Practical Challenges *with* Pod Security Admission

V Körbes
Christian Schlotter

April 21, 2023

tl;dr

Pod Security Admission is great when used as intended.

But every workload and their dog requires privileges of some sort, and then they need exemptions, and then all hell breaks loose.

In this talk we’ll discuss how to navigate that, and what you can do to at least partially *un-hell-break-loose-ify* your Kubernetes.

Agenda

What is Pod Security Admission?

A quick recap.

Adopting Pod Security Admission for Existing Services

The main challenges and pitfalls.

Practical Examples

Action!

Pod Security Admission Guidelines

Using PSA as a guide, not a limitation.

Oops

Ehhh, one more thing…

What is Pod Security Admission?

A quick recap.

What is Pod Security Admission?

A quick recap.

Pod Security Admission is an admission controller that lets you classify workloads into two separate axes.

The first axis is the three distinct profiles: **restricted**, **baseline**, and **privileged**.

The second axis is the three admission control modes: **audit**, **warn**, and **enforce**.

What is Pod Security Admission?

A quick recap.

The **restricted** profile restricts everything unless otherwise specified, **privileged** means there are no restrictions, and **baseline** is a good middle ground.

These restrictions apply to workload capabilities, in terms of (as of today):

- AppArmor
- Capabilities
- Host Namespaces
- Host Ports
- HostPath Volumes
- HostProcess
- /proc Mount Type
- Privilege Escalation
- Privileged Containers
- Running as Non-root user
- Seccomp
- SELinux
- Sysctls
- Volume Types

Full documentation at: https://kubernetes.io/docs/concepts/security/pod-security-standards/

What is Pod Security Admission?

Fun fact #1:

Yes, there is a capability called **Capabilities**!

See:
https://man7.org/linux/man-pages/man7/capabilities.7.html

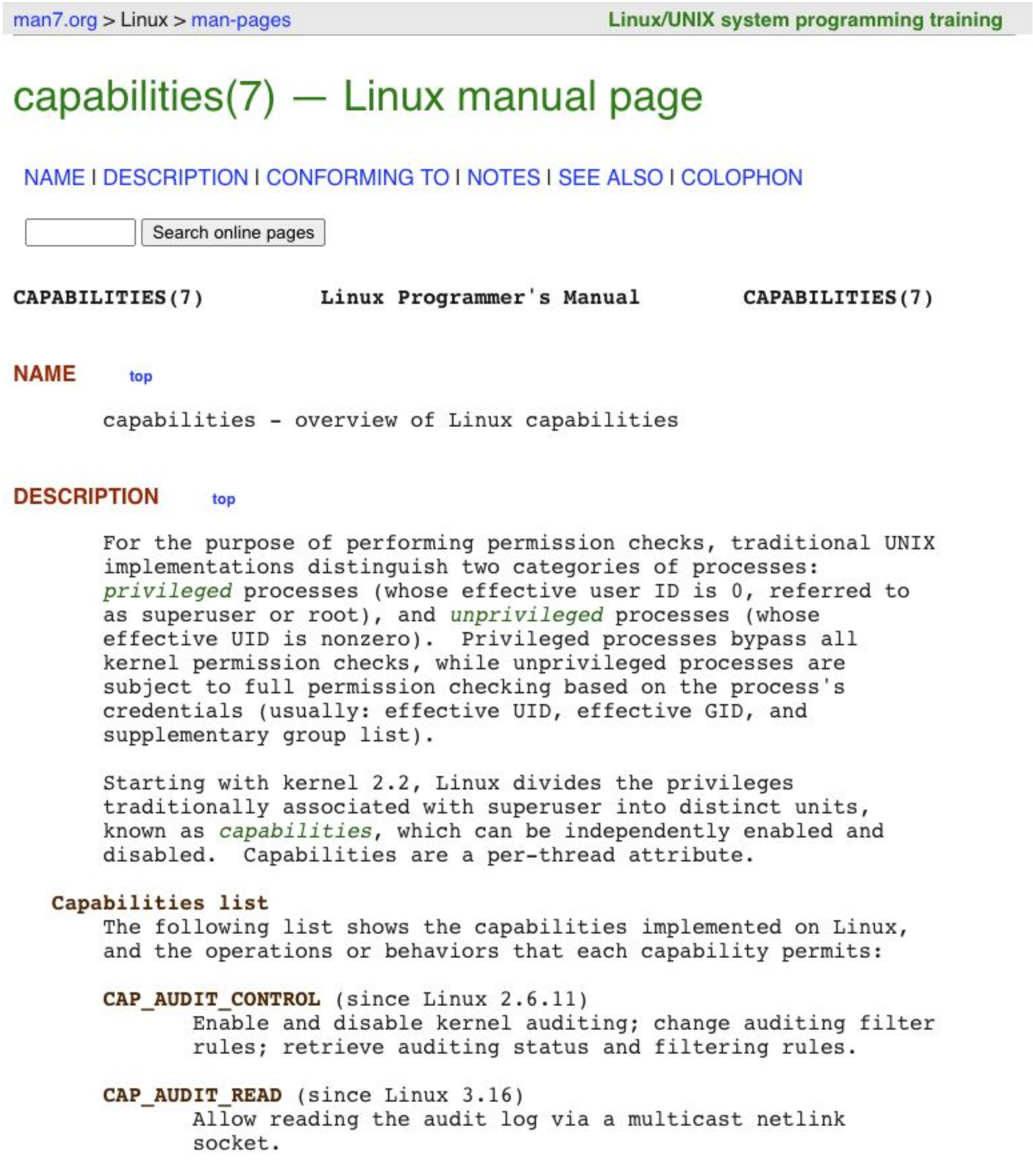

s Pod Security Admission?
ecap.

ricted profile restricts everything unless otherwise specified, pr
ctions, and **baseline** is a good middle ground.

estrictions apply to workload capabilities, in terms of (as of toda

?¿?

- AppArmor
- Capabilities
- Host Namespaces
- Host Ports
- HostPath Volumes
- HostProcess
- /proc Mount Type
- Privilege Escalation
- Privileged Containe
- Running as Non-roo
- Seccomp
- SELinux
- Sysctls
- Volume Types


man7.org > Linux > man-pages   Linux/UNIX system programming training

capabilities(7) — Linux manual page

NAME | DESCRIPTION | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

Search online pages

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

NAME top

```
       capabilities - overview of Linux capabilities
```

DESCRIPTION top

```
       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 nonzero).  Privileged processes bypass all
       kernel permission checks, while unprivileged processes are
       subject to full permission checking based on the process's
       credentials (usually: effective UID, effective GID, and
       supplementary group list).

       Starting with kernel 2.2, Linux divides the privileges
       traditionally associated with superuser into distinct units,
       known as capabilities, which can be independently enabled and
       disabled.  Capabilities are a per-thread attribute.

   Capabilities list
       The following list shows the capabilities implemented on Linux,
       and the operations or behaviors that each capability permits:

       CAP_AUDIT_CONTROL (since Linux 2.6.11)
              Enable and disable kernel auditing; change auditing filter
              rules; retrieve auditing status and filtering rules.

       CAP_AUDIT_READ (since Linux 3.16)
              Allow reading the audit log via a multicast netlink
              socket.
```

What is Pod Security Admission?

