Introduction to AuthX

Get introduced to authentication and authorization in Kubernetes.

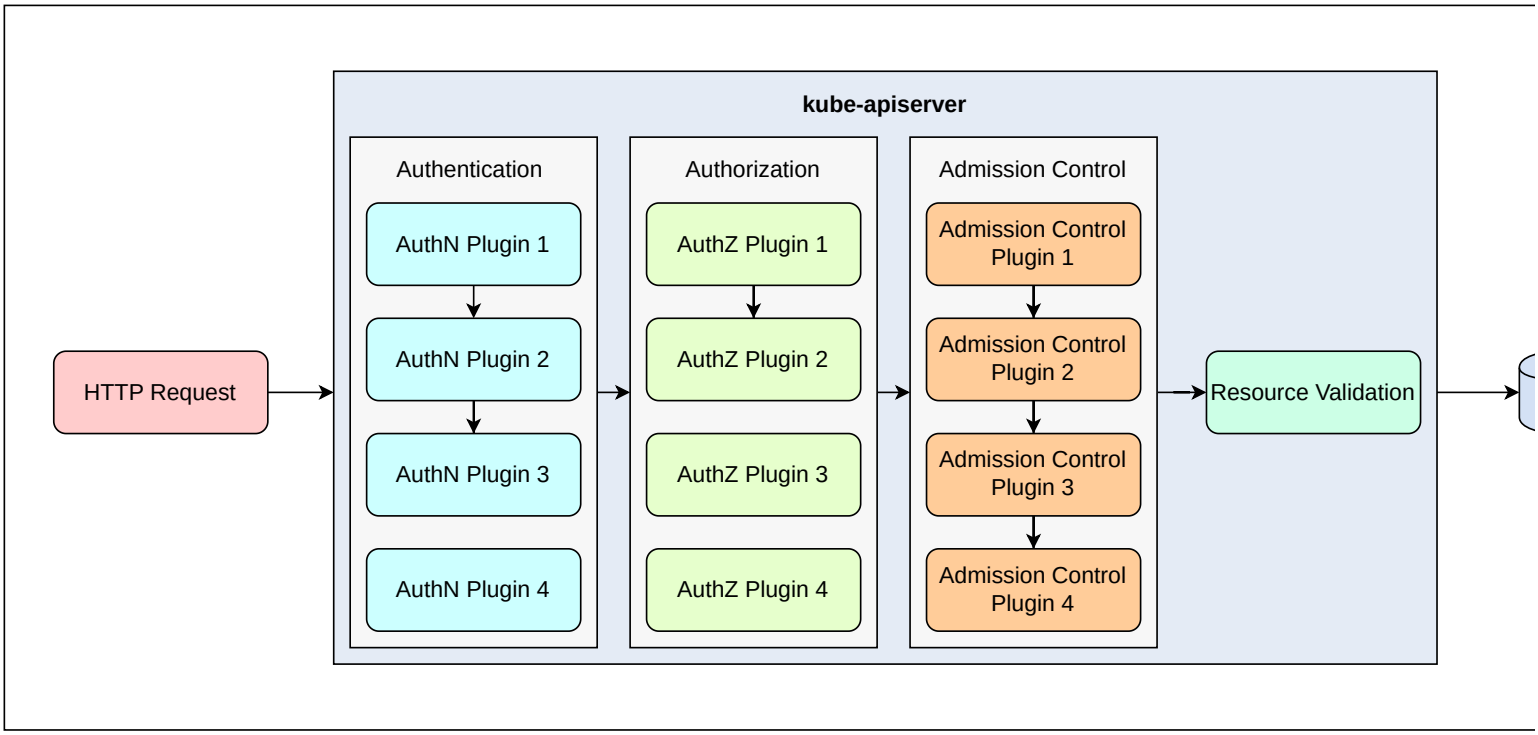
We'll cover the following

The security center of Kubernetes

The `kube-apiserver` is the heart of a Kubernetes cluster. All the system-level authentications (`AuthN`) and authorizations (`AuthZ`) are handled by it. `AuthX` usually refers to both `AuthN` and `AuthZ`. We can also say that the `kube-apiserver` is the security center of Kubernetes.

We could run the `kube-apiserver` with insecure settings, but that isn't suggested, especially in production environments. It's strongly suggested to enable transport layer security (TLS) between all the Kubernetes components. This helps improve the whole cluster's security.

Just as the graph below shows, all the requests that are being sent to the `kube-apiserver` need to pass through the authentication, authorization, and admission control stages, and then come to the final resource validation and persistent storing stages.



Authentication

In Kubernetes, the `kube-apiserver` needs to authenticate every received request to validate the user identity. Such a request may be sent out by either a user or a program. We can think of it as a login process.

