupsPolicy` field for a pod.

The `supplementalGroupsPolicy` field defines the policy for calculating the supplementary groups for the container processes in a pod. There are two valid values for this field:

- `Merge`: The group membership defined in `/etc/group` for the container's primary user will be merged. This is the default policy if not specified.
- `Strict`: Only group IDs in `fsGroup`, `supplementalGroups`, or `runAsGroup` fields are attached as the supplementary groups of the container processes. This means no group membership from `/etc/group` for the container's primary user will be merged.

When the feature is enabled, it also exposes the process identity attached to the first container process in `.status.containerStatuses[].user.linux` field. It would be useful for detecting if implicit group ID's are attached.

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    supplementalGroups: [4000]
```

```
    supplementalGroupsPolicy: Strict
  containers:
  - name: sec-ctx-demo
    image: registry.k8s.io/e2e-test-images/agnhost:2.45
    command: [ "sh", "-c", "sleep 1h" ]
    securityContext:
      allowPrivilegeEscalation: false
```

This pod manifest defines `supplementalGroupsPolicy=Strict`. You can see that no group memberships defined in `/etc/group` are merged to the supplementary groups for container processes.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-6.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Check the process identity:

```
kubectl exec -it security-context-demo -- id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=3000,4000
```

See the Pod's status:

```
kubectl get pod security-context-demo -o yaml
```

You can see that the `status.containerStatuses[].user.linux` field exposes the process identity attached to the first container process.

```
  ...
status:
  containerStatuses:
  - name: sec-ctx-demo
    user:
      linux:
        gid: 3000
        supplementalGroups:
        - 3000
```

```
        - 4000
        uid: 1000
...
```

Note:

Please note that the values in the `status.containerStatuses[].user.linux` field is *the first attached* process identity to the first container process in the container. If the container has sufficient privilege to make system calls related to process identity (e.g. `setuid(2)`, `setgid(2)` or `setgroups(2)`, etc.), the container process can change its identity. Thus, the *actual* process identity will be dynamic.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

The following container runtimes are known to support fine-grained SupplementalGroups control.

CRI-level:

- containerd, since v2.0
- CRI-O, since v1.31

You can see if the feature is supported in the Node status.

```
 apiVersion: v1
kind: Node
...
status:
  features:
    supplementalGroupsPolicy: true
```

Note:

At this alpha release(from v1.31 to v1.32), when a pod with `SupplementalGroupsPolicy=Strict` are scheduled to a node that does NOT support this feature(i.e. `.status.features.supplementalGroupsPolicy=false`), the pod's supplemental groups policy falls back to the `Merge` policy *silently*.

However, since the beta release (v1.33), to enforce the policy more strictly, **such pod creation will be rejected by kubelet because the node cannot ensure the specified policy**. When your pod is rejected, you will see warning events with `reason=SupplementalGroupsPolicyNotSupported` like below:

```
 apiVersion: v1
kind: Event
...
type: Warning
```

```
reason: SupplementalGroupsPolicyNotSupported
message: "SupplementalGroupsPolicy=Strict is not supported in this node"
involvedObject:
  apiVersion: v1
  kind: Pod
  ...
```

Configure volume permission and ownership change policy for Pods

FEATURE STATE: `Kubernetes v1.23 [stable]`

By default, Kubernetes recursively changes ownership and permissions for the contents of each volume to match the `fsGroup` specified in a Pod's `securityContext` when that volume is mounted. For large volumes, checking and changing ownership and permissions can take a lot of time, slowing Pod startup. You can use the `fsGroupChangePolicy` field inside a `securityContext` to control the way that Kubernetes checks and manages ownership and permissions for a volume.

fsGroupChangePolicy - `fsGroupChangePolicy` defines behavior for changing ownership and permission of the volume before being exposed inside a Pod. This field only applies to volume types that support `fsGroup` controlled ownership and permissions. This field has two possible values:

- *OnRootMismatch*: Only change permissions and ownership if the permission and the ownership of root directory does not match with expected permissions of the volume. This could help shorten the time it takes to change ownership and permission of a volume.
- *Always*: Always change permission and ownership of the volume when volume is mounted.

For example:

```
securityContext:
 runAsUser: 1000
 runAsGroup: 3000
 fsGroup: 2000
 fsGroupChangePolicy: "OnRootMismatch"
```

Delegating volume permission and ownership change to CSI driver

FEATURE STATE: `Kubernetes v1.26 [stable]`

If you deploy a Container Storage Interface (CSI) driver which supports the `VOLUME_MOUNT_GROUP` `NodeServiceCapability`, the process of setting file ownership and permissions based on the `fsGroup` specified in the `securityContext` will be performed by the CSI driver instead of Kubernetes. In this case, since Kubernetes doesn't perform any ownership and permission change, `fsGroupChangePolicy` does not take effect, and as specified by CSI, the driver is expected to mount the volume with the provided `fsGroup`, resulting in a volume that is readable/writable by the `fsGroup`.

Set the security context for a Container

To specify security settings for a Container, include the `securityContext` field in the Container manifest. The `securityContext` field is a SecurityContext object. Security settings that you specify for a Container apply only to the individual Container, and they override settings made at the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a `securityContext` field:

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/hello-app:2.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-2.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

In your shell, list the running processes:

The output shows that the processes are running as user 2000. This is the value of `runAsUser` specified for the Container. It overrides the value 1000 that is specified for the Pod.

```
 USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
2000         1  0.0  0.0   4336   764 ?        Ss   20:36   0:00 /bin/sh -c node server.js
```

```
2000         8  0.1  0.5 772124 22604 ?        Sl   20:36   0:00 node server.js
...
```

Exit your shell:

Set capabilities for a Container

With Linux capabilities, you can grant certain privileges to a process without granting all the privileges of the root user. To add or drop Linux capabilities for a Container, include the `capabilities` field in the `securityContext` section of the Container manifest.

First, see what happens when you don't include a `capabilities` field. Here is configuration file that does not add or drop any Container capabilities:

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
  - name: sec-ctx-3
    image: gcr.io/google-samples/hello-app:2.0
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-3.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-3
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-3 -- sh
```

In your shell, list the running processes:

The output shows the process IDs (PIDs) for the Container:

```
 USER  PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root    1  0.0  0.0   4336   796 ?     Ss   18:17   0:00 /bin/sh -c node server.js
root    5  0.1  0.5 772124 22700 ?     Sl   18:17   0:00 node server.js
```

In your shell, view the status for process 1:

The output shows the capabilities bitmap for the process:

```
 ...
CapPrm:  00000000a80425fb
CapEff:  00000000a80425fb
...
```

Make a note of the capabilities bitmap, and then exit your shell:

Next, run a Container that is the same as the preceding container, except that it has additional capabilities set.

Here is the configuration file for a Pod that runs one Container. The configuration adds the `CAP_NET_ADMIN` and `CAP_SYS_TIME` capabilities:

```
 apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/hello-app:2.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-4.yaml
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-4 -- sh
```

In your shell, view the capabilities for process 1:

The output shows capabilities bitmap for the process:

```
 ...
CapPrm:  00000000aa0435fb
CapEff:  00000000aa0435fb
...
```

Compare the capabilities of the two Containers:

```
 00000000a80425fb
00000000aa0435fb
```

In the capability bitmap of the first container, bits 12 and 25 are clear. In the second container, bits 12 and 25 are set. Bit 12 is `CAP_NET_ADMIN`, and bit 25 is `CAP_SYS_TIME`. See capability.h for definitions of the capability constants.

Note:

Linux capability constants have the form `CAP_XXX`. But when you list capabilities in your container manifest, you must omit the `CAP_` portion of the constant. For example, to add `CAP_SYS_TIME`, include `SYS_TIME` in your list of capabilities.

