Policies

Policies you can configure that apply to groups of resources.

- 1: Limit Ranges
- 2: Resource Quotas
- 3: Pod Security Policies
- 4: Process ID Limits And Reservations

1 - Limit Ranges

By default, containers run with unbounded compute resources on a Kubernetes cluster. With resource quotas, cluster administrators can restrict resource consumption and creation on a namespace basis. Within a namespace, a Pod or Container can consume as much CPU and memory as defined by the namespace's resource quota. There is a concern that one Pod or Container could monopolize all available resources. A LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace.

A *LimitRange* provides constraints that can:

- Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
- Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforce a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

Enabling LimitRange

LimitRange support has been enabled by default since Kubernetes 1.10.

A LimitRange is enforced in a particular namespace when there is a LimitRange object in that namespace.

The name of a LimitRange object must be a valid DNS subdomain name.

Overview of Limit Range

- The administrator creates one LimitRange in one namespace.
- Users create resources like Pods, Containers, and PersistentVolumeClaims in the namespace.
- The `LimitRanger` admission controller enforces defaults and limits for all Pods and Containers that do not set compute resource requirements and tracks usage to ensure it does not exceed resource minimum, maximum and ratio defined in any LimitRange present in the namespace.
- If creating or updating a resource (Pod, Container, PersistentVolumeClaim) that violates a LimitRange constraint, the request to the API server will fail with an HTTP status code `403 FORBIDDEN` and a message explaining the constraint that have been violated.
- If a LimitRange is activated in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values. Otherwise, the system may reject Pod creation.
- LimitRange validations occurs only at Pod Admission stage, not on Running Pods.

Examples of policies that could be created using limit range are:

- In a 2 node cluster with a capacity of 8 GiB RAM and 16 cores, constrain Pods in a namespace to request 100m of CPU with a max limit of 500m for CPU and request 200Mi for Memory with a max limit of 600Mi for Memory.
- Define default CPU limit and request to 150m and memory default request to 300Mi for Containers started with no cpu and memory requests in their specs.

In the case where the total limits of the namespace is less than the sum of the limits of the Pods/Containers, there may be contention for resources. In this case, the Containers or Pods will not be created.

Neither contention nor changes to a LimitRange will affect already created resources.

What's next

Refer to the LimitRanger design document for more information.

For examples on using limits, see:

- how to configure minimum and maximum CPU constraints per namespace.
- how to configure minimum and maximum Memory constraints per namespace.
- how to configure default CPU Requests and Limits per namespace.
- how to configure default Memory Requests and Limits per namespace.
- how to configure minimum and maximum Storage consumption per namespace.
- a detailed example on configuring quota per namespace.

2 - Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.

Resource quotas work like this:

- Different teams work in different namespaces. Currently this is voluntary, but support for making this mandatory via ACLs is planned.
- The administrator creates one ResourceQuota for each namespace.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a ResourceQuota.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code `403 FORBIDDEN` with a message explaining the constraint that would have been violated.
- If quota is enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the `LimitRanger` admission controller to force defaults for pods that make no compute resource requirements.

  See the walkthrough for an example of how to avoid this problem.

The name of a ResourceQuota object must be a valid DNS subdomain name.

Examples of policies that could be created using namespaces and quotas are:

- In a cluster with a capacity of 32 GiB RAM, and 16 cores, let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores, and hold 2GiB and 2 cores in reserve for future allocation.
- Limit the "testing" namespace to using 1 core and 1GiB RAM. Let the "production" namespace use any amount.

In the case where the total capacity of the cluster is less than the sum of the quotas of the namespaces, there may be contention for resources. This is handled on a first-come-first-served basis.

Neither contention nor changes to quota will affect already created resources.

Enabling Resource Quota

Resource Quota support is enabled by default for many Kubernetes distributions. It is enabled when the API server `--enable-admission-plugins=` flag has `ResourceQuota` as one of its arguments.

A resource quota is enforced in a particular namespace when there is a ResourceQuota in that namespace.

Compute Resource Quota

You can limit the total sum of compute resources that can be requested in a given namespace.

The following resource types are supported:

Resource Name	Description
`limits.cpu`	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
`limits.memory`	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
`hugepages-<size>`	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
`cpu`	Same as `requests.cpu`
`memory`	Same as `requests.memory`

Resource Quota For Extended Resources

In addition to the resources mentioned above, in release 1.10, quota support for extended resources is added.

As overcommit is not allowed for extended resources, it makes no sense to specify both `requests` and `limits` for the same extended resource in a quota. So for extended resources, only quota items with prefix `requests.` is allowed for now.

Take the GPU resource as an example, if the resource name is `nvidia.com/gpu`, and you want to limit the total number of GPUs requested in a namespace to 4, you can define a quota as follows:

- `requests.nvidia.com/gpu: 4`

See Viewing and Setting Quotas for more detail information.

Storage Resource Quota

You can limit the total sum of storage resources that can be requested in a given namespace.

In addition, you can limit consumption of storage resources based on associated storage-class.

