Kubernetes from Basic to Advanced

kubernetes

Session #08
Security and Scheduling

kubernetes

Session Contents

- RBAC
- Network Policies
- Affinity
- Taints & Tolerations

kubernetes

RBAC

Motivation

- As any system, you want to have a control about who access your resources and which type of action can perform
- Your cluster can be reached by human users and services
- Human users are the operator who access the cluster using API Server (through kubectl)
- Services are your pods running inside your cluster
- Role-based Access Control (RBAC) is the implementation of authn/authz in Kubernetes

kubernetes

RBAC

- Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization
- RBAC authorization uses the **`rbac.authorization.k8s.io`** API group to drive authorization decisions, allowing you to dynamically configure policies through the Kubernetes API
- By default, RBAC is not enabled on a Kubernetes cluster. You need to explicitly enable it

kubernetes

RBAC Objects

- RBAC API declares 4 objects: Role, ClusterRole, RoleBinding and ClusterRoleBinding
- Role and ClusterRole contains rules that represent a set of permissions that are always additive (there are no "deny" rules)
- RoleBinding and ClusterRoleBinding grants the permissions defined in a role to a user or set of users

kubernetes

Role and ClusterRole

- Role and ClusterRole contains rules that represent a set of permissions that are always additive (there are no "deny" rules)
- A Role always sets permissions within a particular namespace; ClusterRole, by contrast, is a non-namespaced resource
- You can use a ClusterRole to:
  - Define permissions on namespaced resources and be granted access within individual namespace(s)
  - Define permissions on namespaced resources and be granted access across all namespaces
  - Define permissions on cluster-scoped resources

kubernetes

Role and ClusterRole

- Role and ClusterRole contains rules that represent a set of permissions that are always additive (there are no "deny" rules)
- A Role always sets permissions within a particular namespace; ClusterRole, by contrast, is a non-namespaced resource
- You can use a ClusterRole to:
  - Define permissions on namespaced resources and be granted access within individual namespace(s)
  - Define permissions on namespaced resources and be granted access across all namespaces
  - Define permissions on cluster-scoped resources

kubernetes

Role and ClusterRole Manifest

- When creating Role or ClusterRole manifest you create a list of rules
- For each rules you need to define
  - **`apiGroups`**
  - **`resources`**
  - **`verbs`**
  - Optionally, **`resourceNames`**
- To get a list of available values use
  - **`kubectl api-resources -o wide`**