The simplest authentication method is using a user name, password, and tokens. The `kube-apiserver` does provide the flag `--token-auth-file` to use a static file that stores all these credentials. However, this method isn't recommended in any production environment. Every time a user is added, updated, and deleted, we need to modify this file and restart the `kube-apiserver` in order for the changes to take effect. Maintaining such a static file is a dangerous and painstaking task. We need to update all the `kube-apiserver` replicas. Inconsistent settings may throw the authentication process into chaos.

Instead of the static tokens above, Kubernetes provides built-in Bootstrap and ServiceAccount tokens. These two kinds of tokens are managed by Kubernetes directly. We can mount the ServiceAccount tokens into Pods so that in-cluster processes can use it to talk to the `kube-apiserver`. It's easy to bind permissions to a ServiceAccount by `RoleBinding` and `ClusterRoleBinding`.

The x509 method is another widely used authentication method in Kubernetes. Any request that presents a valid certificate signed by the CA (certificate authority) of our cluster is considered to be valid. Every certificate has the attribute `subject`, where the `kube-apiserver` gets user info. The `CN` (common name) field can be used for the user name. The group name that the user belongs to can be set in the `O` (organization name) field. For example, if we create the certificate with `Subject: O=system:nodes, CN=system:node:master`, this certificate will be interpreted to the user `system:node:master` of the group `system:nodes` by the `kube-apiserver`.

If we've got self-hosted authentication services like LDAP or Kerberos, the authentication methods above don't meet our requirements. However, Kubernetes does provide us ways to integrate. Using the authentication webhook is an excellent option for such scenarios. The webhook allows us to generate and use user-defined tokens.

The authentication methods given above will be invoked one by one. If none of them can verify the identity of the user, the request is unauthorized and a status code of `401` is returned. After passing the authentication stage, the `kube-apiserver` will pass the user info (including user name, group name, etc.) down to the following authorization stage.

Authorization

Authorization comes after authentication has passed. Authorization plays an important role in rejecting any requests with wrong user privileges. With authorization mechanisms, we can define who has the

privileges to access different resources in our cluster.

Attribute-based access control (ABAC) is the simplest authorization method in Kubernetes. It's defined with the flag `--authorization-policy-file` in the `kube-apiserver`. The file contains a group of user policies with combined attributes. However, this method isn't recommended in any production environment because, as we've said before, it's a dangerous and painstaking task to maintain such static files.

Role-based access control (RBAC) is the most commonly used built-in authorization method in Kubernetes. It's a powerful method that can help define fine-grained rules on who has access to a resource.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: my-apps-creation
rules:
  - apiGroups:
      - "apps.mygroup.io"
    resources:
      - myapps
    verbs:
      - get
      - create
      - list
      - delete
      - update
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-apps-creation
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: my-apps-creation
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: Group
    name: system:mygroup:apps
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: Alice
```

ClusterRole and ClusterRoleBinding of myapps

The code snippet above is an example RBAC rule that defines the users who can get, create, and list `myapps` in the cluster.

`ClusterRoleBinding` works cluster-wide, while `RoleBinding` operates in the context of a namespace. RBAC works with groups, users, and service accounts. It's incredibly easy to use.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: create-deployments
  namespace: my-testing-ns
```

```
rules:
  - apiGroups: [""]
    resources: ["deployments"]
    verbs: ["create", "get", "list"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: create-deployments
  namespace: my-testing-ns
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: create-deployments
subjects:
  - kind: ServiceAccount
    name: my-agent
    namespace: my-testing-ns
```

Role and RoleBinding on creating/listing deployments

Similar to authentication, Kubernetes provides a webhook for authorization, where we can integrate with a self-hosted system. If we follow the guidelines, we could write in any language.

Introducing the webhook doesn't mean we're throwing out existing RBAC rules. All the authorization plugins are bundled as a union chain and will be invoked one by one. If none of them can determine the user privilege, the request is rejected, and a status code of `403` is returned. After passing the authorization stage, the `kube-apiserver` will pass the request down to the admission stage.

Summary

Kubernetes provides several built-in `AuthN` and `AuthZ` methods. Kubernetes also provides extension points—in other words, the authorization webhook—which allow calling out to user-provided code to make an authentication or authorization decision. As a result, we can bind our Kubernetes cluster to any custom authentication or authorization method or user-management system. The webhook feature of Kubernetes offers an excellent way to integrate with self-hosted authentication or authorization systems.

Kubernetes is flexible, so we can choose the authentication or authorization mechanism that best suits our requirements in our organization.

Back

Quiz on Kubernetes Architecture

Next