Resource Name	Description
`requests.storage`	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
`persistentvolumeclaims`	The total number of PersistentVolumeClaims that can exist in the namespace.
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	Across all persistent volume claims associated with the `<storage-class-name>`, the sum of storage requests cannot exceed this value.
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.

For example, if an operator wants to quota storage with `gold` storage class separate from `bronze` storage class, the operator can define a quota as follows:

- `gold.storageclass.storage.k8s.io/requests.storage: 500Gi`
- `bronze.storageclass.storage.k8s.io/requests.storage: 100Gi`

In release 1.8, quota support for local ephemeral storage is added as an alpha feature:

Resource Name	Description
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.
`ephemeral-storage`	Same as `requests.ephemeral-storage`.

Object Count Quota

You can set quota for the total number of certain resources of all standard, namespaced resource types using the following syntax:

- `count/<resource>.<group>` for resources from non-core groups
- `count/<resource>` for resources from the core group

Here is an example set of resources users may want to put under object count quota:

- `count/persistentvolumeclaims`
- `count/services`
- `count/secrets`
- `count/configmaps`
- `count/replicationcontrollers`
- `count/deployments.apps`
- `count/replicasets.apps`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`

The same syntax can be used for custom resources. For example, to create a quota on a `widgets` custom resource in the `example.com` API group, use `count/widgets.example.com`.

When using `count/*` resource quota, an object is charged against the quota if it exists in server storage. These types of quotas are useful to protect against exhaustion of storage resources. For example, you may want to limit the number of Secrets in a server given their large size. Too many Secrets in a cluster can actually prevent servers and controllers from starting. You can set a quota for Jobs to protect against a poorly configured CronJob. CronJobs that create too many Jobs in a namespace can lead to a denial of service.

It is also possible to do generic object count quota on a limited set of resources. The following types are supported:

Resource Name	Description
`configmaps`	The total number of ConfigMaps that can exist in the namespace.

Resource Name	Description
`persistentvolumeclaims`	The total number of PersistentVolumeClaims that can exist in the namespace.
`pods`	The total number of Pods in a non-terminal state that can exist in the namespace. A pod is in a terminal state if `.status.phase in (Failed, Succeeded)` is true.
`replicationcontrollers`	The total number of ReplicationControllers that can exist in the namespace.
`resourcequotas`	The total number of ResourceQuotas that can exist in the namespace.
`services`	The total number of Services that can exist in the namespace.
`services.loadbalancers`	The total number of Services of type `LoadBalancer` that can exist in the namespace.
`services.nodeports`	The total number of Services of type `NodePort` that can exist in the namespace.
`secrets`	The total number of Secrets that can exist in the namespace.

For example, `pods` quota counts and enforces a maximum on the number of `pods` created in a single namespace that are not terminal. You might want to set a `pods` quota on a namespace to avoid the case where a user creates many small pods and exhausts the cluster's supply of Pod IPs.

Quota Scopes

Each quota can have an associated set of `scopes`. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to the quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Scope	Description
`Terminating`	Match pods where `.spec.activeDeadlineSeconds >= 0`
`NotTerminating`	Match pods where `.spec.activeDeadlineSeconds is nil`
`BestEffort`	Match pods that have best effort quality of service.
`NotBestEffort`	Match pods that do not have best effort quality of service.
`PriorityClass`	Match pods that references the specified priority class.

The `BestEffort` scope restricts a quota to tracking the following resource:

- `pods`

The `Terminating`, `NotTerminating`, `NotBestEffort` and `PriorityClass` scopes restrict a quota to tracking the following resources:

- `pods`
- `cpu`
- `memory`
- `requests.cpu`
- `requests.memory`
- `limits.cpu`
- `limits.memory`

Note that you cannot specify both the `Terminating` and the `NotTerminating` scopes in the same quota, and you cannot specify both the `BestEffort` and `NotBestEffort` scopes in the same quota either.

The `scopeSelector` supports the following values in the `operator` field:

- `In`
- `NotIn`
- `Exists`
- `DoesNotExist`

When using one of the following values as the `scopeName` when defining the `scopeSelector`, the `operator` must be `Exists`.

- `Terminating`
- `NotTerminating`
- `BestEffort`
- `NotBestEffort`

If the `operator` is `In` or `NotIn`, the `values` field must have at least one value. For example:

```
  scopeSelector:
    matchExpressions:
      - scopeName: PriorityClass
        operator: In
        values:
          - middle
```

If the `operator` is `Exists` or `DoesNotExist`, the `values` field must *NOT* be specified.

Resource Quota Per PriorityClass

FEATURE STATE: `Kubernetes v1.17 [stable]`

Pods can be created at a specific priority. You can control a pod's consumption of system resources based on a pod's priority, by using the `scopeSelector` field in the quota spec.

A quota is matched and consumed only if `scopeSelector` in the quota spec selects the pod.

