Authorization

By Request attributes used in authorization

Archived: 2026-04-05 14:34:58 UTC

Details of Kubernetes authorization mechanisms and supported authorization modes.

Kubernetes authorization takes place following authentication. Usually, a client making a request must be authenticated (logged in) before its request can be allowed; however, Kubernetes also allows anonymous requests in some circumstances.

For an overview of how authorization fits into the wider context of API access control, read Controlling Access to the Kubernetes API.

Authorization verdicts

Kubernetes authorization of API requests takes place within the API server. The API server evaluates all of the request attributes against all policies, potentially also consulting external services, and then allows or denies the request.

All parts of an API request must be allowed by some authorization mechanism in order to proceed. In other words: access is denied by default.

Note:

Access controls and policies that depend on specific fields of specific kinds of objects are handled by admission controllers.

Kubernetes admission control happens after authorization has completed (and, therefore, only when the authorization decision was to allow the request).

When multiple authorization modules are configured, each is checked in sequence. If any authorizer *approves* or *denies* a request, that decision is immediately returned and no other authorizer is consulted. If all modules have *no opinion* on the request, then the request is denied. An overall deny verdict means that the API server rejects the request and responds with an HTTP 403 (Forbidden) status.

Kubernetes reviews only the following API request attributes:

- **user** - The `user` string provided during authentication.
- **group** - The list of group names to which the authenticated user belongs.
- **extra** - A map of arbitrary string keys to string values, provided by the authentication layer.
- **API** - Indicates whether the request is for an API resource.
- **Request path** - Path to miscellaneous non-resource endpoints like `/api` or `/healthz`.

- **API request verb** - API verbs like `get`, `list`, `create`, `update`, `patch`, `watch`, `delete`, and `deletecollection` are used for resource requests. To determine the request verb for a resource API endpoint, see request verbs and authorization.
- **HTTP request verb** - Lowercased HTTP methods like `get`, `post`, `put`, and `delete` are used for non-resource requests.
- **Resource** - The ID or name of the resource that is being accessed (for resource requests only) -- For resource requests using `get`, `update`, `patch`, and `delete` verbs, you must provide the resource name.
- **Subresource** - The subresource that is being accessed (for resource requests only).
- **Namespace** - The namespace of the object that is being accessed (for namespaced resource requests only).
- **API group** - The API Group being accessed (for resource requests only). An empty string designates the *core* API group.

Request verbs and authorization

Non-resource requests

Requests to endpoints other than `/api/v1/...` or `/apis/<group>/<version>/...` are considered *non-resource requests*, and use the lower-cased HTTP method of the request as the verb. For example, making a `GET` request using HTTP to endpoints such as `/api` or `/healthz` would use **get** as the verb.

Resource requests

To determine the request verb for a resource API endpoint, Kubernetes maps the HTTP verb used and considers whether or not the request acts on an individual resource or on a collection of resources:

HTTP verb	request verb
`POST`	**create**
`GET`, `HEAD`	**get** (for individual resources), **list** (for collections, including full object content), **watch** (for watching an individual resource or collection of resources)
`PUT`	**update**
`PATCH`	**patch**
`DELETE`	**delete** (for individual resources), **deletecollection** (for collections)

Caution:

+The **get**, **list** and **watch** verbs can all return the full details of a resource. In terms of access to the returned data they are equivalent. For example, **list** on `secrets` will reveal the **data** attributes of any returned resources.

Kubernetes sometimes checks authorization for additional permissions using specialized verbs. For example:

- Special cases of authentication
  - **impersonate** verb on `users`, `groups`, and `serviceaccounts` in the core API group, and the `userextras` in the `authentication.k8s.io` API group.
- Authorization of CertificateSigningRequests
  - **approve** verb for CertificateSigningRequests, and **update** for revisions to existing approvals
- RBAC
  - **bind** and **escalate** verbs on `roles` and `clusterroles` resources in the `rbac.authorization.k8s.io` API group.

Authorization context

Kubernetes expects attributes that are common to REST API requests. This means that Kubernetes authorization works with existing organization-wide or cloud-provider-wide access control systems which may handle other APIs besides the Kubernetes API.

Authorization modes

The Kubernetes API server may authorize a request using one of several authorization modes:

`AlwaysAllow`
: This mode allows all requests, which brings security risks. Use this authorization mode only if you do not require authorization for your API requests (for example, for testing).

