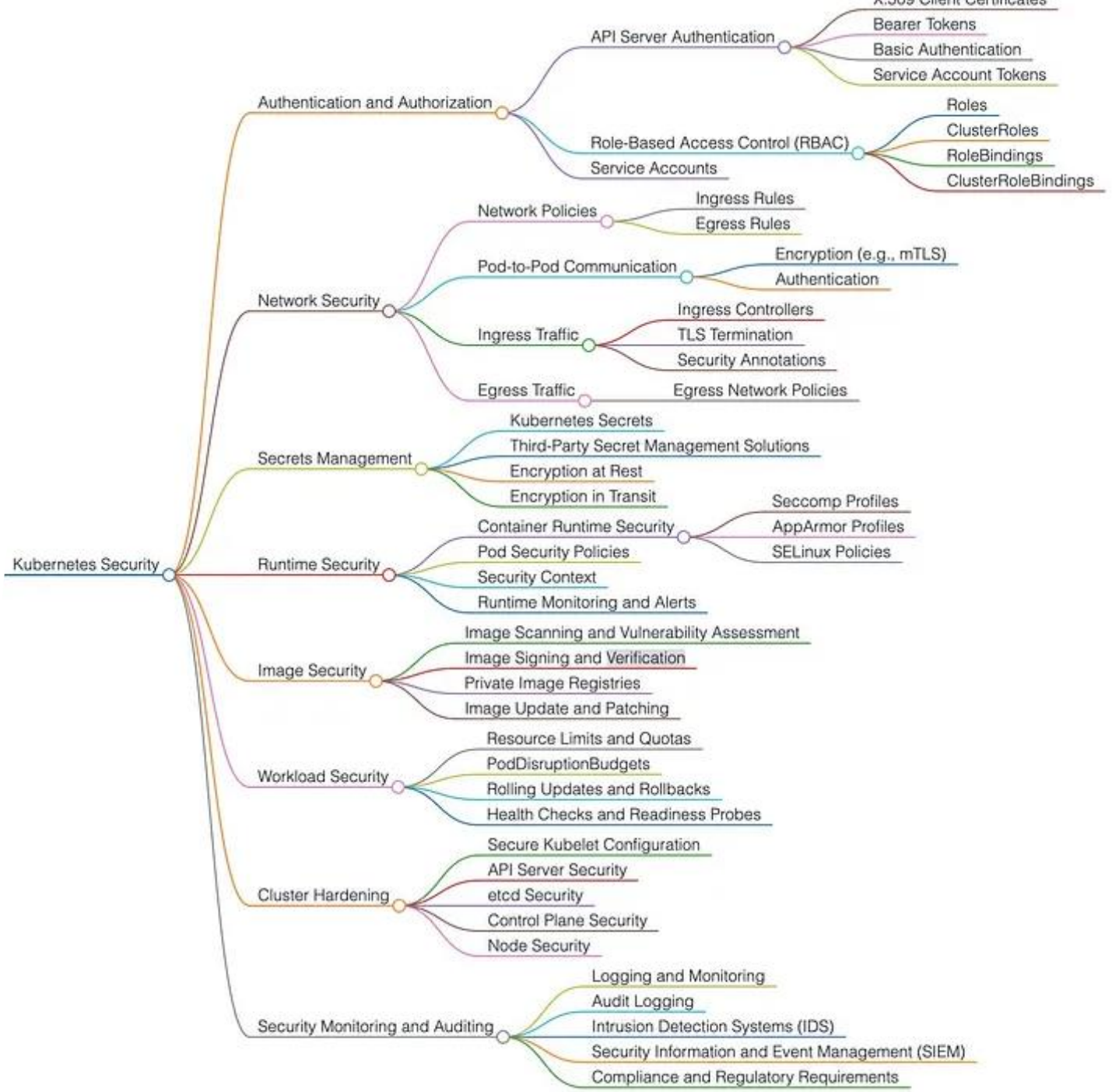
Kubernetes Security

An In-Depth Look

Lisa Zangrando

Kubernetes Security high-level mind map

- Security in Kubernetes is not only about securing the cluster as a whole;
- It is based on a multi-layered model that addresses potential risks, both at the cluster and application levels.
  - **Cluster-level, Workload, Network Ssecurity**
- Each layer requires specific approaches and tools to ensure a secure environment.