When quota is scoped for priority class using `scopeSelector` field, quota object is restricted to track only following resources:

- `pods`
- `cpu`
- `memory`
- `ephemeral-storage`
- `limits.cpu`
- `limits.memory`
- `limits.ephemeral-storage`
- `requests.cpu`
- `requests.memory`
- `requests.ephemeral-storage`

This example creates a quota object and matches it with pods at specific priorities. The example works as follows:

- Pods in the cluster have one of the three priority classes, "low", "medium", "high".
- One quota object is created for each priority.

Save the following YAML to a file `quota.yml`.

```yaml
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "1000"
      memory: 200Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["high"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-medium
  spec:
    hard:
      cpu: "10"
      memory: 20Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["medium"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-low
  spec:
    hard:
      cpu: "5"
      memory: 10Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["low"]
```

Apply the YAML using `kubectl create`.

```
kubectl create –f ./quota.yml
```

```
resourcequota/pods-high created
resourcequota/pods-medium created
resourcequota/pods-low created
```

Verify that `Used` quota is `0` using `kubectl describe quota`.

```
kubectl describe quota
```

```
Name:       pods-high
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     1k
memory      0     200Gi
pods        0     10


Name:       pods-low
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     5
memory      0     10Gi
pods        0     10


Name:       pods-medium
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10
```

Create a pod with priority "high". Save the following YAML to a file `high-priority-pod.yml`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: high-priority
spec:
  containers:
  - name: high-priority
    image: ubuntu
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello; sleep 10;done"]
    resources:
      requests:
        memory: "10Gi"
        cpu: "500m"
      limits:
        memory: "10Gi"
        cpu: "500m"
  priorityClassName: high
```

Apply it with `kubectl create`.

```
kubectl create -f ./high-priority-pod.yml
```

Verify that "Used" stats for "high" priority quota, `pods-high`, has changed and that the other two quotas are unchanged.

```
kubectl describe quota
```

```
Name:       pods-high
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         500m  1k
memory      10Gi  200Gi
pods        1     10


Name:       pods-low
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     5
memory      0     10Gi
pods        0     10


Name:       pods-medium
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10
```

Requests compared to Limits

When allocating compute resources, each container may specify a request and a limit value for either CPU or memory. The quota can be configured to quota either value.

If the quota has a value specified for `requests.cpu` or `requests.memory`, then it requires that every incoming container makes an explicit request for those resources. If the quota has a value specified for `limits.cpu` or `limits.memory`, then it requires that every incoming container specifies an explicit limit for those resources.

Viewing and Setting Quotas

Kubectl supports creating, updating, and viewing quotas:

```
kubectl create namespace myspace
```

```
cat <<EOF > compute-resources.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.nvidia.com/gpu: 4
EOF
```

```
kubectl create -f ./compute-resources.yaml --namespace=myspace
```

```
cat <<EOF > object-counts.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    pods: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
EOF
```

```
kubectl create -f ./object-counts.yaml --namespace=myspace
```

```
kubectl get quota --namespace=myspace
```

```
NAME                    AGE
compute-resources       30s
object-counts           32s
```

```
kubectl describe quota compute-resources --namespace=myspace
```

```
Name:                    compute-resources
Namespace:               myspace
Resource                 Used  Hard
--------                 ----  ----
limits.cpu               0     2
limits.memory            0     2Gi
requests.cpu             0     1
requests.memory          0     1Gi
requests.nvidia.com/gpu  0     4
```

```
kubectl describe quota object-counts --namespace=myspace
```

```
Name:                   object-counts
Namespace:              myspace
Resource                Used    Hard
--------                ----    ----
configmaps              0       10
persistentvolumeclaims  0       4
pods                    0       4
replicationcontrollers  0       20
secrets                 1       10
services                0       10
services.loadbalancers  0       2
```

Kubectl also supports object count quota for all standard namespaced resources using the syntax `count/<resource>.<group>`:

```
kubectl create namespace myspace
```

```
kubectl create quota test --hard=count/deployments.apps=2,count/replicasets.apps=4,
```

```
kubectl create deployment nginx --image=nginx --namespace=myspace --replicas=2
```

```
kubectl describe quota --namespace=myspace
```

```
Name:                         test
Namespace:                    myspace
Resource                      Used  Hard
--------                      ----  ----
count/deployments.apps        1     2
count/pods                    2     3
count/replicasets.apps        1     4
count/secrets                 1     4
```

Quota and Cluster Capacity

ResourceQuotas are independent of the cluster capacity. They are expressed in absolute units. So, if you add nodes to your cluster, this does *not* automatically give each namespace the ability to consume more resources.

Sometimes more complex policies may be desired, such as:

- Proportionally divide total cluster resources among several teams.
- Allow each tenant to grow resource usage as needed, but have a generous limit to prevent accidental resource exhaustion.
- Detect demand from one namespace, add nodes, and increase quota.

Such policies could be implemented using `ResourceQuotas` as building blocks, by writing a "controller" that watches the quota usage and adjusts the quota hard limits of each namespace according to other signals.

Note that resource quota divides up aggregate cluster resources, but it creates no restrictions around nodes: pods from several namespaces may run on the same node.

Limit Priority Class consumption by default

It may be desired that pods at a particular priority, eg. "cluster-services", should be allowed in a namespace, if and only if, a matching quota object exists.

With this mechanism, operators are able to restrict usage of certain high priority classes to a limited number of namespaces and not every namespace will be able to consume these priority classes by default.

To enforce this, `kube-apiserver` flag `--admission-control-config-file` should be used to pass path to the following configuration file:

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: "ResourceQuota"
  configuration:
    apiVersion: apiserver.config.k8s.io/v1
```

```
    kind: ResourceQuotaConfiguration
    limitedResources:
    - resource: pods
      matchScopes:
      - scopeName: PriorityClass
        operator: In
        values: ["cluster-services"]
```