`AlwaysDeny`
: This mode blocks all requests. Use this authorization mode only for testing.

`ABAC` (attribute-based access control)
: Kubernetes ABAC mode defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together. The policies can use any type of attributes (user attributes, resource attributes, object, environment attributes, etc).

`RBAC` (role-based access control)
: Kubernetes RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In this context, access is the ability of an individual user to perform a specific task, such as view, create, or modify a file.
In this mode, Kubernetes uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure permission policies through the Kubernetes API.

`Node`
: A special-purpose authorization mode that grants permissions to kubelets based on the pods they are scheduled to run. To learn more about the Node authorization mode, see Node Authorization.

`Webhook`
: Kubernetes webhook mode for authorization makes a synchronous HTTP callout, blocking the request until the remote HTTP service responds to the query.You can write your own software to handle the callout, or use solutions from the ecosystem.

The system:masters group

The `system:masters` group is a built-in Kubernetes group that grants unrestricted access to the API server. Any user assigned to this group has full cluster administrator privileges, bypassing any authorization restrictions imposed by the RBAC or Webhook mechanisms. Avoid adding users to this group. If you do need to grant a user cluster-admin rights, you can create a ClusterRoleBinding to the built-in `cluster-admin` ClusterRole.

Authorization mode configuration

You can configure the Kubernetes API server's authorizer chain using either a configuration file only or command line arguments.

You have to pick one of the two configuration approaches; setting both `--authorization-config` path and configuring an authorization webhook using the `--authorization-mode` and `--authorization-webhook-*` command line arguments is not allowed. If you try this, the API server reports an error message during startup, then exits immediately.

Configuring the API Server using an authorization config file

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default)

Kubernetes lets you configure authorization chains that can include multiple webhooks. The authorization items in that chain can have well-defined parameters that validate requests in a particular order, offering you fine-grained control, such as explicit Deny on failures.

The configuration file approach even allows you to specify CEL rules to pre-filter requests before they are dispatched to webhooks, helping you to prevent unnecessary invocations. The API server also automatically reloads the authorizer chain when the configuration file is modified.

You specify the path to the authorization configuration using the `--authorization-config` command line argument.

If you want to use command line arguments instead of a configuration file, that's also a valid and supported approach. Some authorization capabilities (for example: multiple webhooks, webhook failure policy, and pre-filter rules) are only available if you use an authorization configuration file.