Fun fact #2:

The `privileged` Pod Security profile **is not** the same as `privileged: true`!

Story:

> "Hey, does your app require the `privileged` Pod Security profile or is it okay with `baseline` ?"
>
> "We're not setting the `privileged: true` boolean so it'd be fine… right?"

The Pod Security profile takes a look at lots of more fields in a Pods spec!

```
❯ kubectl explain --recursive pod.spec # (reduced output!)
...
FIELDS:
   containers / ephemeralContainers / initContainers:
      ports <[]Object>
      securityContext    <Object>
         allowPrivilegeEscalation    <boolean>
         capabilities    <Object>
         privileged <boolean>
         procMount   <string>
         runAsNonRoot    <boolean>
         seLinuxOptions <Object>
         seccompProfile <Object>
         ...
      ...
   hostIPC   <boolean>
   hostNetwork   <boolean>
   hostPID   <boolean>
   securityContext   <Object>
      runAsUser <integer>
      seLinuxOptions     <Object>
      seccompProfile     <Object>
      sysctls    <[]Object>
      ...
   volumes   <[]Object>
   ...
```

What is Pod Security Admission?

As for the three enforcement modes:

Audit creates audit log messages.

Warn does return warnings to the clients e.g. kubectl, client-go, etc.

Enforce actually stops the workload from being deployed.

All three only apply for creating or updating Pods!

What is Pod Security Admission?

So these are the options you've got:

- The policy: one of **`restricted`**, **`baseline`** and **`privileged`**
- The mode: one of **audit**, **warn** or **enforce**

It's pretty simple, but quite powerful. Elegant!

These settings can be applied **per namespace**, or **cluster-wide**.

If you need more granular control than this, look into tools such as:

- Kubewarden
- Kyverno
- OPA Gatekeeper

What is Pod Security Admission?

Configuration: per Namespace

Per-Namespace configuration can be done via labels on Namespaces.

Two labels per mode

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: my-baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.27
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: v1.27
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: v1.27
```

What is Pod Security Admission?

Configuration: cluster-wide

Cluster-wide configuration can be done via `AdmissionConfiguration` for the Kubernetes API Server.

Defaults if no Namespace labels are set

Cluster-wide exemptions