Then, create a resource quota object in the `kube-system` namespace:

policy/priority-class-resourcequota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-cluster-services
spec:
  scopeSelector:
    matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["cluster-services"]
```

```
$ kubectl apply -f https://k8s.io/examples/policy/priority-class-resourcequota.yaml
```

```
resourcequota/pods-cluster-services created
```

In this case, a pod creation will be allowed if:

1. the Pod's `priorityClassName` is not specified.
2. the Pod's `priorityClassName` is specified to a value other than `cluster-services`.
3. the Pod's `priorityClassName` is set to `cluster-services`, it is to be created in the `kube-system` namespace, and it has passed the resource quota check.

A Pod creation request is rejected if its `priorityClassName` is set to `cluster-services` and it is to be created in a namespace other than `kube-system`.

What's next

- See ResourceQuota design doc for more information.
- See a detailed example for how to use resource quota.
- Read Quota support for priority class design doc.
- See LimitedResources

3 - Pod Security Policies

FEATURE STATE: `Kubernetes v1.20 [beta]`

Pod Security Policies enable fine-grained authorization of pod creation and updates.

What is a Pod Security Policy?

A *Pod Security Policy* is a cluster-level resource that controls security sensitive aspects of the pod specification. The PodSecurityPolicy objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields. They allow an administrator to control the following:

Control Aspect	Field Names
Running of privileged containers	`privileged`
Usage of host namespaces	`hostPID`, `hostIPC`
Usage of host networking and ports	`hostNetwork`, `hostPorts`
Usage of volume types	`volumes`
Usage of the host filesystem	`allowedHostPaths`
Allow specific FlexVolume drivers	`allowedFlexVolumes`
Allocating an FSGroup that owns the pod's volumes	`fsGroup`
Requiring the use of a read only root file system	`readOnlyRootFilesystem`
The user and group IDs of the container	`runAsUser`, `runAsGroup`, `supplementalGroups`
Restricting escalation to root privileges	`allowPrivilegeEscalation`, `defaultAllowPrivilegeEscalation`
Linux capabilities	`defaultAddCapabilities`, `requiredDropCapabilities`, `allowedCapabilities`
The SELinux context of the container	`seLinux`
The Allowed Proc Mount types for the container	`allowedProcMountTypes`
The AppArmor profile used by containers	annotations
The seccomp profile used by containers	annotations
The sysctl profile used by containers	`forbiddenSysctls`, `allowedUnsafeSysctls`

Enabling Pod Security Policies

Pod security policy control is implemented as an optional (but recommended) admission controller. PodSecurityPolicies are enforced by enabling the admission controller, but doing so without authorizing any policies **will prevent any pods from being created** in the cluster.

Since the pod security policy API (`policy/v1beta1/podsecuritypolicy`) is enabled independently of the admission controller, for existing clusters it is recommended that policies are added and authorized before enabling the admission controller.

Authorizing Policies

When a PodSecurityPolicy resource is created, it does nothing. In order to use it, the requesting user or target pod's service account must be authorized to use the policy, by allowing the `use` verb on the policy.

Most Kubernetes pods are not created directly by users. Instead, they are typically created indirectly as part of a Deployment, ReplicaSet, or other templated controller via the controller manager. Granting the controller access to the policy would grant access for *all* pods created by that controller, so the preferred method for authorizing policies is to grant access to the pod's service account (see example).

Via RBAC

RBAC is a standard Kubernetes authorization mode, and can easily be used to authorize use of policies.

First, a `Role` or `ClusterRole` needs to grant access to `use` the desired policies. The rules to grant access look like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <role name>
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames:
  - <list of policies to authorize>
```

Then the `(Cluster)Role` is bound to the authorized user(s):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <binding name>
roleRef:
  kind: ClusterRole
  name: <role name>
  apiGroup: rbac.authorization.k8s.io
subjects:
# Authorize specific service accounts:
- kind: ServiceAccount
  name: <authorized service account name>
  namespace: <authorized pod namespace>
# Authorize specific users (not recommended):
- kind: User
  apiGroup: rbac.authorization.k8s.io
  name: <authorized user name>
```

If a `RoleBinding` (not a `ClusterRoleBinding`) is used, it will only grant usage for pods being run in the same namespace as the binding. This can be paired with system groups to grant access to all pods run in the namespace:

```
# Authorize all service accounts in a namespace:
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:serviceaccounts
# Or equivalently, all authenticated users in a namespace:
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:authenticated
```

For more examples of RBAC bindings, see Role Binding Examples. For a complete example of authorizing a PodSecurityPolicy, see below.

Troubleshooting

- The controller manager must be run against the secured API port and must not have superuser permissions. See Controlling Access to the Kubernetes API to learn about API server access controls.
  If the controller manager connected through the trusted API port (also known as the `localhost` listener), requests would bypass authentication and authorization modules; all PodSecurityPolicy objects would be allowed, and users would be able to create grant themselves the ability to create privileged containers.

  For more details on configuring controller manager authorization, see Controller Roles.

Policy Order

In addition to restricting pod creation and update, pod security policies can also be used to provide default values for many of the fields that it controls. When multiple policies are available, the pod security policy controller selects policies according to the following criteria:

1. PodSecurityPolicies which allow the pod as-is, without changing defaults or mutating the pod, are preferred. The order of these non-mutating PodSecurityPolicies doesn't matter.
2. If the pod must be defaulted or mutated, the first PodSecurityPolicy (ordered by name) to allow the pod is selected.

> **Note:** During update operations (during which mutations to pod specs are disallowed) only non-mutating PodSecurityPolicies are used to validate the pod.

Example

This example assumes you have a running cluster with the PodSecurityPolicy admission controller enabled and you have cluster admin privileges.

Set up

Set up a namespace and a service account to act as for this example. We'll use this service account to mock a non-admin user.

```
kubectl create namespace psp-example
kubectl create serviceaccount -n psp-example fake-user
kubectl create rolebinding -n psp-example fake-editor --clusterrole=edit --servicea
```

To make it clear which user we're acting as and save some typing, create 2 aliases:

```
alias kubectl-admin='kubectl -n psp-example'
alias kubectl-user='kubectl --as=system:serviceaccount:psp-example:fake-user -n psp-
```

Create a policy and a pod

Define the example PodSecurityPolicy object in a file. This is a policy that prevents the creation of privileged pods. The name of a PodSecurityPolicy object must be a valid DNS subdomain name.

policy/example-psp.yaml

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false  # Don't allow privileged pods!
  # The rest fills in some required fields.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

And create it with kubectl:

```
kubectl-admin create -f example-psp.yaml
```

Now, as the unprivileged user, try to create a simple pod:

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
```

The output is similar to this:

```
Error from server (Forbidden): error when creating "STDIN": pods "pause" is forbidde
```

What happened? Although the PodSecurityPolicy was created, neither the pod's service account nor `fake-user` have permission to use the new policy:

```
kubectl-user auth can-i use podsecuritypolicy/example
no
```

Create the rolebinding to grant `fake-user` the `use` verb on the example policy:

> **Note:** This is not the recommended way! See the next section for the preferred approach.

```
kubectl-admin create role psp:unprivileged \
    --verb=use \
    --resource=podsecuritypolicy \
    --resource-name=example
role "psp:unprivileged" created

kubectl-admin create rolebinding fake-user:psp:unprivileged \
    --role=psp:unprivileged \
    --serviceaccount=psp-example:fake-user
rolebinding "fake-user:psp:unprivileged" created

kubectl-user auth can-i use podsecuritypolicy/example
yes
```

Now retry creating the pod:

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
```

The output is similar to this

```
pod "pause" created
```

It works as expected! But any attempts to create a privileged pod should still be denied:

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: privileged
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
```

The output is similar to this:

```
Error from server (Forbidden): error when creating "STDIN": pods "privileged" is for
```

Delete the pod before moving on:

```
kubectl-user delete pod pause
```

Run another pod

Let's try that again, slightly differently:

```
kubectl-user create deployment pause --image=k8s.gcr.io/pause
deployment "pause" created

kubectl-user get pods
No resources found.

kubectl-user get events | head -n 2
LASTSEEN   FIRSTSEEN   COUNT     NAME              KIND         SUBOBJECT
1m         2m          15        pause-7774d79b5   ReplicaSet
```

What happened? We already bound the `psp:unprivileged` role for our `fake-user`, why are we getting the error `Error creating: pods "pause-7774d79b5-" is forbidden: no providers available to validate pod request`? The answer lies in the source - `replicaset-controller`. Fake-user successfully created the deployment (which successfully created a replicaset), but when the replicaset went to create the pod it was not authorized to use the example podsecuritypolicy.

