1

Outline

- Introduction to Access Control
- User Identity in Kubernetes
- Using kubeconfig
- RBAC
- Network Policy
- Security Context
- Security Best Practices

2

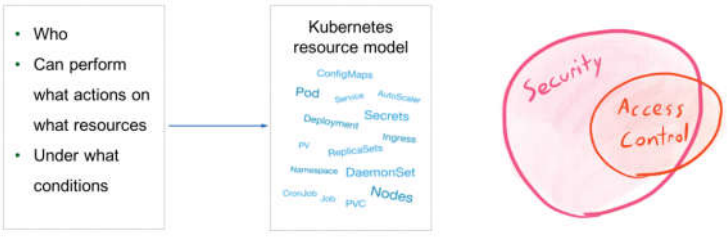
Introduction to Access Control

3

Controlling access

Access control

- **Access control is an important part of Kubernetes security.**
- **Access control can be abstractly defined as the control over who can perform what operations on what resources under what conditions.**
  - Resources refer to the resource models in Kubernetes, such as pods, ConfigMaps, Deployments, and Secrets.

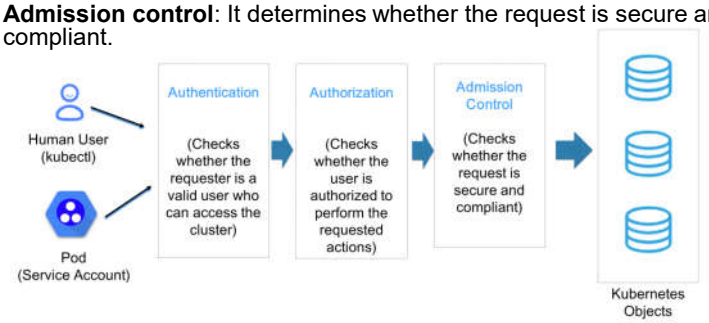


4

Controlling access

Kubernetes API Requests

- **The API server enables access control after receiving the request. Access control is divided into three phases:**
  - **Authentication**: The API server determines whether the requester is a valid user allowed to access the cluster
  - **Authorization**: If the requester is valid, the API server determines whether the requester is authorized to perform the requested operation
  - **Admission control**: It determines whether the request is secure and compliant.

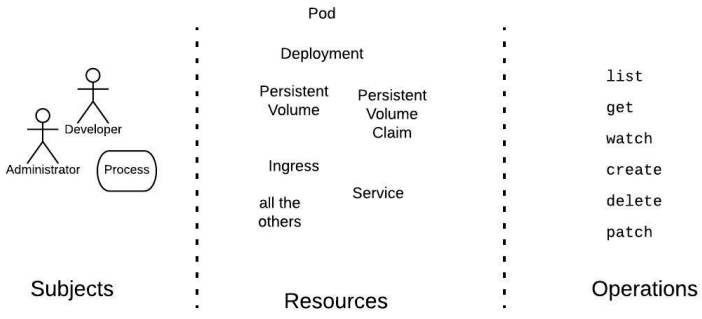


5

Controlling access

Role-based access control (RBAC)

- **RBAC in Kubernetes is the way that you restrict who can access what within the cluster.**
  - A user or group is given permission to perform actions upon certain 'resources'. The association of operations to resources is what is known as a Role.



6

User Identity in Kubernetes

7

User Identity in Kubernetes

- **Kubernetes distinguishes between two kinds of clients connection to API Server**
  - **Users (Actual human user)**
    - Kubernetes doesn't have built in user account management system
    - It should use integrate with external identity management system like OpenID Connect (OAuth2)/Webhook
  - **Service account (Machine like Pod)**
    - Identity of Pod to call API
      - Create : kubectl create serviceaccount {service account name}
      - List : kubectl list sa
      - Assign SA to POD