Access Security

API-server

- The API Server is the central entry point of Kubernetes.
- Every interaction with the cluster, whether from users or applications, passes through it.
- Protecting access to the API Server is essential to ensure the overall security of the system.
- Access security is divided into three fundamental aspects end related best practices:
  - **authentication, authorization, and auditing**

Authentication in Kubernetes

- Authentication in Kubernetes is the first layer of security to control who can access the cluster.
- Main authentication methods:
  - **Basic Authentication (Not Recommended)**
  - **X.509 Client Certificates**
  - **OpenID Connect (OIDC)**
  - **Webhook authentication**

- Best practices for authentication in Kubernetes:
  - **Limit public access to the API Server:** vpn, firewall, etc
  - **Disable Basic Authentication:** deprecated method
  - **Limit client certificates:** they are difficult to revoke and manage at scale
  - **Use OIDC for centralized user management**
    - To enable OIDC, configure the API Server:

```
--oidc-issuer-url=https://<OIDC_PROVIDER_URL>
--oidc-client-id=<CLIENT_ID>
--oidc-username-claim=email
--oidc-groups-claim=groups
```

Authorization in Kubernetes

- This authorization layer, ensure that each entity can only access the resources and actions it is actually authorized for.
- Main authorization methods:
  - **ABAC (Attribute-Based Access Control)**
  - **RBAC (Role-Based Access Control)**
  - **Node authorization**
  - **Webhook authorization**
- Best practices
  - **Principle of Least privilege**
  - **Use Service Accounts for applications**
  - **Isolate Permissions by namespace**
  - **Avoid unnecessary ClusterRoleBindings**
  - **Audit RBAC Configurations**
  - **Automate RBAC Management**
  - **Configure Default Deny Access**

Role-Based Access Control

- **Definition**:
  - Role-Based Access Control (RBAC) is the most common method for managing permissions in Kubernetes.
  - It allows you to define who can perform specific actions on which resources within the cluster by assigning roles to users or groups.
  - RBAC is flexible, scalable, and ensures that only authorized users can interact with the Kubernetes cluster.
- **Key concepts:**
  - **Role**: a set of permissions that define what actions can be performed on which resources. Roles are specific to namespaces.
  - **ClusterRole**: a role that defines permissions at the cluster level, including across namespaces.
  - **RoleBinding**: a binding that associates a Role with a user or group within a namespace.
  - **ClusterRoleBinding**: a binding that associates a ClusterRole with a user or group across the entire cluster.

Role

- A **Role** is a set of permissions that specify **what actions** can be performed on **which resources**.
- These permissions are scoped to a specific namespace, making Roles ideal for managing access within a limited scope.
- **Actions**: common verbs include `get`, `list`, `create`, `update`, `delete`, etc.
- **Resources**: these can be Kubernetes resources like Pods, Deployments, ConfigMaps, Secrets, etc.

The following example defines a Role that allows users to manage Pods in the development namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: pod-manager
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "create", "delete"]
  - apiGroups: [""]
    resources: ["configmaps"]
    resourceNames: ["my-config"]
    verbs: ["get"]
```

In this example:

- the Role applies to the development namespace.
- the user can manage Pods but cannot interact with other resources like Services, but he can get access to my-config ConfigMaps.

ClusterRole

- A **ClusterRole** is similar to a Role but operates at the cluster level. It can define permissions that span **multiple namespaces** or even apply to cluster-wide resources such as nodes or PersistentVolumes.
- **Use Cases**
  - Granting access to non-namespaced resources (e.g., nodes, persistentvolumes).
  - Managing permissions across all namespaces.

The following ClusterRole allows users to view nodes and PersistentVolumes:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-viewer
rules:
  - apiGroups: [""]
    resources: ["nodes", "persistentvolumes"]
    verbs: ["get", "list"]
```

This ClusterRole can be bound to users or groups that need to interact with cluster-wide resources.

RoleBinding

- A **RoleBinding** connects a Role to specific users, groups, or service accounts within a namespace.
- It grants the permissions defined in the Role to the specified subjects.
- OIDC TOKEN

```
{ "sub": "1234567890",
  "name": "John Doe",
  "email": johndoe@example.com,
  "groups": ["developers", "admins"]
}
```

The following RoleBinding associates the pod-manager Role with a specific user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-pod-manager
  namespace: development
subjects:
  - kind: User
    name: johndoe@example.com
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-manager
  apiGroup: rbac.authorization.k8s.io
```

In this example, the user `johndoe@example.com` can now manage Pods in the development namespace:

- **Subjects**: defines who receives the permissions (e.g., users, groups, or service accounts).
- **roleRef**: specifies the Role being assigned.

RoleBinding

- A **RoleBinding** connects a Role to specific users, groups, or service accounts within a namespace.
- It grants the permissions defined in the Role to the specified subjects.
- OIDC TOKEN

```
{ "sub": "1234567890",
  "name": "John Doe",
  "email": johndoe@example.com,
  "groups": ["developers", "admins"]
}
```

The following RoleBinding associates the pod-manager Role with a specific user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-pod-manager
  namespace: development
subjects:
  - kind: Group
    name: developers
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-manager
  apiGroup: rbac.authorization.k8s.io
```

In this example, the group **developers** can now manage Pods in the development namespace

ClusterRoleBinding

- A **ClusterRoleBinding** is the cluster-wide equivalent of a RoleBinding.
- It associates a ClusterRole with users, groups, or service accounts, granting permissions across the entire cluster.