```yaml
apiVersion: apiserver.config.k8s.io/v1 # see compatibility note
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "privileged"
      enforce-version: "latest"
      audit: "privileged"
      audit-version: "latest"
      warn: "privileged"
      warn-version: "latest"
    exemptions:
      usernames: [] # Array of authenticated usernames to exempt.
      runtimeClasses: [] # Array of runtime class names to exempt.
      namespaces: [] # Array of namespaces to exempt.
```

Adopting Pod Security Admission for Existing Services

Main challenges and pitfalls.

Adopting Pod Security Admission for Existing Services

Main challenges and pitfalls.

Adopting Pod Security Admission has three steps:

- **Get everything running:** Pick the right profile.
- **Optimize:** Remove unnecessary privileges.
- **Continuity:** Develop with security profiles in mind.

Let’s start with the first two. Theory, then action.

Adopting Pod Security Admission for Existing Services

Step 1: Pick the right profile to get everything running

The rule of thumb is that if it needs:

- Host namespaces
- Privileged (in the pod spec!)
- Administrative capabilities (e.g. `CAP_SYS_ADMIN`, `NET_ADMIN`, …)
- HostPath volumes
- Special AppArmor, SELinux or `seccomp` configuration
- `/proc` mount
- Special unsafe `sysctl`'s

…it will need the **privileged** profile!

For a detailed list, see https://kubernetes.io/docs/concepts/security/pod-security-standards/

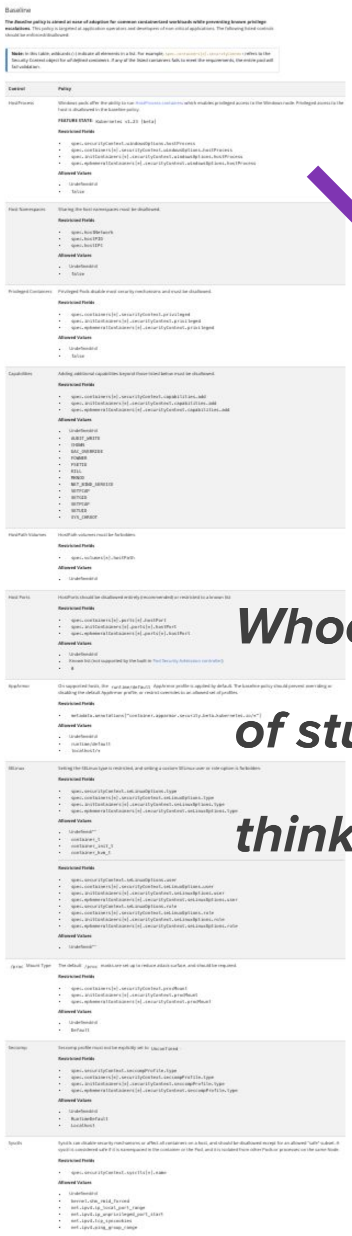
Privileged:

The *Privileged* policy is purposely-open, and entirely unrestricted. This type of policy is typically aimed at system- and infrastructure-level workloads managed by privileged, trusted users.

The Privileged policy is defined by an absence of restrictions. Allow-by-default mechanisms (such as gatekeeper) may be Privileged by default. In contrast, for a deny-by-default mechanism (such as Pod Security Policy) the Privileged policy should disable all restrictions.

"Do what you want."

Baseline:



Whoooooole lot of stuff to think about.

Control	Policy
HostProcess	Windows pods offer the ability to run HostProcess containers which enables privileged access to the Windows node. Privileged access to the host is disallowed in the baseline policy. **FEATURE STATE:** `Kubernetes v1.23 [beta]` **Restricted Fields** • `spec.securityContext.windowsOptions.hostProcess` • `spec.containers[*].securityContext.windowsOptions.hostProcess` • `spec.initContainers[*].securityContext.windowsOptions.hostProcess` • `spec.ephemeralContainers[*].securityContext.windowsOptions.hostProcess` **Allowed Values** • Undefined/nil • `false`
Host Namespaces	Sharing the host namespaces must be disallowed. **Restricted Fields** • `spec.hostNetwork` • `spec.hostPID` • `spec.hostIPC` **Allowed Values** • Undefined/nil • `false`
Privileged Containers	Privileged Pods disable most security mechanisms and must be disallowed. **Restricted Fields**

In the beginning you'll have to set a lot of workloads to **privileged** to get things running.

That's okay.

We'll fix it in the next step.

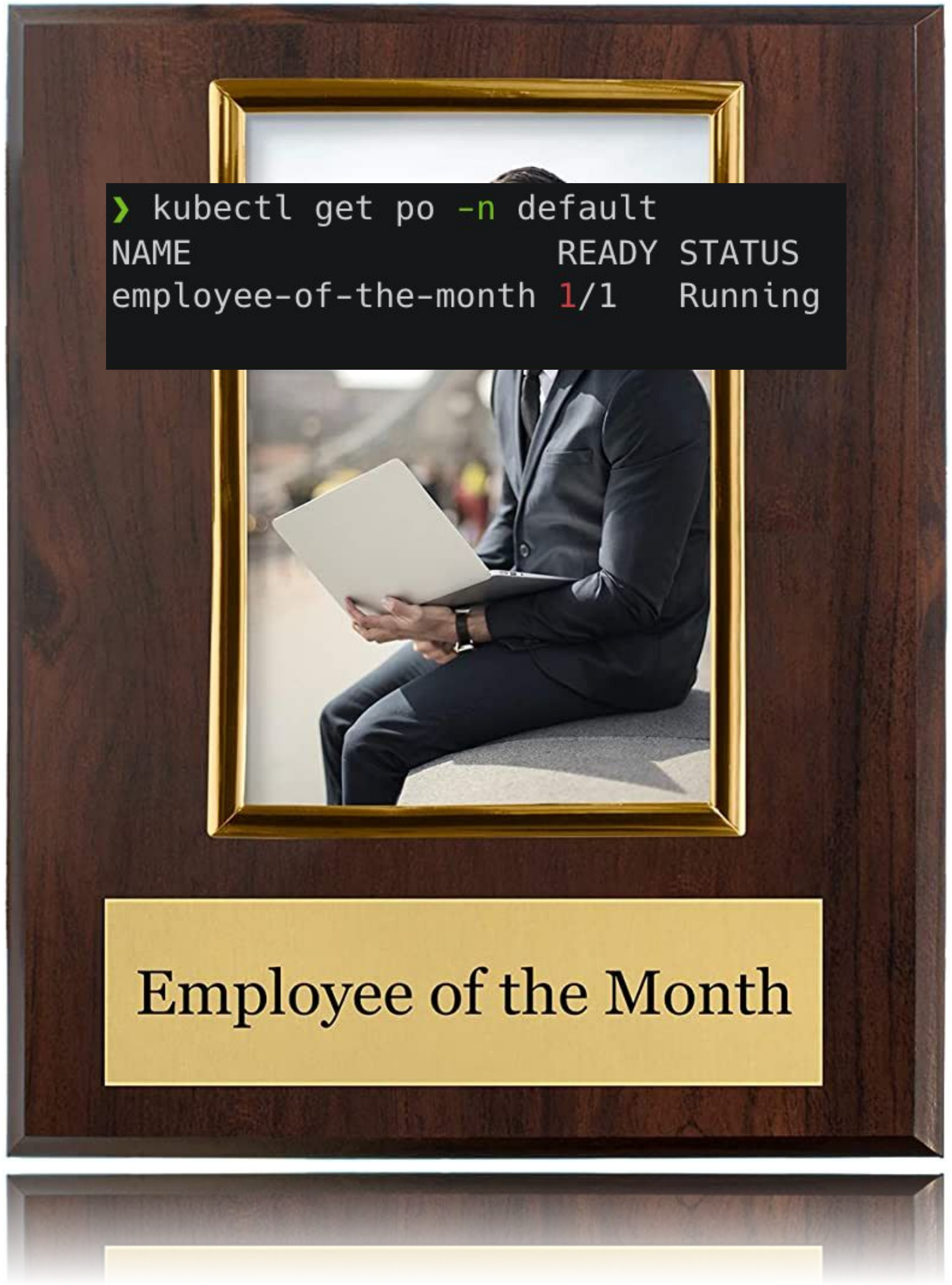
Adopting Pod Security Admission for Existing Services

Step 1: Pick the right profile to get everything running

If you try to restrict something that needs privileges, bad things can happen:

- A Pod does not get scheduled
  - When a stricter Pod Security policy is enforced on the namespace or cluster-wide and the Pod does not meet the requirements
- The process may not even start ➜ `CrashLoopBackoff`
- The process may start and is `Running`, but:
  - Does **not** get `ready` and does **not** fulfill its desired work
  - Does get `ready` but does **not** fulfill its desired work

This is important.



❯ kubectl get po -n default
NAME READY STATUS
employee-of-the-month 1/1 Running
Employee of the Month

Adopting Pod Security Admission for Existing Services

Step 2: Optimize by removing unnecessary privileges

Must-Set Settings:

- There are fields in the Pod specification which:
  - **don’t get set** explicitly, and;
  - have **insecure** defaults

To be compliant with the **restricted** policy you **have** to explicitly set them to a secure value, or you will be insecure by default.

The **restricted** policy requires you to actively **remove** privileges that are not needed.

Let’s see…

Adopting Pod Security Admission for Existing Services

Step 2: Must-Set Settings

`Pod.spec.securityContext`:

- Add: `runAsNonRoot: true`
  - Maybe also requires `runAsUser` in container spec or adjusting the container image / Dockerfile
  - Be aware: Container may not start if the image defines
    - the user by name, not id
    - The `root` user / `id=0`

What?

This setting ensures that the containers **must** run as non-root user. The kubelet validates the image and container spec at runtime to ensure it doesn't start as user id `0`.

If no explicit user id is configured at `Pod.Spec.containers.securityContext.runAsUser`, the image must have a user id configured which is not `0`.

Note: It is also sufficient if it is defined on all container's `securityContext`.

Adopting Pod Security Admission for Existing Services

Step 2: Must-Set Settings

`Pod.spec.securityContext`:

- Add: `seccompProfile.type`, e.g.:

```
seccompProfile:
  type: "RuntimeDefault"
```

What?

The `seccomp` profile for all containers must be set and not be `unconfined`. `seccomp` filters the syscalls the processes can make.

With Kubernetes v1.27 the feature gate `SeccompDefault` will reach GA. This results in defaulting the `seccompProfile` to `RuntimeDefault` instead of `Unconfined`.

Note: It is also sufficient if it is defined on all container's `securityContext`.

Adopting Pod Security Admission for Existing Services

Step 2: Must-Set Settings

`Pod.spec.containers[].securityContext`:

- Add `allowPrivilegeEscalation: false`
- Add `capabilities: {“drop”:[ALL]}`

What?

Dropping permissions via `capabilities.drop` prevents multiple ways of privilege escalation.

The **restricted** profile allows one to re-add the `NET_BIND_SERVICE` capability.

The `allowPrivilegeEscalation` setting prevents processes or attackers from gaining new privileges by the `no_new_privs` flag.

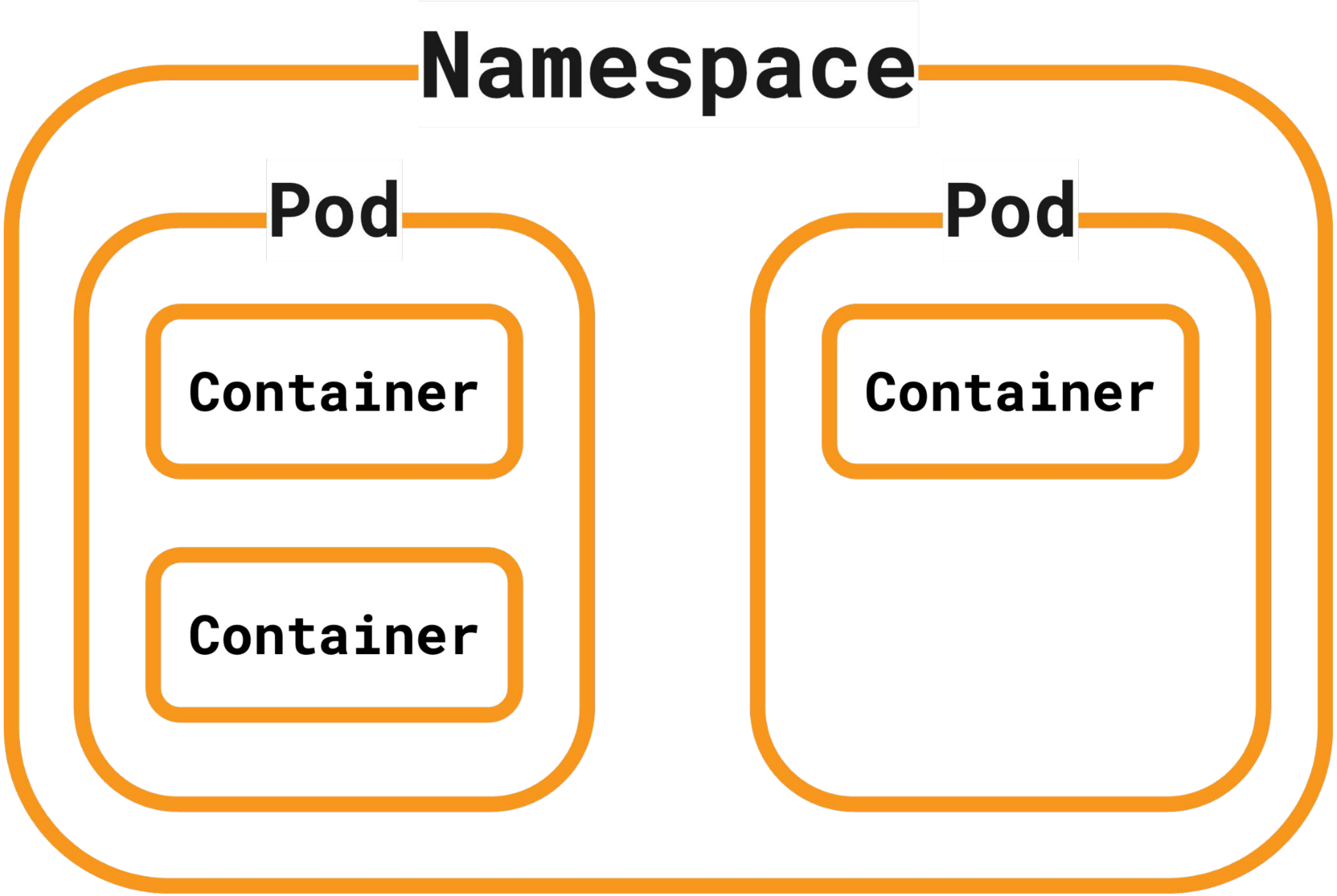
Adopting Pod Security Admission for Existing Services

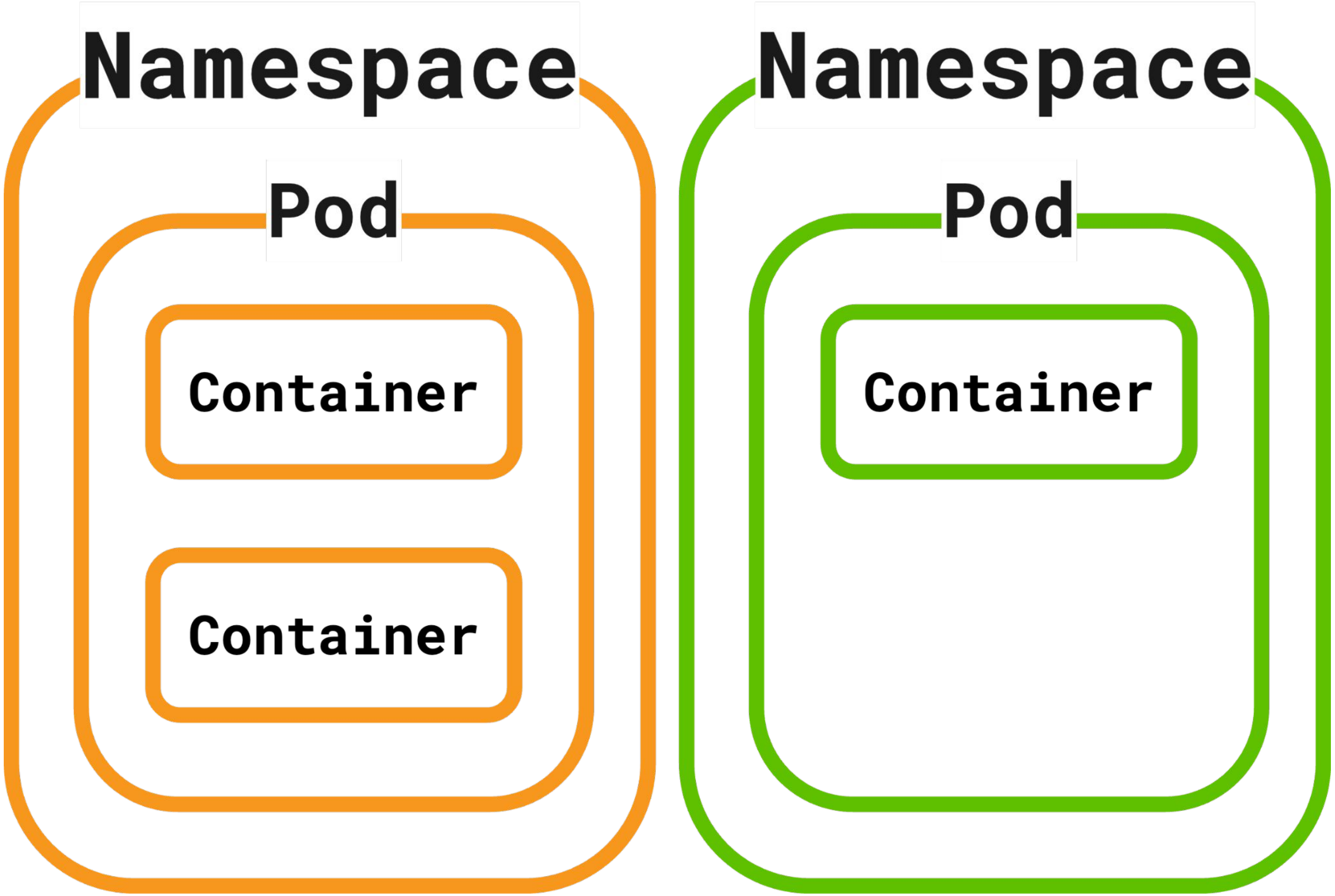
So far we talked about *removing* privileges.

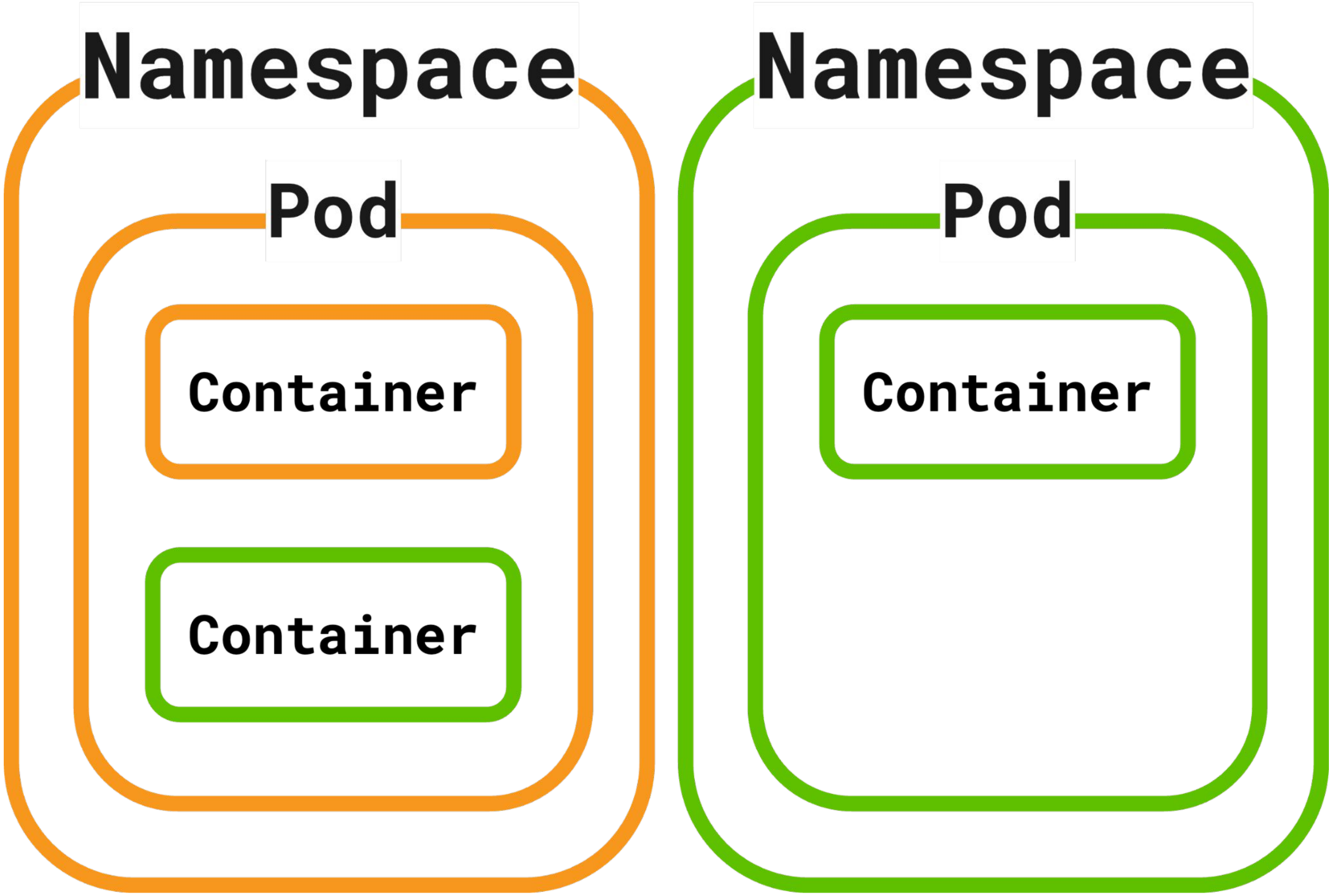
But what if a workload does need them?

Set the whole thing to privileged and we’re done, right?

Uhh, no.


Namespace
Pod
Container
Container
Pod
Container


Namespace
Pod
Container
Container
Namespace
Pod
Container


Namespace
Pod
Container
Container
Namespace
Pod
Container

For workloads that do need certain privileges, those are often only required for **specific parts** of the application.

The rest can safely remain within tighter restrictions.

Practical Examples

Practical Examples

Keep in mind the types of workloads that actually do need privileges to work:

- Hostpath
  - Parts of CSI (Container Storage Interfaces)
  - Monitoring, e.g. node-exporter
  - Any kind of log shipper, e.g. fluentbit
  - Cluster API Provider Docker
- HostNetwork
  - All CNIs (Container Network Interfaces)
  - A lot of Pods usually deployed to kube-system, e.g. kube-proxy
- HostPID, Privileged, …
  - Parts of CSI

Practical Examples

As mentioned earlier:

If you try to restrict something that needs privileges, bad things *will* happen.

Now, assuming **bad** things *have* happened…

How do you **identify** what's wrong with your workload?

Practical Examples

Option 1: Audit log

Audit and **warning** modes provide the same data in different ways.

For Audit mode, here’s how you see it:

```
❯ cat /var/log/kubernetes/audit.log  | jq 'select(.annotations["pod-security.kubernetes.io/audit-violations"])'
{
  "kind": "Event",
  "verb": "create",
  "requestObject": { "kind": "Pod", "apiVersion": "v1", "metadata": {...} },
  "annotations": {
    "pod-security.kubernetes.io/audit-violations": "would violate PodSecurity \"restricted:latest\":
allowPrivilegeEscalation != false (container \"my-privileged-pod\" must set securityContext.allowPrivilegeEscalation=false),
unrestricted capabilities (container \"my-privileged-pod\" must set securityContext.capabilities.drop=[\"ALL\"]),
runAsNonRoot != true (pod or container \"my-privileged-pod\" must set securityContext.runAsNonRoot=true), seccompProfile
(pod or container \"my-privileged-pod\" must set securityContext.seccompProfile.type to \"RuntimeDefault\" or
\"Localhost\")",
    "pod-security.kubernetes.io/enforce-policy": "privileged:latest"
    ...
  }
  ...
}
```

Practical Examples

Option 2: Warnings

On **warning** mode, here's how you get the same data you've seen before:

- Scenario 1: You have a workload already running, and try to set the **warning** mode at the namespace, then you get the warnings
  - Also works without actual changing the label via `--dry-run=server`