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
  ...
```


8

Controlling access

Authenticate

- **User identity is used for authenticating request for API Server**
- **How to authenticate user request**
  - Basic HTTP Auth
  - Access token via HTTP Header
  - **Client cert (X509 Certificates)**
  - Custom made
- **X.509 Certificate Authentication (The most commonly used)**
  - The API server starts the Transport Layer Security (TLS)-based handshake process when receiving an access request.
    - The request is initiated through the client certificate which is signed by the cluster-dedicated Certificate Authority (CA) or by the trusted CA in the API server's client CA.
  - By default, it is used by Kubernetes components to authenticate each other and provides access credentials that are often used by kube-config for the kubectl client.
  - X.509 authentication uses JSON Web Tokens (JWTs) that contain metadata such as the issuer, user identity, and expiration time.

9

Controlling access

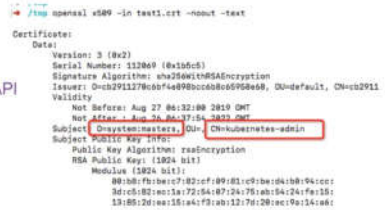
Certificate Authentication

- **The cluster contains a root CA that signs the certificates required by all cluster components to communicate with each other.**
  - A certificate contains the common name (CN) and organization (O), which are the fields related to identity credentials.

•CA

Public key /etc/kubernetes/pki/ca.crt Private key /etc/kubernetes/pki/ca.key

- The certificates used by cluster components to communicate with each other are signed by the cluster's
- A certificate contains two important fields related to identity credentials:
  - Comman Name(CN):indicates a specific user when the API server implements authentication.
  - Organization(O):indicates a specific group when the API server implements authentication.



10

Controlling access

User Certificate signing

- **User certificate signing consists of the following steps:**

Developer

- Create a private key through a certificate tool such as OpenSSL.

```
openssl genrsa –out test.key 2048
```

- Create a CSR.

```
openssl req –new –key test.key –out test.csr –subj "/CN=dahu/O=devs"
```

user group

- Create a Kubernetes CSR instance through an API, or submit the created CSR file to the administrator.

Cluster administrator

- Sign a certificate through the cluster's CA keypair based on the CSR file or instance. The following is an example.

```
openssl x509 –req –in dahu.csr –CA CA_LOCATION/ca.crt –Cakey CA_LOCATION/ca.key –
Cacreateserial –out dahu.crt –days 365
```


11

Controlling access

Service Account Authentication

- **The API server also supports service account authentication in addition to certificate authentication.**

- A service account is an access credential for the API server and is also the only credential in Kubernetes that can be managed through APIs.
- The service account is typically used by the business process of a pod to interact with the API server.
- After a namespace is created, a default service account and a secret instance are also created in this namespace.
- You can also create a differently named service account through an API and attach this service account to a pod in the namespace during runtime.

```
kubectl get serviceaccounts

kubectl get serviceaccounts/build-robot -o yaml
kubectl delete serviceaccount/build-robot

kubectl patch serviceaccount default -p
'{"imagePullSecrets": [{"name": "myregistrykey"}]}'
```

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
```


12

Using kubeconfig

13

How to Create kubeconfig ?

- **kubeconfig is an important access credential for connecting a local device to a Kubernetes cluster. This section describes how to configure and use kubeconfig.**

Configure kubeconfig locally as follows:

- Download the cluster CA.
- Add the cluster connection information through kubectl.

```
kubectl config set-cluster sandbox --certificate-authority=ca.pem --embed-certs=true -server=https://<
public IP address of the target cluster>:6443
```

- Add the new key information to the kubectl configuration.

```
kubectl config set-credentials dahu --client-certificate=dahu.crt --client-key=dahu.key  --embed-
certs=true
```

- Add the new context portal to the kubectl configuration.

```
kubectl config set-context sandbox-dahu --cluster=sandbox --user=dahu
```


14

How to Use kubeconfig

- **In order to access your Kubernetes cluster, kubectl uses a configuration file. The default kubectl configuration file is located at ~/.kube/config and is referred to as the kubeconfig file.**
- **kubeconfig files organize information about clusters, users, namespaces, and authentication mechanisms. The kubectl command uses these files to find the information it needs to choose a cluster and communicate with it.**
- **The loading order follows these rules:**
  1. If the **--kubeconfig** flag is set, then only the given file is loaded. The flag may only be set once and no merging takes place.

     ```
     kubectl get pods --kubeconfig=file1
     ```
  2. If the **$KUBECONFIG** environment variable is set, then it is parsed as a list of filesystem paths according to the normal path delimiting rules for y our system.