In order to fix this, bind the `psp:unprivileged` role to the pod's service account instead. In this case (since we didn't specify it) the service account is `default`:

```
kubectl-admin create rolebinding default:psp:unprivileged \
    --role=psp:unprivileged \
    --serviceaccount=psp-example:default
rolebinding "default:psp:unprivileged" created
```

Now if you give it a minute to retry, the replicaset-controller should eventually succeed in creating the pod:

```
kubectl-user get pods --watch
NAME                    READY     STATUS    RESTARTS   AGE
pause-7774d79b5-qrgcb   0/1       Pending   0          1s
pause-7774d79b5-qrgcb   0/1       Pending   0          1s
pause-7774d79b5-qrgcb   0/1       ContainerCreating   0          1s
pause-7774d79b5-qrgcb   1/1       Running   0          2s
```

Clean up

Delete the namespace to clean up most of the example resources:

```
kubectl-admin delete ns psp-example
namespace "psp-example" deleted
```

Note that `PodSecurityPolicy` resources are not namespaced, and must be cleaned up separately:

```
```

```
kubectl-admin delete psp example
podsecuritypolicy "example" deleted
```

Example Policies

This is the least restrictive policy you can create, equivalent to not using the pod security policy admission controller:

policy/privileged-psp.yaml

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'
```

This is an example of a restrictive policy that requires users to run as an unprivileged user, blocks possible escalations to root, and requires use of several security mechanisms.

policy/restricted-psp.yaml

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default,runti
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
```

```
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor rather than SELinux.
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

See Pod Security Standards for more examples.

Policy Reference

Privileged

Privileged - determines if any container in a pod can enable privileged mode. By default a container is not allowed to access any devices on the host, but a "privileged" container is given access to all devices on the host. This allows the container nearly all the same access as processes running on the host. This is useful for containers that want to use linux capabilities like manipulating the network stack and accessing devices.

Host namespaces

HostPID - Controls whether the pod containers can share the host process ID namespace. Note that when paired with ptrace this can be used to escalate privileges outside of the container (ptrace is forbidden by default).

HostIPC - Controls whether the pod containers can share the host IPC namespace.

HostNetwork - Controls whether the pod may use the node network namespace. Doing so gives the pod access to the loopback device, services listening on localhost, and could be used to snoop on network activity of other pods on the same node.

HostPorts - Provides a list of ranges of allowable ports in the host network namespace. Defined as a list of `HostPortRange`, with `min` (inclusive) and `max` (inclusive). Defaults to no allowed host ports.

Volumes and file systems

Volumes - Provides a list of allowed volume types. The allowable values correspond to the volume sources that are defined when creating a volume. For the complete list of volume types, see Types of Volumes. Additionally, `*` may be used to allow all volume types.

The **recommended minimum set** of allowed volumes for new PSPs are:

- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- secret
- projected

> **Warning:** PodSecurityPolicy does not limit the types of `PersistentVolume` objects that may be referenced by a `PersistentVolumeClaim`, and hostPath type `PersistentVolumes` do not support read-only access mode. Only trusted users should be granted permission to create `PersistentVolume` objects.

FSGroup - Controls the supplemental group applied to some volumes.

- *MustRunAs* - Requires at least one `range` to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MayRunAs* - Requires at least one `range` to be specified. Allows `FSGroups` to be left unset without providing a default. Validates against all ranges if `FSGroups` is set.
- *RunAsAny* - No default provided. Allows any `fsGroup` ID to be specified.

AllowedHostPaths - This specifies a list of host paths that are allowed to be used by hostPath volumes. An empty list means there is no restriction on host paths used. This is defined as a list of objects with a single `pathPrefix` field, which allows hostPath volumes to mount a path that begins with an allowed prefix, and a `readOnly` field indicating it must be mounted read-only. For example:

```
allowedHostPaths:
  # This allows "/foo", "/foo/", "/foo/bar" etc., but
  # disallows "/fool", "/etc/foo" etc.
  # "/foo/../" is never valid.
  - pathPrefix: "/foo"
    readOnly: true # only allow read-only mounts
```

> **Warning:**
> There are many ways a container with unrestricted access to the host filesystem can escalate privileges, including reading data from other containers, and abusing the credentials of system services, such as Kubelet.
>
> Writeable hostPath directory volumes allow containers to write to the filesystem in ways that let them traverse the host filesystem outside the `pathPrefix`. `readOnly: true`, available in Kubernetes 1.11+, must be used on **all** `allowedHostPaths` to effectively limit access to the specified `pathPrefix`.

ReadOnlyRootFilesystem - Requires that containers must run with a read-only root filesystem (i.e. no writable layer).

FlexVolume drivers

This specifies a list of FlexVolume drivers that are allowed to be used by flexvolume. An empty list or nil means there is no restriction on the drivers. Please make sure `volumes` field contains the `flexVolume` volume type; no FlexVolume driver is allowed otherwise.

For example:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: allow-flex-volumes
spec:
```

```
  # ... other spec fields
  volumes:
    - flexVolume
  allowedFlexVolumes:
    - driver: example/lvm
    - driver: example/cifs
```

Users and groups

RunAsUser - Controls which user ID the containers are run with.

- *MustRunAs* - Requires at least one `range` to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MustRunAsNonRoot* - Requires that the pod be submitted with a non-zero `runAsUser` or have the `USER` directive defined (using a numeric UID) in the image. Pods which have specified neither `runAsNonRoot` nor `runAsUser` settings will be mutated to set `runAsNonRoot=true`, thus requiring a defined non-zero numeric `USER` directive in the container. No default provided. Setting `allowPrivilegeEscalation=false` is strongly recommended with this strategy.
- *RunAsAny* - No default provided. Allows any `runAsUser` to be specified.