The following ClusterRoleBinding grants cluster-wide read-only access to all resources for a specific group:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-only-cluster-binding
subjects:
  - kind: Group
    name: developers
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-viewer
  apiGroup: rbac.authorization.k8s.io
```

Here, members of the `developers` group can view resources cluster-wide.

RBAC: Group membership

- If a user belongs to multiple groups, such as user and admin, Kubernetes **does not choose a single group**.
- Instead, it evaluates the **combined permissions** of all groups.
- **Implications:**
  - Kubernetes checks all **RoleBindings** and **ClusterRoleBindings** associated with the user's groups.
  - The user is granted the **union of permissions** from all their groups.
- Kubernetes RBAC is **permissive**. If one group grants elevated access, it prevails over restrictive rules from other groups.

Example scenario:

- User belongs to two groups:
  - **user** → Limited permissions (e.g. Read-only access to Pods in app-namespace)
  - **admin** → Full administrative permissions.
- The user gains **all permissions** granted by both groups.

RoleBinding with ClusterRole

- Create a RoleBinding that refers to a ClusterRole is allowed.
- The RoleBinding is used to bind a ClusterRole to a specific group of users within a given namespace, allowing them to inherit the permissions associated with the ClusterRole.
- In this example, the RoleBinding allows users in the developer group to inherit cluster-wide permissions from the ClusterRole but applies them in the app-namespace.
- Key benefit: enables flexible access control, allowing cluster-wide permissions to be scoped within a specific namespace.

The ClusterRole grants read-only access to Pods and other resources across the entire cluster.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # Name of the ClusterRole
  name: view-only
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: developer-view-binding
  namespace: app-namespace
subjects:
- kind: Group
  name: developer
  apiGroup: ""
roleRef:
  kind: ClusterRole
  name: view-only
  apiGroup: rbac.authorization.k8s.io
```

How RBAC works?

- **Defining Roles and ClusterRoles**:
  - Administrators define what actions are permissible on specific resources by creating Roles or ClusterRoles.
- **Binding Roles to Users**:
  - RoleBindings and ClusterRoleBindings assign these roles to specific users, groups, or service accounts.
- **Authorization**:
  - When a user or process sends a request to the API server, Kubernetes checks the RBAC policies to determine if the action is allowed.

RBAC limitations

- **No support for fine-grained permissions**
  - e.g.: restrict users to only delete their own Pods
  - ownership not natively supported
  - requires combining **RBAC** with **Admission Controllers** to enforce label-based restrictions or using tools like **OPA (Open Policy Agent)**
- **Granularity of permissions**
  - e.g.: controlling who can modify only specific annotations or labels within a Pod
- **No support for Resource Quotas**
  - e.g.: restrict the number of Pods a user can create

Audit logging

- **Definition**:
  - Audit logs record detailed information about API requests and responses, including the user or service that made the request, the resources accessed, and the outcome of the action.
  - These logs are invaluable for detecting unauthorized access, troubleshooting issues, and ensuring that Kubernetes operates according to organizational policies.
- **Key features:**
  - Tracks all requests made to the API Server
  - Captures detailed metadata, including user identity, resource types, actions, and timestamps
  - Helps in monitoring and enforcing compliance with security policies
  - Enables detailed analysis for auditing and debugging purposes

How to configure Audit

- To enable audit logging in Kubernetes, you need to configure the API Server to capture and store the audit logs.
- **step 1:** Create an Audit Policy
  - An audit policy defines which events should be logged and at which level of detail. The policy is specified in a YAML file (see exmple).
- **step 2:** Enable Audit logging on the API Server
  - You need to modify the API Server's configuration to enable audit logging and specify the audit policy file.
- **Best Practices**
  - Enable Audit logs
  - Define a custom Audit Policy
  - Monitor logs for anomalies
  - Link Audit to Access Control
  - Automate log analysis
  - Schedule periodic reviews

```yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  - level: Metadata
    resources:
      - group: ""
        resources: ["pods"]
  - level: RequestResponse
    resources:
      - group: "apps"
        resources: ["deployments"]
        namespaces: ["developers"]
```

In this example:

- Metadata level logs basic details (e.g., request timestamp, user, resource name).
- RequestResponse level logs full request and response data for deployments in the apps group.

API Server configuration example:

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml
--audit-log-path=/var/log/kubernetes/audit.log
--audit-log-maxage=30
--audit-log-maxbackup=10
--audit-log-maxsize=100
```

Thanks!

References

- https://kubernetes.io/docs/concepts/security/
- https://kubernetes.io/docs/concepts/security/controlling-access/
- https://kubernetes.io/docs/reference/access-authn-authz/authentication/
- https://kubernetes.io/docs/reference/access-authn-authz/rbac
- https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/