```
❯ kubectl label --dry-run=server namespace foo pod-security.kubernetes.io/enforce=restricted
Warning: existing pods in namespace "foo" violate the new PodSecurity enforce level "restricted:latest"
Warning: kuard-dcfccb87d-z4vbm: allowPrivilegeEscalation != false, unrestricted capabilities, runAsNonRoot != true,
seccompProfile
namespace/foo labeled
```

Practical Examples

Option 2: Warnings

On **warning** mode, here's how you get the same data you've seen before:

- Scenario 2: You have configured the **restricted** policy, and you try to create a workload

```
❯ kubectl apply --namespace foo -f foo.yaml
Warning: would violate PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container "kuard" must set
securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container "kuard" must set
securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "kuard" must set
securityContext.runAsNonRoot=true), seccompProfile (pod or container "kuard" must set securityContext.seccompProfile.type to
"RuntimeDefault" or "Localhost")
pod/kuard created
```

Note:

- With cluster-wide **warning** mode set to the **restricted** policy, you get the warnings when creating any kind of the built-in objects (Deployment, StatefulSet, ReplicaSet, Pod, …)
- Without cluster-wide **warning** mode enabled: Warnings will only be visible when applying a Pod, not when applying a Deployment (in that case you'd get a warning at the ReplicaSet)

Practical Examples

Option 3: Errors - No Pod

```
❯ kubectl get deploy,rs
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/kuard   0/1     0            0           64s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/kuard-6695df4ddd   1         0         0       64s

❯ kubectl get rs kuard-6695df4ddd -o jsonpath="{.status.conditions[0].message}"
pods "kuard-6695df4ddd-mw4xl" is forbidden: violates PodSecurity "restricted:latest":
allowPrivilegeEscalation != false (container "kuard" must set
securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container "kuard" must
set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "kuard" must
set securityContext.runAsNonRoot=true), seccompProfile (pod or container "kuard" must set
securityContext.seccompProfile.type to "RuntimeDefault" or "Localhost")
```

Practical Examples

Identifying issues

Remember:

If you try to restrict something that needs privileges, bad things happen.

- A Pod does not get scheduled
    - When a lower Pod Security policy is enforced on the namespace the Pod does not meet the requirements
- The process may not even start ➔ `CrashLoopBackoff`
- **The process may start and is `Running`, but:**
    - Does not get `ready` and does not fulfill its desired work
    - **Does get `ready` but does not fulfill its desired work**

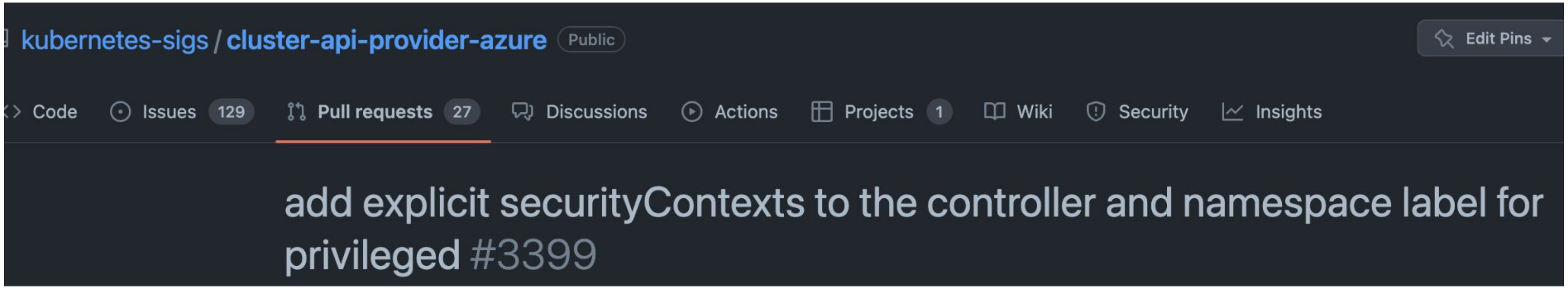
So we need to…

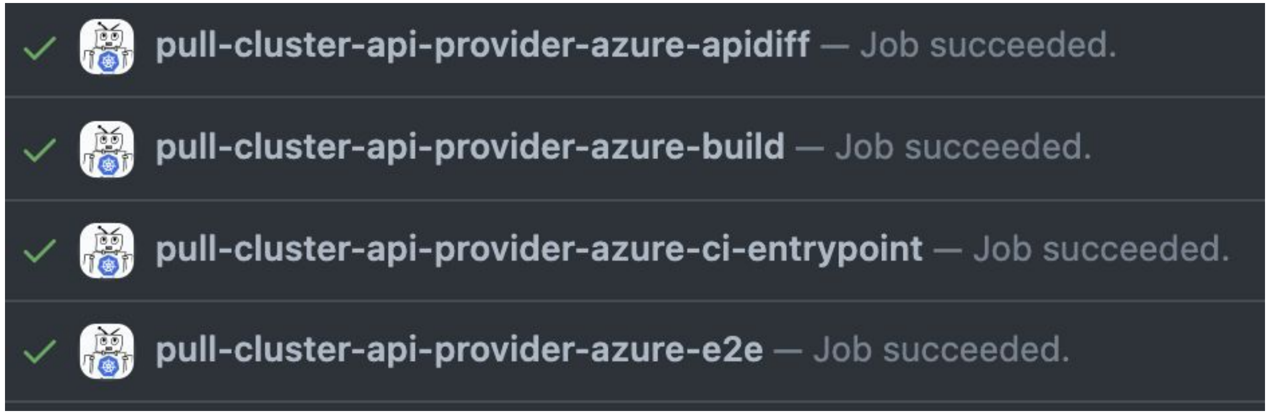
Practical Examples

Identifying issues

Use PSA as regression tests: Check the functionality afterwards! Logs! Tests!

- E.g. Cluster API was still provisioning clusters and worked as expected
- E.g. CSI Pod was mounting the disk correctly / pass upstream CSI tests





[Source]

Practical Examples

Identifying issues

If the tests pass? Great!

But if they don't, consider...

The fact that out-of-the-box it breaks doesn't mean **yet** that it's functionally incompatible.

It might be a matter of tweaking the configurations so they reflect more accurately the **actual requirements** of the application.

Source

Practical Examples

Is it a quick configuration fix?

Examples of upstream changes:

- Certificate manager / cert-manager: [Pull Request #5259]
    - We did use manifests from v1.9.x
    - E.g. Capability drop was added to v1.10.x yamls – from `baseline` to `restricted`