Set the Seccomp Profile for a Container

To set the Seccomp profile for a Container, include the `seccompProfile` field in the `securityContext` section of your Pod or Container manifest. The `seccompProfile` field is a SeccompProfile object consisting of `type` and `localhostProfile`. Valid options for `type` include `RuntimeDefault`, `Unconfined`, and `Localhost`. `localhostProfile` must only be set if `type: Localhost`. It indicates the path of the pre-configured profile on the node, relative to the kubelet's configured Seccomp profile location (configured with the `--root-dir` flag).

Here is an example that sets the Seccomp profile to the node's container runtime default profile:

```
 ...
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

Here is an example that sets the Seccomp profile to a pre-configured file at `<kubelet-root-dir>/seccomp/my-profiles/profile-allow.json`:

```
 ...
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: my-profiles/profile-allow.json
```

Set the AppArmor Profile for a Container

To set the AppArmor profile for a Container, include the `appArmorProfile` field in the `securityContext` section of your Container. The `appArmorProfile` field is a AppArmorProfile object consisting of `type` and

`localhostProfile`. Valid options for `type` include `RuntimeDefault` (default), `Unconfined`, and `Localhost`. `localhostProfile` must only be set if `type` is `Localhost`. It indicates the name of the pre-configured profile on the node. The profile needs to be loaded onto all nodes suitable for the Pod, since you don't know where the pod will be scheduled. Approaches for setting up custom profiles are discussed in Setting up nodes with profiles.

Note: If `containers[*].securityContext.appArmorProfile.type` is explicitly set to `RuntimeDefault`, then the Pod will not be admitted if AppArmor is not enabled on the Node. However if `containers[*].securityContext.appArmorProfile.type` is not specified, then the default (which is also `RuntimeDefault`) will only be applied if the node has AppArmor enabled. If the node has AppArmor disabled the Pod will be admitted but the Container will not be restricted by the `RuntimeDefault` profile.

Here is an example that sets the AppArmor profile to the node's container runtime default profile:

```
 ...
containers:
- name: container-1
  securityContext:
    appArmorProfile:
      type: RuntimeDefault
```

Here is an example that sets the AppArmor profile to a pre-configured profile named `k8s-apparmor-example-deny-write`:

```
 ...
containers:
- name: container-1
  securityContext:
    appArmorProfile:
      type: Localhost
      localhostProfile: k8s-apparmor-example-deny-write
```

For more details please see, Restrict a Container's Access to Resources with AppArmor.

Assign SELinux labels to a Container

To assign SELinux labels to a Container, include the `seLinuxOptions` field in the `securityContext` section of your Pod or Container manifest. The `seLinuxOptions` field is an SELinuxOptions object. Here's an example that applies an SELinux level:

```
 ...
securityContext:
  seLinuxOptions:
    level: "s0:c123,c456"
```

Note:

To assign SELinux labels, the SELinux security module must be loaded on the host operating system. On Windows and Linux worker nodes without SELinux support, this field and any SELinux feature gates described below have no effect.

Efficient SELinux volume relabeling

FEATURE STATE: `Kubernetes v1.28 [beta]` (enabled by default)

Note:

Kubernetes v1.27 introduced an early limited form of this behavior that was only applicable to volumes (and PersistentVolumeClaims) using the `ReadWriteOncePod` access mode.

Kubernetes v1.33 promotes `SELinuxChangePolicy` and `SELinuxMount` feature gates as beta to widen that performance improvement to other kinds of PersistentVolumeClaims, as explained in detail below. While in beta, `SELinuxMount` is still disabled by default.

With `SELinuxMount` feature gate disabled (the default in Kubernetes 1.33 and any previous release), the container runtime recursively assigns SELinux label to all files on all Pod volumes by default. To speed up this process, Kubernetes can change the SELinux label of a volume instantly by using a mount option `-o context=<label>`.

To benefit from this speedup, all these conditions must be met:

- The feature gate `SELinuxMountReadWriteOncePod` must be enabled.
- Pod must use PersistentVolumeClaim with applicable `accessModes` and feature gates:
  - Either the volume has `accessModes: ["ReadWriteOncePod"]`, and feature gate `SELinuxMountReadWriteOncePod` is enabled.
  - Or the volume can use any other access modes and all feature gates `SELinuxMountReadWriteOncePod`, `SELinuxChangePolicy` and `SELinuxMount` must be enabled and the Pod has `spec.securityContext.seLinuxChangePolicy` either nil (default) or `MountOption`.
- Pod (or all its Containers that use the PersistentVolumeClaim) must have `seLinuxOptions` set.
- The corresponding PersistentVolume must be either:
  - A volume that uses the legacy in-tree `iscsi`, `rbd` or `fc` volume type.
  - Or a volume that uses a CSI driver. The CSI driver must announce that it supports mounting with `-o context` by setting `spec.seLinuxMount: true` in its CSIDriver instance.

When any of these conditions is not met, SELinux relabelling happens another way: the container runtime recursively changes the SELinux label for all inodes (files and directories) in the volume. Calling out explicitly, this applies to Kubernetes ephemeral volumes like `secret`, `configMap` and `projected`, and all volumes whose CSIDriver instance does not explicitly announce mounting with `-o context`.

When this speedup is used, all Pods that use the same applicable volume concurrently on the same node **must have the same SELinux label**. A Pod with a different SELinux label will fail to start and will be `ContainerCreating` until all Pods with other SELinux labels that use the volume are deleted.

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default)

For Pods that want to opt-out from relabeling using mount options, they can set `spec.securityContext.seLinuxChangePolicy` to `Recursive`. This is required when multiple pods share a single volume on the same node, but they run with different SELinux labels that allows simultaneous access to the volume. For example, a privileged pod running with label `spc_t` and an unprivileged pod running with the default label `container_file_t`. With unset `spec.securityContext.seLinuxChangePolicy` (or with the default value `MountOption`), only one of such pods is able to run on a node, the other one gets ContainerCreating with error `conflicting SELinux labels of volume <name of the volume>: <label of the running pod> and <label of the pod that can't start>`.

SELinuxWarningController

To make it easier to identify Pods that are affected by the change in SELinux volume relabeling, a new controller called `SELinuxWarningController` has been introduced in kube-controller-manager. It is disabled by default and can be enabled by either setting the `--controllers=*,selinux-warning-controller` command line flag, or by setting `genericControllerManagerConfiguration.controllers` field in KubeControllerManagerConfiguration. This controller requires `SELinuxChangePolicy` feature gate to be enabled.

When enabled, the controller observes running Pods and when it detects that two Pods use the same volume with different SELinux labels:

1. It emits an event to both of the Pods. `kubectl describe pod <pod-name>` the shows `SELinuxLabel "<label on the pod>" conflicts with pod <the other pod name> that uses the same volume as this pod with SELinuxLabel "<the other pod label>". If both pods land on the same node, only one of them may access the volume`.
2. Raise `selinux_warning_controller_selinux_volume_conflict` metric. The metric has both pod names + namespaces as labels to identify the affected pods easily.

A cluster admin can use this information to identify pods affected by the planning change and proactively opt-out Pods from the optimization (i.e. set `spec.securityContext.seLinuxChangePolicy: Recursive`).

Warning:

We strongly recommend clusters that use SELinux to enable this controller and make sure that `selinux_warning_controller_selinux_volume_conflict` metric does not report any conflicts before enabling `SELinuxMount` feature gate or upgrading to a version where `SELinuxMount` is enabled by default.

Feature gates

The following feature gates control the behavior of SELinux volume relabeling:

- `SELinuxMountReadWriteOncePod`: enables the optimization for volumes with `accessModes: ["ReadWriteOncePod"]`. This is a very safe feature gate to enable, as it cannot happen that two pods can share one single volume with this access mode. This feature gate is enabled by default sine v1.28.

- `SELinuxChangePolicy`: enables `spec.securityContext.seLinuxChangePolicy` field in Pod and related SELinuxWarningController in kube-controller-manager. This feature can be used before enabling `SELinuxMount` to check Pods running on a cluster, and to pro-actively opt-out Pods from the optimization. This feature gate requires `SELinuxMountReadWriteOncePod` enabled. It is beta and enabled by default in 1.33.
- `SELinuxMount` enables the optimization for all eligible volumes. Since it can break existing workloads, we recommend enabling `SELinuxChangePolicy` feature gate + SELinuxWarningController first to check the impact of the change. This feature gate requires `SELinuxMountReadWriteOncePod` and `SELinuxChangePolicy` enabled. It is beta, but disabled by default in 1.33.

Managing access to the `/proc` filesystem

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default)

For runtimes that follow the OCI runtime specification, containers default to running in a mode where there are multiple paths that are both masked and read-only. The result of this is the container has these paths present inside the container's mount namespace, and they can function similarly to if the container was an isolated host, but the container process cannot write to them. The list of masked and read-only paths are as follows:

- Masked Paths:
  - `/proc/asound`
  - `/proc/acpi`
  - `/proc/kcore`
  - `/proc/keys`
  - `/proc/latency_stats`
  - `/proc/timer_list`
  - `/proc/timer_stats`
  - `/proc/sched_debug`
  - `/proc/scsi`
  - `/sys/firmware`
  - `/sys/devices/virtual/powercap`
- Read-Only Paths:
  - `/proc/bus`
  - `/proc/fs`
  - `/proc/irq`
  - `/proc/sys`
  - `/proc/sysrq-trigger`

For some Pods, you might want to bypass that default masking of paths. The most common context for wanting this is if you are trying to run containers within a Kubernetes container (within a pod).

The `securityContext` field `procMount` allows a user to request a container's `/proc` be `Unmasked`, or be mounted as read-write by the container process. This also applies to `/sys/firmware` which is not in `/proc`.