RunAsGroup - Controls which primary group ID the containers are run with.

- *MustRunAs* - Requires at least one `range` to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MayRunAs* - Does not require that RunAsGroup be specified. However, when RunAsGroup is specified, they have to fall in the defined range.
- *RunAsAny* - No default provided. Allows any `runAsGroup` to be specified.

SupplementalGroups - Controls which group IDs containers add.

- *MustRunAs* - Requires at least one `range` to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MayRunAs* - Requires at least one `range` to be specified. Allows `supplementalGroups` to be left unset without providing a default. Validates against all ranges if `supplementalGroups` is set.
- *RunAsAny* - No default provided. Allows any `supplementalGroups` to be specified.

Privilege Escalation

These options control the `allowPrivilegeEscalation` container option. This bool directly controls whether the `no_new_privs` flag gets set on the container process. This flag will prevent `setuid` binaries from changing the effective user ID, and prevent files from enabling extra capabilities (e.g. it will prevent the use of the `ping` tool). This behavior is required to effectively enforce `MustRunAsNonRoot`.

AllowPrivilegeEscalation - Gates whether or not a user is allowed to set the security context of a container to `allowPrivilegeEscalation=true`. This defaults to allowed so as to not break setuid binaries. Setting it to `false` ensures that no child process of a container can gain more privileges than its parent.

DefaultAllowPrivilegeEscalation - Sets the default for the `allowPrivilegeEscalation` option. The default behavior without this is to allow privilege escalation so as to not break setuid binaries. If that behavior is not desired, this field can be used to default to disallow, while still permitting pods to request `allowPrivilegeEscalation` explicitly.

Capabilities

Linux capabilities provide a finer grained breakdown of the privileges traditionally associated with the superuser. Some of these capabilities can be used to escalate privileges or for container breakout, and may be restricted by the PodSecurityPolicy. For more details on Linux capabilities, see capabilities(7).

The following fields take a list of capabilities, specified as the capability name in ALL_CAPS without the `CAP_` prefix.

AllowedCapabilities - Provides a list of capabilities that are allowed to be added to a container. The default set of capabilities are implicitly allowed. The empty set means that no additional capabilities may be added beyond the default set. `*` can be used to allow all capabilities.

RequiredDropCapabilities - The capabilities which must be dropped from containers. These capabilities are removed from the default set, and must not be added. Capabilities listed in `RequiredDropCapabilities` must not be included in `AllowedCapabilities` or `DefaultAddCapabilities`.

DefaultAddCapabilities - The capabilities which are added to containers by default, in addition to the runtime defaults. See the Docker documentation for the default list of capabilities when using the Docker runtime.

SELinux

- *MustRunAs* - Requires `seLinuxOptions` to be configured. Uses `seLinuxOptions` as the default. Validates against `seLinuxOptions`.
- *RunAsAny* - No default provided. Allows any `seLinuxOptions` to be specified.

AllowedProcMountTypes

`allowedProcMountTypes` is a list of allowed ProcMountTypes. Empty or nil indicates that only the `DefaultProcMountType` may be used.

`DefaultProcMount` uses the container runtime defaults for readonly and masked paths for /proc. Most container runtimes mask certain paths in /proc to avoid accidental security exposure of special devices or information. This is denoted as the string `Default`.

The only other ProcMountType is `UnmaskedProcMount`, which bypasses the default masking behavior of the container runtime and ensures the newly created /proc the container stays intact with no modifications. This is denoted as the string `Unmasked`.

AppArmor

Controlled via annotations on the PodSecurityPolicy. Refer to the AppArmor documentation.

Seccomp

As of Kubernetes v1.19, you can use the `seccompProfile` field in the `securityContext` of Pods or containers to control use of seccomp profiles. In prior versions, seccomp was controlled by adding annotations to a Pod. The same PodSecurityPolicies can be used with either version to enforce how these fields or annotations are applied.

seccomp.security.alpha.kubernetes.io/defaultProfileName - Annotation that specifies the default seccomp profile to apply to containers. Possible values are:

- `unconfined` - Seccomp is not applied to the container processes (this is the default in Kubernetes), if no alternative is provided.
- `runtime/default` - The default container runtime profile is used.
- `docker/default` - The Docker default seccomp profile is used. Deprecated as of Kubernetes 1.11. Use `runtime/default` instead.
- `localhost/<path>` - Specify a profile as a file on the node located at `<seccomp_root>/<path>`, where `<seccomp_root>` is defined via the `--seccomp-profile-root` flag on the Kubelet. If the `--seccomp-profile-root` flag is not defined, the default path will be used, which is `<root-dir>/seccomp` where `<root-dir>` is specified by the `--root-dir` flag.

> **Note:** The `--seccomp-profile-root` flag is deprecated since Kubernetes v1.19. Users are encouraged to use the default path.

seccomp.security.alpha.kubernetes.io/allowedProfileNames - Annotation that specifies which values are allowed for the pod seccomp annotations. Specified as a comma-delimited list of allowed values. Possible values are those listed above, plus `*` to allow all profiles. Absence of this annotation means that the default cannot be changed.