```diff
   containerSecurityContext:
     allowPrivilegeEscalation: false
-    # capabilities:
-    #   drop:
-    #   - ALL
```

```diff
   containerSecurityContext:
     allowPrivilegeEscalation: false
+    capabilities:
+      drop:
+      - ALL
```

- CAPI
    - Version went from `baseline` to `restricted` profile: [Pull Request #831]

A workload might work with less privileges, but be **configured** to require more. And then it fails.

But if the **functionality** doesn't actually need the privileges, it'll work just fine if you simply fix the settings.

Practical Examples

For other workloads privileges will be **functionally required**, and a straightforward fix might not be possible.

But there are still ways to **improve** security for them too:

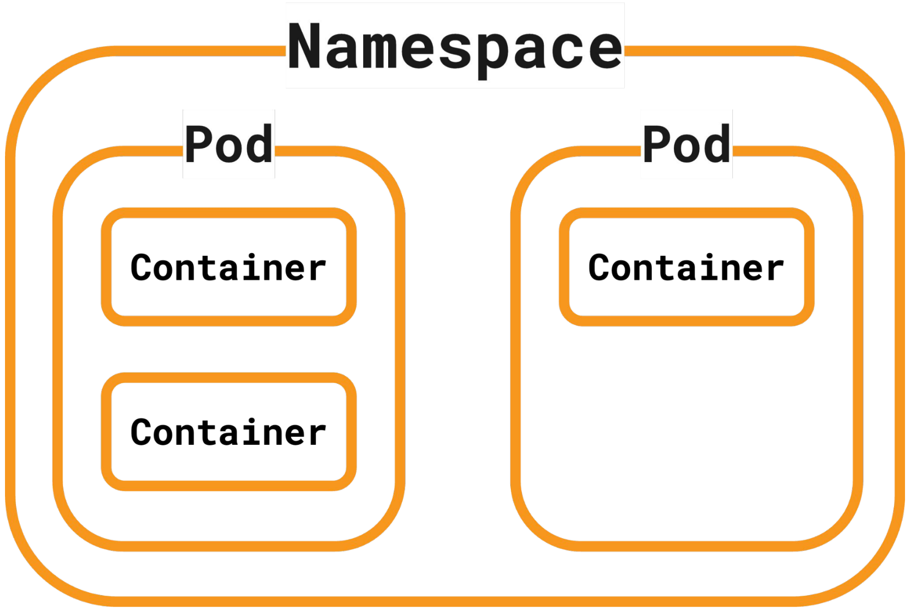
- **Namespace separation** for different profiles:
  - Take parts of an application that require privileges and put them in a privileged namespace. Everything else lives in a restricted namespace.
- **Per-container capabilities** assignment:
  - If multiple containers on the same pod can't be separated, only give special capabilities to the container that needs them.

Practical Examples

Real-world example: Container Storage Interface (CSI)

“It’s a CSI controller so it needs privileges, run it all as privileged!”

- Deployment **vsphere-csi-controller**
  - **csi-attacher**
  - **csi-resizer**
  - **vsphere-csi-controller**
  - **liveness-probe**
  - **vsphere-syncer**
  - **csi-provisioner**
  - **csi-snapshotter**
- DaemonSet **vsphere-csi-node**
  - **node-driver-registrar**
  - **vsphere-csi-node**
  - **liveness-probe**



Practical Examples

Real-world example: Container Storage Interface (CSI)

The workload required for CSI is often split in multiple Pods that require different capabilities. **Only some parts need privileges**, the rest need less privileges.

- The Deployment is able to run as **restricted**
  - It contains the CSI parts which talk to API's
- The DaemonSet requires **privileged** for **hostPaths** and **runAsUser: 0**, in order to:
  - Find, format, and mount disks
  - Talk to the kubelet via socket – CSI plug-in mechanism
  - *(The LivenessProbe container doesn't require privileges though!)*
  - *(And since it's an http endpoint, it's an attack-vector!)*

More:

Example manifests:

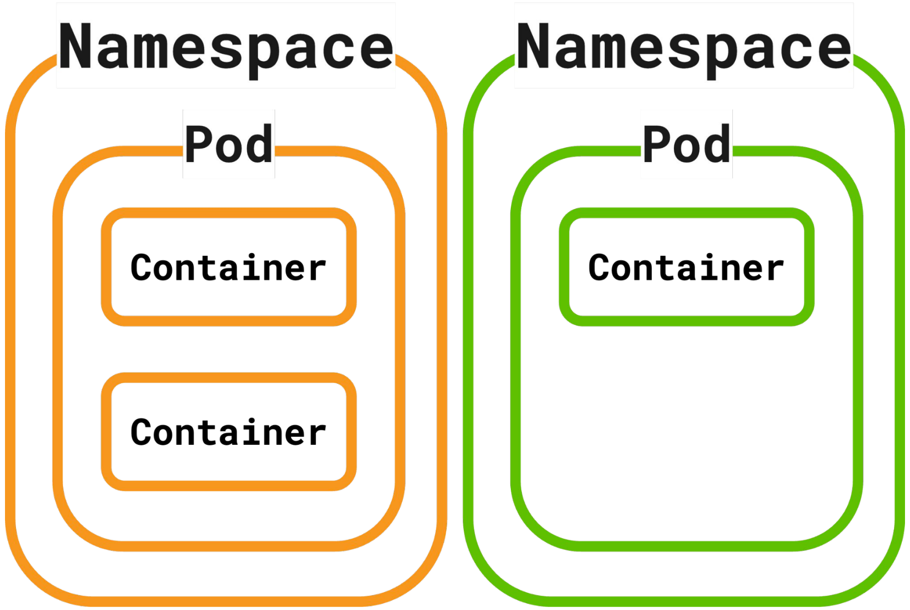
- VSphere CSI Driver
- AWS EBS CSI Driver

Practical Examples

Real-world example: Container Storage Interface (CSI)

“Privilege only for the required parts.”

- Deployment **vsphere-csi-controller** ➔ `restricted`
    - **csi-attacher**
    - **csi-resizer**
    - **vsphere-csi-controller**
    - **liveness-probe**
    - **vsphere-syncer**
    - **csi-provisioner**
    - **csi-snapshotter**
- DaemonSet **vsphere-csi-node**
    - **node-driver-registrar**
    - **vsphere-csi-node**
    - **liveness-probe**



Practical Examples

Example: LivenessProbe container of the CSI Daemonset (privileged)

There are 3 containers in this Pod:

- **Two** of them talk to the filesystem and the kubelet so they **need privileges**.
    - These containers require high privileges.

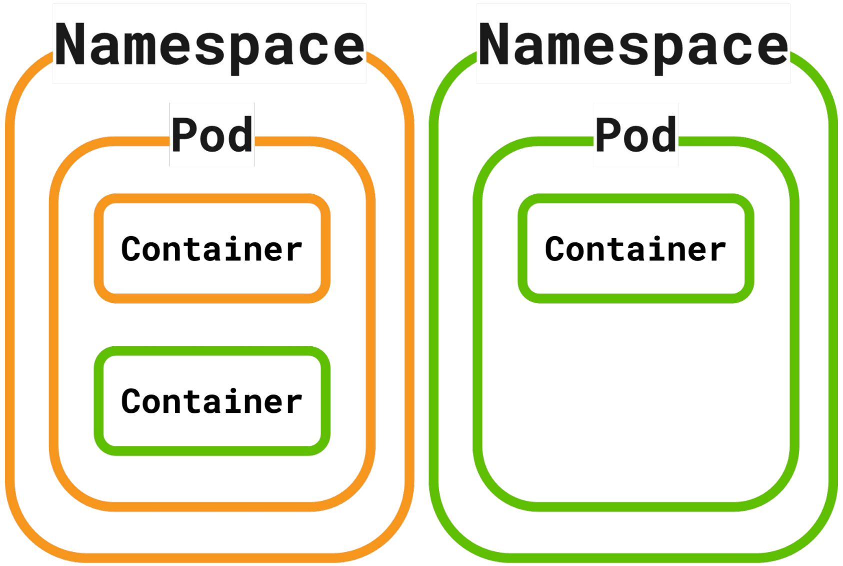
- The **third** one is a liveness probe, which needs **'nothing.'**
    - This container, which exposes a `/healthz` http endpoint, does not require privileges.
    - This container can be scoped down by
        - Set `runAsNonroot: true`
        - Set `allowPrivilegeEscalation: false`
        - Drop all capabilities

Practical Examples

Real-world example: Container Storage Interface (CSI)

“Privilege only for the required parts.”

- Deployment **vsphere-csi-controller** → `restricted`
  - **csi-attacher**
  - **csi-resizer**
  - **vsphere-csi-controller**
  - **liveness-probe**
  - **vsphere-syncer**
  - **csi-provisioner**
  - **csi-snapshotter**
- DaemonSet **vsphere-csi-node** → `privileged`
  - **node-driver-registrar** (but: no http server, only talks to kubelet)
  - **vsphere-csi-node** (same as above)
  - **liveness-probe** → “mostly” restricted



Practical Examples

Does every part of the application require privileges?

Real-world examples: Observability, e.g. Prometheus including `node-exporter`

- Prometheus, grafana, … could run `restricted`
- Node-exporter requires `hostPath` mounts
- Idea — should work!
  - `restricted` or at least `baseline`:
    - `deployment.apps/alertmanager`
    - `deployment.apps/prometheus-kube-state-metrics`
    - `deployment.apps/prometheus-pushgateway`
    - `deployment.apps/prometheus-server`
  - `privileged`:
    - `daemonset.apps/prometheus-node-exporter`
    - Because it requires `hostPath` mount to read information

Example: [prometheus-community/helm-charts]()

Some workloads require more privileges, but not every part of a workload does.

The smaller the surface area with high privileges, the better.

Pod Security Admission Guidelines

Using PSA as a guide, not a limitation.

Pod Security Admission Guidelines

As a developer, in a system with high PSA enforcement, what are the guidelines to create or configure new services?

The main advice is to **think of PSA preemptively** and let it guide you through the development process.

Pod Security Admission Guidelines

Guidelines on **transitioning** existing clusters to a higher-security profile:

1. Enable the **warn** and **audit** mode in the cluster-wide configuration for the target profile.
    - Helps to *identify* not-yet-transitioned workloads.
2. **Enforce** the higher-security profiles on a **per-namespace** basis.
    - *Prevents regressions* from happening on an already-transitioned namespace.
3. **Enforce cluster-wide defaults**.
    - Once you're sure it won't prevent pods from running.

Pod Security Admission Guidelines

Guidelines on **starting** new work, aiming for high-security profiles:

- Create a **new namespace** for your application
- Set **enforce mode** and version from the start — and work backwards to meet it
  - ➔ this ensures you won’t hit regressions by accident
- Have **E2E** tests to verify the **functionality**

Need privileges? **Compartmentalize**:

- ○ If the container can be in a **separate** Pod and Namespace:
  - ➔ Move to a **separate** Pod and Namespace
- ○ They need to stay in the **same** Pod?
  - ➔ **Remove privileges** from containers that don’t need them

Pod Security Admission Guidelines

Further reading:

- https://k8s.io/docs/setup/best-practices/enforcing-pod-security-standards/
- https://k8s.io/docs/tasks/configure-pod-container/enforce-standards-admission-controller/
- https://k8s.io/docs/tasks/configure-pod-container/migrate-from-psp/
- https://cloud.google.com/kubernetes-engine/docs/how-to/migrate-podsecuritypolicy

Pod Security Admission Guidelines

Use Kyverno or other tools to lint misconfigurations already in your CI:

They provide configuration for all three policies of Pod Security:

- https://github.com/kyverno/policies/pod-security/