```
...
securityContext:
  procMount: Unmasked
```

Note:

Setting `procMount` to Unmasked requires the `spec.hostUsers` value in the pod spec to be `false`. In other words: a container that wishes to have an Unmasked `/proc` or unmasked `/sys` must also be in a user namespace. Kubernetes v1.12 to v1.29 did not enforce that requirement.

Discussion

The security context for a Pod applies to the Pod's Containers and also to the Pod's Volumes when applicable. Specifically `fsGroup` and `seLinuxOptions` are applied to Volumes as follows:

- `fsGroup`: Volumes that support ownership management are modified to be owned and writable by the GID specified in `fsGroup`. See the Ownership Management design document for more details.
- `seLinuxOptions`: Volumes that support SELinux labeling are relabeled to be accessible by the label specified under `seLinuxOptions`. Usually you only need to set the `level` section. This sets the Multi-Category Security (MCS) label given to all Containers in the Pod as well as the Volumes.

Warning:

After you specify an MCS label for a Pod, all Pods with the same label can access the Volume. If you need inter-Pod protection, you must assign a unique MCS label to each Pod.

Clean up

Delete the Pod:

```
kubectl delete pod security-context-demo
kubectl delete pod security-context-demo-2
kubectl delete pod security-context-demo-3
kubectl delete pod security-context-demo-4
```

What's next

- PodSecurityContext
- SecurityContext
- CRI Plugin Config Guide

- Security Contexts design document
- Ownership Management design document
- PodSecurity Admission
- AllowPrivilegeEscalation design document
- For more information about security mechanisms in Linux, see Overview of Linux Kernel Security Features (Note: Some information is out of date)
- Read about User Namespaces for Linux pods.
- Masked Paths in the OCI Runtime Specification

Source: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/