     ```
     KUBECONFIG=file1 kubectl get pods
     ```
  3. Otherwise, the **${HOME}/.kube/config** file is used and no merging takes place.
- **Tip : Merging kubeconfig files :**

```
KUBECONFIG=file1:file2:file3 kubectl config view --merge --flatten > out.txt
```


15

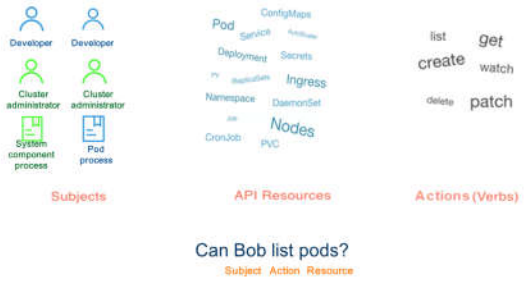
RBAC

16

Controlling access

Three Elements of RBAC

- **Subjects** may be natural persons such as developers and cluster administrators, system component processes, or logical processes in pods.
- **API resources** are the access targets of requests. API resources include all types of resources in a Kubernetes cluster.
- **Verbs** indicate the actions that can be performed on the requested object resources, such as Add, Delete, Modify, Query, List, Get, and Watch.



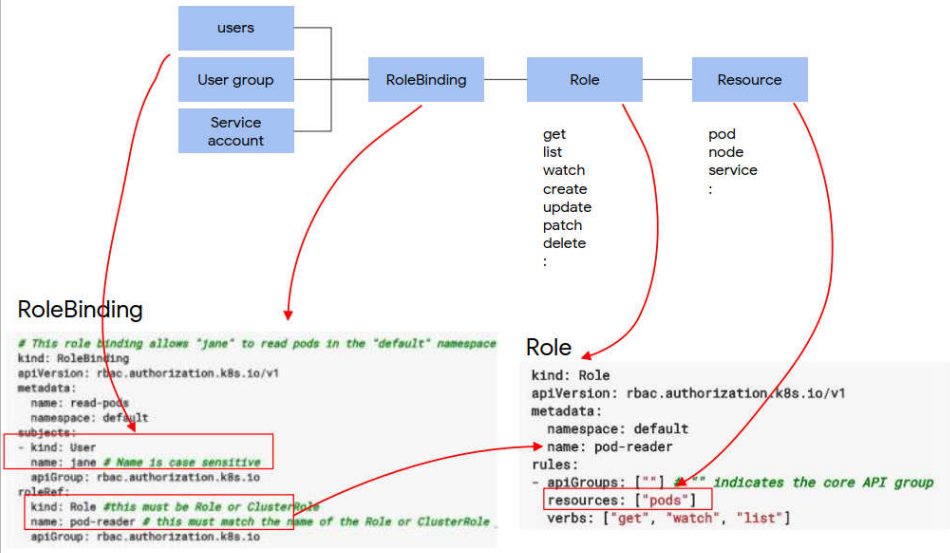
M.Romdhani, 2021

17

17

Controlling access

Role and RoleBinding



M.Romdhani, 2021

18

18

Controlling access

Role vs. ClusterRole

- **Role & RoleBinding are namespaced resource, ClusterRole & ClusterRoleBinding are Cluster wide resource**
  - For example, the permissions for resources such as persistent volumes (PVs) and nodes in a namespace are invisible, but you can define these permissions through a cluster role.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-access
  namespace: test
rules:
- apiGroups: [""]
  resources:  ["pods", "pods/attach"]
  verbs: ["get", "list", "watch"]
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: all-pod-access
rules:
- apiGroups: [""]
  resources:  ["pods", "pods/attach"]
  verbs: ["get", "list", "watch"]
```

The only difference

19

Controlling access

Default Cluster Role Bindings

Default ClusterRole	Default ClusterRoleBinding	Description
system:basic-user	**system:authenticated** group	Allows a user read-only access to basic information about themselves. Prior to v1.14, this role was also bound to `system:unauthenticated` by default.
system:discovery	**system:authenticated** group	Allows read-only access to API discovery endpoints needed to discover and negotiate an API level. Prior to v1.14, this role was also bound to `system:unauthenticated` by default.
system:public-info-viewer	**system:authenticated** and **system:unauthenticated** groups	Allows read-only access to non-sensitive information about the cluster. Introduced in Kubernetes v1.14.

20

Network Policy

21

Controlling access

What is a Network Policy ?

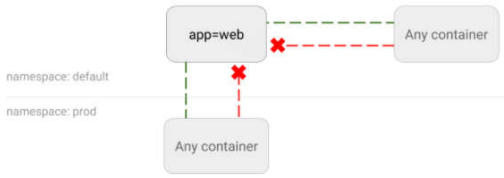
- **Network policies are Kubernetes resources that control the traffic between pods and/or network endpoints.**
  - They control traffic flow at the IP address or port level (OSI layer 3 or 4)
  - They use labels to select pods and specify the traffic that is directed toward those pods using rules.
- **NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "endpoints" and "services"**
- **Network policy can control ingress & egress traffic for Pod**
  - It is based on
    - Label (label selector)
    - Protocol (TCP/UDP), Port
    - IP range (CIDR)

22

Controlling access

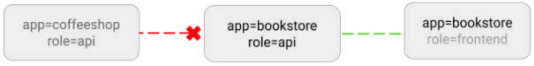
Ingres Control

- **Deny all ingress traffic to Pod**