kubernetes

Role and ClusterRole

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: deployments-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["replicasets"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["deployments"
  verbs: ["get", "watch", "list"]
```

kubernetes

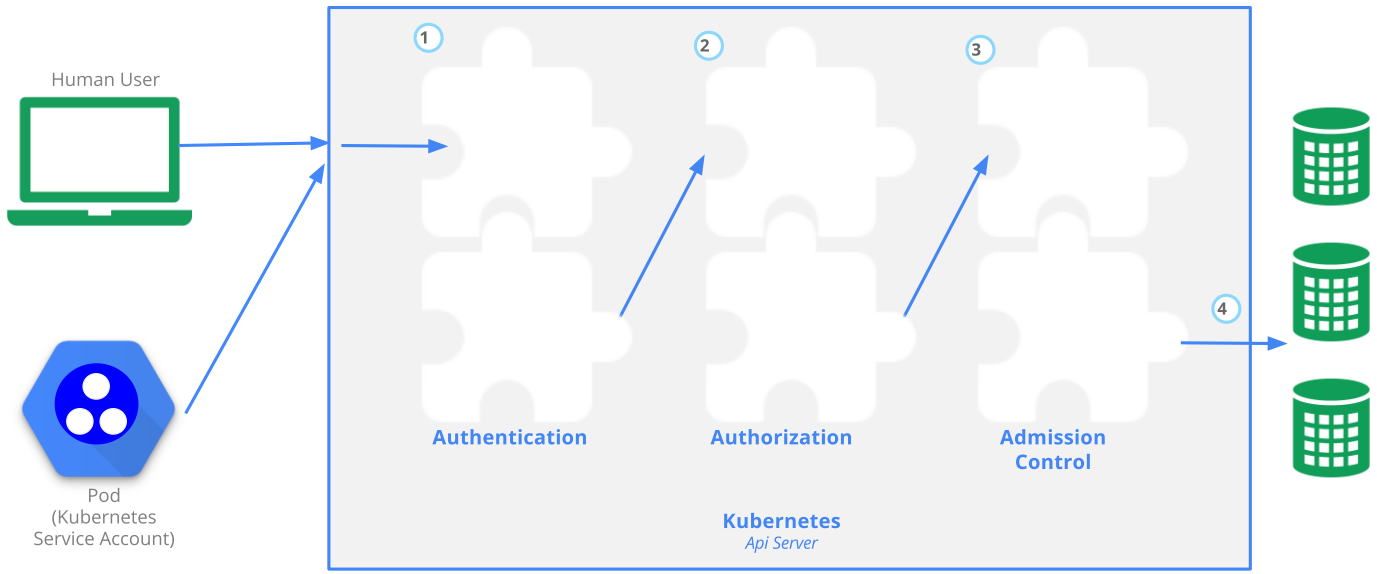
RoleBinding and ClusterRoleBinding

- RoleBinding and ClusterRoleBinding rants the permissions defined in a role to a user or set of users
- It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted
- A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide
- A RoleBinding may reference any Role in the same namespace or reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding
- If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding

kubernetes

Binding Subjects

- When creating a binding, you can specify 3 different subjects: users, groups and ServiceAccounts



kubernetes

Users and Groups

- Kubernetes don't have a traditional user and groups definition
- To add a user, you need to use one of the available authentication methods
  - X509 Client Certs
  - Static Token File
  - Bootstrap Tokens
  - Service Account Tokens
  - OpenID Connect Tokens
  - Webhook Token Authentication
  - Authenticating Proxy
- X509 Client Certs is the preferable way to create local users

kubernetes

Users and Groups

- X509 Client Certs is the preferable way to create local users
- OpenID Connect allow you to connect your cluster with an identity external provider
- Groups are not defined directly on cluster configuration
- On X509 strategy, you can define the group(s) on key definition

kubernetes

Service Accounts

- A Service Account is a type of non-human account that provides a distinct identity in a Kubernetes cluster
- Pods can use a specific ServiceAccount's credentials to identify as that ServiceAccount
- This identity is useful in various situations, including authenticating to the API server or implementing identity-based security policies
- Service Accounts are a namespaced resource
- Every namespace have a default Service Account that is automatically used when any is configured on your resources

kubernetes

Service Accounts Use Cases

- Your pod needs to use API Server and you want to control resources and verbs to be used
- Access a private registry to fetch container images
- Third-party software installed on your cluster needs a Service Account to run

kubernetes

Demo | RBAC

Network Policies

Motivation

- By default, your pods can reach and be reached by any other pod running inside your cluster
- Usually you have your cluster shared by different environment and/or different solutions
- Is important to have a fine-grained way to define egress and ingress policies inside your cluster
- NetworkPolicies allow you to do that

kubernetes

Network Policies

- NetworkPolicies allow you to specify how a pod is allowed to communicate with various network "entities" (services, pods) over the network
- The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers, stating allowed connections:
  - Other pods (exception: a pod cannot block access to itself)
  - Namespaces
  - IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

kubernetes

Network Policies

- When defining a pod- or namespace- based NetworkPolicy, you use a selector to specify what traffic is allowed to and from the Pod(s) that match the selector
- Meanwhile, when IP based NetworkPolicies are created, we define policies based on IP blocks (CIDR ranges)
- However, like ingress, Kubernetes don't have a native implementation of NetworkPolicies
- Network policies are implemented by the network plugin which supports NetworkPolicy
- Creating a NetworkPolicy resource without a controller that implements it will have no effect.

kubernetes

Network Policies: Best Practices

- Start with “default-deny-all” to block everything and then start whitelisting needed connection for each application
- Understand rule evaluation since every rule is added to previous ones (rules are OR’ed and not AND’ed)
- Test your policies carefully and using several patterns
  - Allowed/blocked connections
  - External connectivity
  - Intra-namespace connectivity
- Good source to look into “recipes”: https://github.com/ahmetb/kubernetes-network-policy-recipes

kubernetes

Demo | Network Policies

Affinity

Motivation

- Kubernetes runs a Scheduler to automatically selected best node to run my pd based on some specific metrics
- But sometimes makes sense to give additional information to help scheduler to decide
- Scheduler will use those information to select all nodes that respect the restriction shared, and after select the best one where to schedule the pods

kubernetes

nodeName

- **`nodeName`** is the more direct form to select a node
- If present, Scheduler don't do any calculation and tries to directly schedule the pod
- This property overrules any other related with scheduling
- Some limitations
  - If the named node does not exist, the Pod will not run, and in some cases may be automatically deleted.
