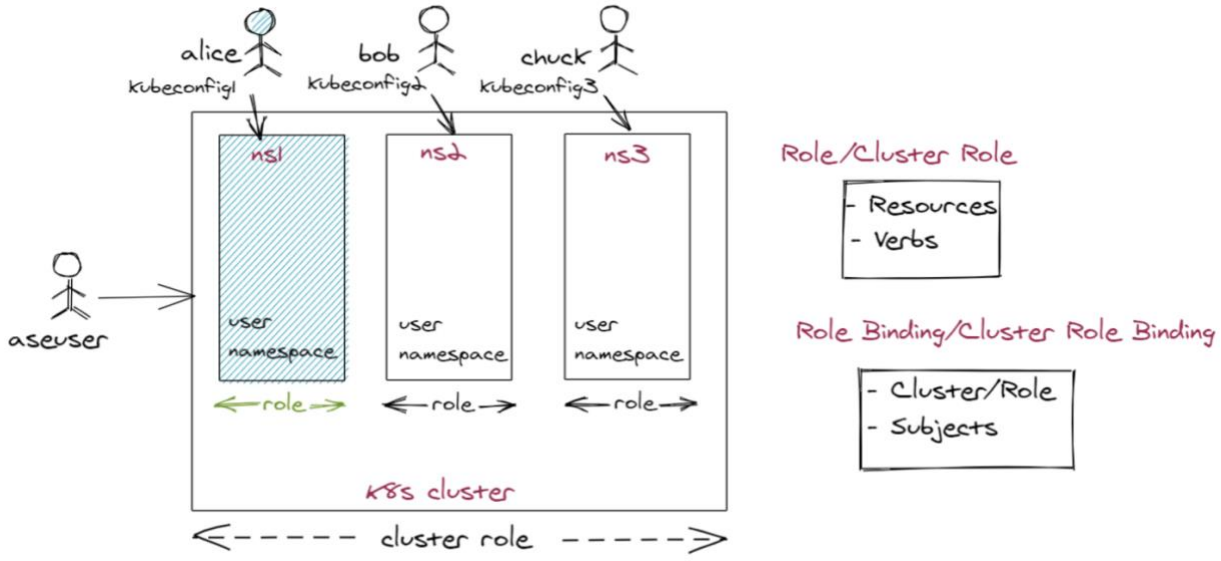
K8s Security and Access Control – Role Based Access Control (RBAC)

Securing a Kubernetes cluster involves protecting its **control plane**, **data**, and **workloads** from unauthorized access, misconfigurations, and runtime threats. Kubernetes provides several native mechanisms for security and access control that work together to enforce **who can do what**, **how pods behave**, and **how sensitive data is managed**.
Key areas of Kubernetes security include:

- **Authentication** – Verifying the identity of users and service accounts.
- **Authorization (RBAC)** – Controlling what authenticated identities can access or modify.
- **Admission Control** – Validating or modifying requests before resources are created.
- **Secrets Management** – Securely storing and delivering sensitive data like passwords and tokens.
- **Pod Security** – Defining how workloads should run (e.g., non-root, restricted capabilities).
- **TLS and API Security** – Securing communication between clients and the Kubernetes API server.

Role-Based Access Control (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 an enterprise. In Kubernetes, RBAC lets you control who can do what within your cluster by defining roles with specific permissions, then assigning these roles to users, groups, or service accounts. This ensures only authorized identities perform allowed actions on cluster resources.



Alice, Bob, and Chuck have admin access limited to their respective namespaces (ns1, ns2, ns3). The cluster admin has full access to system namespaces and cluster-wide resources.

Key Concepts:

- **Role:** A namespace-scoped set of permissions (rules) defining allowed actions on resources.
- **ClusterRole:** Similar to Role but applies cluster-wide or across multiple namespaces.
- **RoleBinding:** Grants permissions from a Role to users/groups within a namespace.
- **ClusterRoleBinding:** Grants ClusterRole permissions cluster-wide.
- **Subjects:** Users, groups, or service accounts granted access.
- **Verbs:** Allowed actions like get, list, create, delete.
- **Resources:** Kubernetes objects such as pods, services, secrets.

Best Practices:

- **Principle of Least Privilege:** Grant minimal necessary permissions.
- **Namespace Scoping:** Prefer Roles and RoleBindings over cluster-wide permissions.
- **Avoid Wildcards:** Do not use broad * permissions unless absolutely required.
- **Separation of Duties:** Different roles for developers, operators, CI systems.
- **Use Groups:** Assign permissions to groups, simplifying management at scale.
- **Regular Audits:** Review RBAC rules periodically to remove stale permissions.