```
❯ kustomize build https://github.com/kyverno/policies/pod-security/restricted > restricted.yaml
❯ kyverno apply restricted.yaml --resource foo.yaml

Applying 19 policy rules to 1 resource...

policy disallow-capabilities-strict -> resource default/Pod/kuard failed:
1. require-drop-all: validation failure: Containers must drop `ALL` capabilities.

pass: 18, fail: 1, warn: 0, error: 0, skip: 38
```

Pod Security Admission Guidelines

Wait, what about the `version` parameter of Pod Security Admission?

- **Do not** use `latest` or you may run in unexpected issues in future
  - Similar story as for `latest` in container image references; things may change on version upgrades
- As of today (and PSA is pretty new) there had been no big changes in-between PSA versions (yet).
- *But there might be!*

```yaml
apiVersion: apiserver.config.k8s.io/v1 # see compatibility note
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "privileged"
      enforce-version: "latest"
      audit: "privileged"
      audit-version: "latest"
      warn: "privileged"
      warn-version: "latest"
    exemptions:
      usernames: [] # Array of authenticated usernames to exempt.
      runtimeClasses: [] # Array of runtime class names to exempt.
      namespaces: [] # Array of namespaces to exempt.
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: my-baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.27
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: v1.27
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: v1.27
```

Just one more thing...

...and then you’re totally secure!

Just one more thing…

…and then you're totally secure!

- OPA Gatekeeper?
- eBPF?
- Secrets?
- RBAC?
- Plaintext credentials?
- Signed images?
- gVisor?
- Scanners?
- Espionage!
- Geese?

- Isolated node pools?
- KubeArmor?
- Tetragon?
- Falco?
- Base64 secrets?
- Node impersonation?
- SBOMs?
- Blinky lights?
- Firewall rules?
- Runtime behavior analytics?
- Secure enclave?

…and done. **Profit**!

Thank you!