  - If the named node does not have the resources to accommodate the Pod, the Pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
  - Node names in cloud environments are not always predictable or stable.

kubernetes

nodeSelector

- **`nodeSelector`** is the simplest recommended form of node selection constraint.
- You can add the **`nodeSelector`** field to your Pod specification and specify the node labels you want the target node to have
- Kubernetes only schedules the Pod onto nodes that have each of the labels you specify
- With this approach you are free from all limitation from **`nodeName`** approach

kubernetes

nodeSelector

- **`nodeSelector`** is the simplest recommended form of node selection constraint.
- You can add the **`nodeSelector`** field to your Pod specification and specify the node labels you want the target node to have
- Kubernetes only schedules the Pod onto nodes that have each of the labels you specify
- With this approach you are free from all limitation from **`nodeName`** approach

kubernetes

Affinity and anti-affinity

- Affinity and anti-affinity works on a similar way as **`nodeSelector`** but expands the types of constraints you can define
- **`nodeSelector`** only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
- You can define rules that are mandatory or preferable
- You can define affinity based on Pods running on nodes instead of nodes properties

kubernetes

Affinity and anti-affinity

- There are two types of node affinity:
  - **`requiredDuringSchedulingIgnoredDuringExecution`**: The scheduler can't schedule the Pod unless the rule is met. This functions like nodeSelector, but with a more expressive syntax.
  - **`preferredDuringSchedulingIgnoredDuringExecution`**: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.
- **`IgnoredDuringExecution`** means that if the node labels change after Kubernetes schedules the Pod, the Pod continues to run.

kubernetes

Node Affinity Weight

- You can specify a weight between 1 and 100 for each instance of the **`preferredDuringSchedulingIgnoredDuringExecution`** affinity type
- When the scheduler finds nodes that meet all the other scheduling requirements of the Pod, the scheduler iterates through every preferred rule that the node satisfies and adds the value of the weight for that expression to a sum
- Nodes with the highest total score are prioritized when the scheduler makes a scheduling decision for the Pod

Pod Affinity

- Inter-pod affinity and anti-affinity require substantial amount of processing which can slow down scheduling
- This amount is impacted from number of rules to process
- Preferable rules have a bigger computing cost than required ones
- Nevertheless, Kubernetes community states that you should only see impact on clusters larger than several hundred nodes

kubernetes

Pod Affinity and Topology Key

- Topology key is mandatory when defining Pod Affinity rules
- This property specify a label to be present on every node to be used to identify them on an unique way
- On on-prem clusters, **`kubernetes.io/hostname`** is commonly used
- But on cloud cluster you use different keys, like region where node is located, a zone inside a region or any additional information

kubernetes

Demo | Affinity

Taints & Tolerations

Motivation

- Affinity and node selectors define how a pod can “select” a node to schedule its containers
- But from Node perspective, could have some reasons for a node to “repel” a pod from being
- To have that behavior, a node can be tainted (marked, contaminated) and to only schedule pods that tolerate that taint

kubernetes

Taints & Tolerations

- Taints (one or more) are applied to the nodes
- Tolerations are applied to pods allowing the scheduler to schedule pods with matching taints
- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes
- By default, nodes don’t have any taint. You need to add it manually

kubernetes

Taints & Tolerations

- Add a taint

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

- Remove a taint

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

kubernetes

Taints & Tolerations: Effects

- When you create a taint you can define 3 effects: **`NoSchedule`**, **`PreferNoSchedule`** or **`NoExecute`**
- **`NoSchedule`**, only allow to schedule pods with toleration
- **`PreferNoSchedule`**, is a soft version of NoSchedule. Pod can be scheduled even without toleration if there is any other node to use
- **`NoExecute`**, when taint is added any pod running on that node without toleration will be evicted

kubernetes

Use Cases

- Dedicated Nodes: If you want to dedicate a set of nodes for exclusive use by a particular set of users/pods. Is used to have system pods on specific nodes without user pods
- Nodes with Special Hardware: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes
- Taint based Evictions: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section

kubernetes

Demo | Taints & Tolerations

Questions?

kubernetes

Lab #07: Network Policies

kubernetes