```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

- **LIMIT traffic to an application**




```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: bookstore
```

M.Romdhani, 2021

23

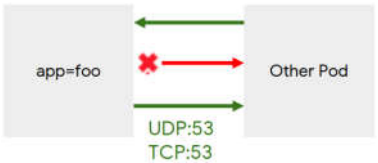
Controlling access

Egress control

- **DENY egress traffic from an application**


```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: foo-deny-egress
spec:
  podSelector:
    matchLabels:
      app: foo
  policyTypes:
  - Egress
  egress: []
```

- **Limit egress traffic**




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: foo-deny-external-egress
spec:
  podSelector:
    matchLabels:
      app: foo
  policyTypes:
  - Egress
  egress:
  - ports:
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
  - to:
    - namespaceSelector: {}
```

M.Romdhani, 2021

24

24

Security Context

25

Security Context

- **Security-related feature can be configured on Pod and its containerthrough-out security-Context properties**
- **It can**
  - Specify the user under which the process in the container will run
  - Prevent the container from running as root
  - Privileged mode (full access to it's node's kernel)
  - Fine grained privileged mode (partial access for node's kernel)

26

Controlling access

Security context example

- **Prevent container run as root**

```
apiVersion: v1
kind: Pod
metadata:
      name: hello-world
spec:
      containers:
# specification of the pod's containers
# ...
securityContext:
      readOnlyRootFilesystem: true
      runAsNonRoot: true
```

- **Run container with specified user**

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```


27

Controlling access

Security context example

- **Run container with full kernel capabilities (Privileged mode) For example NFS**

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-server
  labels:
    role: nfs-server
spec:
  containers:
    - name: nfs-server
      image: jsafrane/nfs-data
      ports:
        - name: nfs
          containerPort: 2049
      securityContext:
        privileged: true
```

- **Set capabilities for Container Adding individual kernel capabilities to a container**

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```


28

Security Best practices

Reference
https://kubernetes.io/blog/2016/08/security-best-practices-kubernetesdeployment

29

Container Image control

- Implement continuous security vulnerability scanning Include security scanning process in CI/CD pipeline
- Regularly apply security updates
- Update container image to latest version (ex node.js etc)
- Ensure that only authorized images are used in your environment

30

Controlling access

Authorization control

- **Create Administrative Boundaries between Resources**

```
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
  "spec": {
    "user": "alice",
    "namespace": "fronto",
    "resource": "pods",
    "readonly": true
  }
}
```

- **Limit direct access to Kubernetes Nodes**
  - You should limit SSH access to Kubernetes nodes. (instead of that user to use kubectl exec)

M.Romdhani, 2021 31

31

Controlling access

Quota control

- **Define Resource Quota**

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

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

Give resource limit to namespace

M.Romdhani, 2021 32

32

Network control

- Implement network segmentation

```
POST /apis/net.alpha.kubernetes.io/v1alpha1/
namespaces/tenant-a/networkpolicys
{
  "kind": "NetworkPolicy",
  "metadata": {
    "name": "pol1"
  },
  "spec": {
  "allowIncoming": {
    "from": [{
      "pods": { "segment": "frontend" }
    }],
    "toPorts": [{
      "port": 80,
      "protocol": "TCP"
    }]
    },
    "podSelector": {
      "segment": "backend"
    }
  }
}
```


33

Security context for Pod

- SecurityContext->runAsNonRoot : Indicates that containers should run as non-root user
- SecurityContext->Capabilities : Controls the Linux capabilities assigned to the container.
- SecurityContext->readOnlyRootFilesystem : Controls whether a container will be able to write into the root filesystem.
- PodSecurityContext->runAsNonRoot: Prevents running a container with 'root' user as part of the pod

34