Example configuration

```
 ---
#
# DO NOT USE THE CONFIG AS IS. THIS IS AN EXAMPLE.
#
apiVersion: apiserver.config.k8s.io/v1
kind: AuthorizationConfiguration
authorizers:
  - type: Webhook
    # Name used to describe the authorizer
    # This is explicitly used in monitoring machinery for metrics
```

```
# Note:
#   - Validation for this field is similar to how K8s labels are validated today.
# Required, with no default
name: webhook
webhook:
  # The duration to cache 'authorized' responses from the webhook
  # authorizer.
  # Same as setting `--authorization-webhook-cache-authorized-ttl` flag
  # Default: 5m0s
  authorizedTTL: 30s
  # If set to false, 'authorized' responses from the webhook are not cached
  # and the specified authorizedTTL is ignored/has no effect.
  # Same as setting `--authorization-webhook-cache-authorized-ttl` flag to `0`.
  # Note: Setting authorizedTTL to `0` results in its default value being used.
  # Default: true
  cacheAuthorizedRequests: true
  # The duration to cache 'unauthorized' responses from the webhook
  # authorizer.
  # Same as setting `--authorization-webhook-cache-unauthorized-ttl` flag
  # Default: 30s
  unauthorizedTTL: 30s
  # If set to false, 'unauthorized' responses from the webhook are not cached
  # and the specified unauthorizedTTL is ignored/has no effect.
  # Same as setting `--authorization-webhook-cache-unauthorized-ttl` flag to `0`.
  # Note: Setting unauthorizedTTL to `0` results in its default value being used.
  # Default: true
  cacheUnauthorizedRequests: true
  # Timeout for the webhook request
  # Maximum allowed is 30s.
  # Required, with no default.
  timeout: 3s
  # The API version of the authorization.k8s.io SubjectAccessReview to
  # send to and expect from the webhook.
  # Same as setting `--authorization-webhook-version` flag
  # Required, with no default
  # Valid values: v1beta1, v1
  subjectAccessReviewVersion: v1
  # MatchConditionSubjectAccessReviewVersion specifies the SubjectAccessReview
  # version the CEL expressions are evaluated against
  # Valid values: v1
  # Required, no default value
  matchConditionSubjectAccessReviewVersion: v1
  # Controls the authorization decision when a webhook request fails to
  # complete or returns a malformed response or errors evaluating
  # matchConditions.
  # Valid values:
  #   - NoOpinion: continue to subsequent authorizers to see if one of
```

```
      #     them allows the request
      #   - Deny: reject the request without consulting subsequent authorizers
      # Required, with no default.
      failurePolicy: Deny
      connectionInfo:
        # Controls how the webhook should communicate with the server.
        # Valid values:
        # - KubeConfigFile: use the file specified in kubeConfigFile to locate the
        #   server.
        # - InClusterConfig: use the in-cluster configuration to call the
        #   SubjectAccessReview API hosted by kube-apiserver. This mode is not
        #   allowed for kube-apiserver.
        type: KubeConfigFile
        # Path to KubeConfigFile for connection info
        # Required, if connectionInfo.Type is KubeConfigFile
        kubeConfigFile: /kube-system-authz-webhook.yaml
        # matchConditions is a list of conditions that must be met for a request to be sent to this
        # webhook. An empty list of matchConditions matches all requests.
        # There are a maximum of 64 match conditions allowed.
        #
        # The exact matching logic is (in order):
        #   1. If at least one matchCondition evaluates to FALSE, then the webhook is skipped.
        #   2. If ALL matchConditions evaluate to TRUE, then the webhook is called.
        #   3. If at least one matchCondition evaluates to an error (but none are FALSE):
        #      - If failurePolicy=Deny, then the webhook rejects the request
        #      - If failurePolicy=NoOpinion, then the error is ignored and the webhook is skipped
      matchConditions:
      # expression represents the expression which will be evaluated by CEL. Must evaluate to bool.
      # CEL expressions have access to the contents of the SubjectAccessReview in v1 version.
      # If version specified by subjectAccessReviewVersion in the request variable is v1beta1,
      # the contents would be converted to the v1 version before evaluating the CEL expression.
      #
      # Documentation on CEL: https://kubernetes.io/docs/reference/using-api/cel/
      #
      # only send resource requests to the webhook
      - expression: has(request.resourceAttributes)
      # only intercept requests to kube-system
      - expression: request.resourceAttributes.namespace == 'kube-system'
      # don't intercept requests from kube-system service accounts
      - expression: "!('system:serviceaccounts:kube-system' in request.groups)"
  - type: Node
    name: node
  - type: RBAC
    name: rbac
  - type: Webhook
    name: in-cluster-authorizer
    webhook:
```

```
    authorizedTTL: 5m
    unauthorizedTTL: 30s
    timeout: 3s
    subjectAccessReviewVersion: v1
    failurePolicy: NoOpinion
    connectionInfo:
      type: InClusterConfig
```

When configuring the authorizer chain using a configuration file, make sure all the control plane nodes have the same file contents. Take a note of the API server configuration when upgrading / downgrading your clusters. For example, if upgrading from Kubernetes 1.34 to Kubernetes 1.35, you would need to make sure the config file is in a format that Kubernetes 1.35 can understand, before you upgrade the cluster. If you downgrade to 1.34, you would need to set the configuration appropriately.

Authorization configuration and reloads

Kubernetes reloads the authorization configuration file when the API server observes a change to the file, and also on a 60 second schedule if no change events were observed.

Note:

You must ensure that all non-webhook authorizer types remain unchanged in the file on reload.

A reload **must not** add or remove Node or RBAC authorizers (they can be reordered, but cannot be added or removed).

Command line authorization mode configuration

You can use the following modes:

- `--authorization-mode=ABAC` (Attribute-based access control mode)
- `--authorization-mode=RBAC` (Role-based access control mode)
- `--authorization-mode=Node` (Node authorizer)
- `--authorization-mode=Webhook` (Webhook authorization mode)
- `--authorization-mode=AlwaysAllow` (always allows requests; carries security risks)
- `--authorization-mode=AlwaysDeny` (always denies requests)

You can choose more than one authorization mode; for example: `--authorization-mode=Node,RBAC,Webhook`

Kubernetes checks authorization modules based on the order that you specify them on the API server's command line, so an earlier module has higher priority to allow or deny a request.