Sysctl

By default, all safe sysctls are allowed.

- `forbiddenSysctls` - excludes specific sysctls. You can forbid a combination of safe and unsafe sysctls in the list. To forbid setting any sysctls, use `*` on its own.
- `allowedUnsafeSysctls` - allows specific sysctls that had been disallowed by the default list, so long as these are not listed in `forbiddenSysctls`.

Refer to the Sysctl documentation.

What's next

- See Pod Security Standards for policy recommendations.
- Refer to Pod Security Policy Reference for the api details.

4 - Process ID Limits And Reservations

FEATURE STATE: `Kubernetes v1.20 [stable]`

Kubernetes allow you to limit the number of process IDs (PIDs) that a Pod can use. You can also reserve a number of allocatable PIDs for each node for use by the operating system and daemons (rather than by Pods).

Process IDs (PIDs) are a fundamental resource on nodes. It is trivial to hit the task limit without hitting any other resource limits, which can then cause instability to a host machine.

Cluster administrators require mechanisms to ensure that Pods running in the cluster cannot induce PID exhaustion that prevents host daemons (such as the kubelet or kube-proxy, and potentially also the container runtime) from running. In addition, it is important to ensure that PIDs are limited among Pods in order to ensure they have limited impact on other workloads on the same node.

> **Note:** On certain Linux installations, the operating system sets the PIDs limit to a low default, such as `32768`. Consider raising the value of `/proc/sys/kernel/pid_max`.

You can configure a kubelet to limit the number of PIDs a given Pod can consume. For example, if your node's host OS is set to use a maximum of `262144` PIDs and expect to host less than `250` Pods, one can give each Pod a budget of `1000` PIDs to prevent using up that node's overall number of available PIDs. If the admin wants to overcommit PIDs similar to CPU or memory, they may do so as well with some additional risks. Either way, a single Pod will not be able to bring the whole machine down. This kind of resource limiting helps to prevent simple fork bombs from affecting operation of an entire cluster.

Per-Pod PID limiting allows administrators to protect one Pod from another, but does not ensure that all Pods scheduled onto that host are unable to impact the node overall. Per-Pod limiting also does not protect the node agents themselves from PID exhaustion.

You can also reserve an amount of PIDs for node overhead, separate from the allocation to Pods. This is similar to how you can reserve CPU, memory, or other resources for use by the operating system and other facilities outside of Pods and their containers.

PID limiting is a an important sibling to compute resource requests and limits. However, you specify it in a different way: rather than defining a Pod's resource limit in the `.spec` for a Pod, you configure the limit as a setting on the kubelet. Pod-defined PID limits are not currently supported.

> **Caution:** This means that the limit that applies to a Pod may be different depending on where the Pod is scheduled. To make things simple, it's easiest if all Nodes use the same PID resource limits and reservations.

Node PID limits

Kubernetes allows you to reserve a number of process IDs for the system use. To configure the reservation, use the parameter `pid=<number>` in the `--system-reserved` and `--kube-reserved` command line options to the kubelet. The value you specified declares that the specified number of process IDs will be reserved for the system as a whole and for Kubernetes system daemons respectively.

> **Note:** Before Kubernetes version 1.20, PID resource limiting with Node-level reservations required enabling the feature gate `SupportNodePidsLimit` to work.

Pod PID limits

Kubernetes allows you to limit the number of processes running in a Pod. You specify this limit at the node level, rather than configuring it as a resource limit for a particular Pod. Each Node can have a different PID limit.

To configure the limit, you can specify the command line parameter `--pod-max-pids` to the kubelet, or set `PodPidsLimit` in the kubelet configuration file.

> **Note:** Before Kubernetes version 1.20, PID resource limiting for Pods required enabling the feature gate `SupportPodPidsLimit` to work.

PID based eviction

You can configure kubelet to start terminating a Pod when it is misbehaving and consuming abnormal amount of resources. This feature is called eviction. You can Configure Out of Resource Handling for various eviction signals. Use `pid.available` eviction signal to configure the threshold for number of PIDs used by Pod. You can set soft and hard eviction policies. However, even with the hard eviction policy, if the number of PIDs growing very fast, node can still get into unstable state by hitting the node PIDs limit. Eviction signal value is calculated periodically and does NOT enforce the limit.

PID limiting - per Pod and per Node sets the hard limit. Once the limit is hit, workload will start experiencing failures when trying to get a new PID. It may or may not lead to rescheduling of a Pod, depending on how workload reacts on these failures and how liveleness and readiness probes are configured for the Pod. However, if limits were set correctly, you can guarantee that other Pods workload and system processes will not run out of PIDs when one Pod is misbehaving.

What's next

- Refer to the PID Limiting enhancement document for more information.
- For historical context, read Process ID Limiting for Stability Improvements in Kubernetes 1.14.
- Read Managing Resources for Containers.
- Learn how to Configure Out of Resource Handling.