You cannot combine the `--authorization-mode` command line argument with the `--authorization-config` command line argument used for configuring authorization using a local file.

For more information on command line arguments to the API server, read the `kube-apiserver` reference.

Privilege escalation via workload creation or edits

Users who can create/edit pods in a namespace, either directly or through an object that enables indirect workload management, may be able to escalate their privileges in that namespace. The potential routes to privilege escalation include Kubernetes API extensions and their associated controllers.

Caution:

As a cluster administrator, use caution when granting access to create or edit workloads. Some details of how these can be misused are documented in escalation paths.

Escalation paths

There are different ways that an attacker or untrustworthy user could gain additional privilege within a namespace, if you allow them to run arbitrary Pods in that namespace:

- Mounting arbitrary Secrets in that namespace
  - Can be used to access confidential information meant for other workloads
  - Can be used to obtain a more privileged ServiceAccount's service account token
- Using arbitrary ServiceAccounts in that namespace
  - Can perform Kubernetes API actions as another workload (impersonation)
  - Can perform any privileged actions that ServiceAccount has
- Mounting or using ConfigMaps meant for other workloads in that namespace
  - Can be used to obtain information meant for other workloads, such as database host names.
- Mounting volumes meant for other workloads in that namespace
  - Can be used to obtain information meant for other workloads, and change it.

Caution:

As a system administrator, you should be cautious when deploying CustomResourceDefinitions that let users make changes to the above areas. These may open privilege escalations paths. Consider the consequences of this kind of change when deciding on your authorization controls.

Checking API access

`kubectl` provides the `auth can-i` subcommand for quickly querying the API authorization layer. The command uses the `SelfSubjectAccessReview` API to determine if the current user can perform a given action, and works regardless of the authorization mode used.

```
kubectl auth can-i create deployments --namespace dev
```

The output is similar to this:

```
yes
```

```
kubectl auth can-i create deployments --namespace prod
```

The output is similar to this:

```
no
```

Administrators can combine this with user impersonation to determine what action other users can perform.

```
kubectl auth can-i list secrets --namespace dev --as dave
```

The output is similar to this:

```
no
```

Similarly, to check whether a ServiceAccount named `dev-sa` in Namespace `dev` can list Pods in the Namespace `target`:

```
kubectl auth can-i list pods \
   --namespace target \
   --as system:serviceaccount:dev:dev-sa
```

The output is similar to this:

```
yes
```

SelfSubjectAccessReview is part of the `authorization.k8s.io` API group, which exposes the API server authorization to external services. Other resources in this group include:

SubjectAccessReview
: Access review for any user, not only the current one. Useful for delegating authorization decisions to the API server. For example, the kubelet and extension API servers use this to determine user access to their own APIs.

LocalSubjectAccessReview
: Like SubjectAccessReview but restricted to a specific namespace.

SelfSubjectRulesReview
: A review which returns the set of actions a user can perform within a namespace. Useful for users to quickly summarize their own access, or for UIs to hide/show actions.

These APIs can be queried by creating normal Kubernetes resources, where the response `status` field of the returned object is the result of the query. For example:

```
 kubectl create -f - -o yaml << EOF
apiVersion: authorization.k8s.io/v1
kind: SelfSubjectAccessReview
spec:
  resourceAttributes:
    group: apps
    resource: deployments
    verb: create
    namespace: dev
EOF
```

The generated SelfSubjectAccessReview is similar to:

```
 apiVersion: authorization.k8s.io/v1
kind: SelfSubjectAccessReview
metadata:
  creationTimestamp: null
spec:
  resourceAttributes:
    group: apps
    resource: deployments
    namespace: dev
    verb: create
status:
  allowed: true
  denied: false
```

What's next

- To learn more about Authentication, see Authentication.
- For an overview, read Controlling Access to the Kubernetes API.
- To learn more about Admission Control, see Using Admission Controllers.
- Read more about Common Expression Language in Kubernetes.

Source: https://kubernetes.io/docs/reference/access-authn-authz/authorization/