Use Cases:

- **Developer Isolation:** Developers get RoleBindings scoped to their project namespaces allowing pod/deployment management but no cross-namespace access.
- **Cluster Admins:** ClusterRoleBindings grant full admin rights to core platform teams managing cluster lifecycle.
- **Service Accounts:** Fine-grained roles for microservices or controllers that need API access only to their domain resources.
- **Multi-Tenancy:** Enforce strict tenant isolation by combining namespaces and RBAC to segment teams/customers sharing infrastructure.

Important Considerations:

- RBAC permissions are additive; overlapping bindings can lead to privilege creep.
- ClusterRoles can be bound within namespaces; Roles cannot be used cluster-wide, however ClusterRoles can be used within namespaces.
- Lack of explicit permission means access is denied (default deny).
- RBAC is enforced by the API server on every request.
- Proper RBAC setup supports detailed audit trails and compliance.

Real World Insights:

- Misconfigured verbs like escalate, bind, or impersonate risk privilege escalation.
- Over-permissioned service accounts in pods present serious attack surfaces if pods are compromised.
- Complement RBAC with Pod Security Admission controls for workload security.
- Use automation (kubectl auth can-i) to verify effective permissions.
- Conduct regular RBAC reviews to minimize the attack surface in dynamic, large clusters.

RBAC Example

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: my-namespace
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: my-namespace
subjects:
- kind: ServiceAccount
  name: my-serviceaccount
  namespace: my-namespace
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Service Accounts

Service accounts provide an identity for processes that run in a pod. Essentially, a service account is like a user identity specifically for applications running inside pods. It enables these applications to authenticate securely to the Kubernetes API, without relying on human user credentials.

Key Concepts:

- **Service Account:** Namespaced identity for pod processes.
- **Tokens:** Automatically generated JWT tokens stored in secrets, mounted into pods for API authentication.
- **Default Service Account:** Auto-assigned to pods unless specified otherwise.
- **Bindings:** Permissions are assigned via RoleBindings/ClusterRoleBindings like user accounts.
- **Token Mounting:** Pods mount service account tokens by default.

Best Practices:

- Use dedicated service accounts per application or microservice, not the default account.
- Apply least privilege principles when assigning permissions.
- Avoid wildcard permissions for service accounts.
- Disable token auto-mounting (automountServiceAccountToken: false) if the pod doesn't need API access.
- Use namespace-scoped service accounts, minimizing cluster-wide privileges.

- Prefer projected service account tokens with expirations for improved security.
- Avoid binding service accounts to highly privileged roles like cluster-admin unless strictly necessary.

Use Cases:

- **Controllers/Operators:** Custom controllers use service accounts scoped to their managed resources.
- **API Access for Apps:** Apps inside pods authenticate with service accounts to securely access API for configmaps, secrets, or monitoring data.
- **Tenant Isolation:** Separate service accounts per tenant/application with tightly scoped roles.
- **Automation:** CI/CD systems use service accounts to deploy or modify cluster resources programmatically.

Important Considerations:

- Tokens are mounted by default, which can leak if pods are compromised.
- Service accounts are namespace-bound unless ClusterRoleBindings extend permissions cluster-wide.
- API permissions depend entirely on RBAC bindings.
- Projected tokens are preferred over default long-lived tokens for better security.
- Token revocation requires RoleBinding changes or service account deletion.

Real World Insights:

- Using default service accounts inadvertently grants over-permissioned access—a common security flaw in enterprises.
- Compromised pods with service account tokens can be exploited to escalate privileges or access sensitive data.
- Organizations create granular service accounts per microservice for tight permission control.
- Service accounts enable secure, auditable automation without sharing human credentials.
- Combine service account restrictions with Pod Security Standards and network policies for layered defense.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  annotations:
    kubernetes.io/enforce-mountable-secrets: "true"
  name: my-serviceaccount
  namespace: my-namespace
```

Secrets Management

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Secrets store sensitive information separately from your application code, and Kubernetes manages them securely to allow pods to access these secrets safely without exposing sensitive data in plaintext.

Key Concepts:

- **Secret Object:** Kubernetes resource storing base64-encoded sensitive data.
- **Secret Types:** Opaque (generic), docker-registry, TLS, etc.
- **Storage:** Secrets reside in etcd, often encrypted at rest.
- **Consumption:** Pods use secrets via environment variables, mounted volumes, or API calls.
- **RBAC Controlled:** Access limited by RBAC policies.
- **Automatic Injection:** Secrets can be injected into pods automatically.

Best Practices:

- Enable encryption at rest for secrets in etcd.
- Apply least privilege RBAC for secret access.
- Never store secrets in code or ConfigMaps.
- Bind secrets to dedicated service accounts with minimal scope.
- Rotate secrets regularly to reduce compromise windows.
- Use external secret management tools (HashiCorp Vault, AWS Secrets Manager) for lifecycle and audit management.
- Prefer mounting secrets as files over environment variables to reduce exposure.
- Enable audit logging to track secret access.

Use Cases:

- **Database Credentials:** Securely provide database passwords to applications.
- **TLS Certificates:** Distribute certs for secure communication between microservices.
- **OAuth Tokens/API Keys:** Manage third-party service credentials.
- **Private Docker Registry:** Authenticate pods pulling images from private registries.
- **Encryption Keys:** Store app-level encryption keys for data protection.

Important Considerations:

- Base64 encoding is not encryption—enable etcd encryption to secure secrets.
- Any entity with appropriate RBAC can read secrets via API.
- Secret size is limited (~1MB); avoid storing large binaries.
- Secret updates may not propagate immediately to running pods without reload or restart.
- Pod compromise can expose secrets accessible to that pod.
- Secure etcd backups as they contain secrets.
- Audit secret access regularly to detect misuse.

Real World Insights:

- Many enterprises neglect etcd encryption by default, exposing secrets at rest.
- External secret stores are preferred at scale for rotation, auditing, and fine-grained access.
- Environment variables expose secrets to process listing or logs; mounting as files is safer.
- Secret sprawl increases management complexity; enforce strict secret policies.
- Monitoring secret access has helped detect insider threats and misconfigurations early.
- Least privilege service accounts and network policies reduce risk if pods are compromised.

Secrets Management Example YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
  namespace: my-namespace
type: Opaque
data:
  username: YWRtaW4=   # base64 for 'admin'
  password: c2VjcmV0   # base64 for 'secret'
```

Admission Controllers

Admission Controllers are plugins that intercept requests to the Kubernetes API server after authentication and authorization but before the object is persisted. They can validate, mutate, or reject requests to enforce cluster policies and enhance security, compliance, and governance.

Key Concepts:

- Mutating Admission Controllers: Modify objects before they are persisted (e.g., injecting sidecars).
- Validating Admission Controllers: Validate objects and reject those that violate policies.
- Built-in Controllers: Kubernetes ships with controllers like NamespaceLifecycle, ResourceQuota, PodSecurity, etc.
- Webhook Admission Controllers: Custom controllers deployed as webhooks to enforce organization-specific policies.
- Execution Order: Admission controllers run in a predefined sequence.

Best Practices:

- Enable critical admission controllers by default (e.g., NamespaceLifecycle, LimitRanger, PodSecurity).
- Use validating webhooks to enforce custom organizational policies.
- Keep webhook admission controllers highly available to avoid API server request delays or failures.
- Test admission controllers thoroughly in staging before production rollout.

- Avoid excessive mutation to keep object state predictable.

Use Cases:

- Enforcing label or annotation standards on all pods.
- Automatically injecting sidecar containers (e.g., Istio Envoy proxies) into pods.
- Denying creation of pods that violate security contexts or resource limits.
- Preventing privileged container creation via policy enforcement.
- Quota enforcement preventing resource overconsumption.

Important Considerations:

- Admission controllers can block or alter requests, impacting cluster operations.
- Misconfigured webhooks can cause API request timeouts or failures.
- Some controllers cannot be disabled (critical to cluster function).
- Admission control decisions are synchronous, impacting API latency.
- Logging and monitoring admission controller activity is essential for troubleshooting.

Real World Insights:

- FANG companies rely heavily on webhook admission controllers to enforce complex security and compliance rules dynamically.
- Automated policy enforcement reduces manual errors and improves security posture at scale.
- Careful management of admission controllers improves cluster stability and governance.
- Continuous testing and gradual rollout prevent disruptions caused by new admission controller policies.
- Integrating admission control with CI/CD pipelines enables automated security gating.

Pod Security

Pod Security is a set of controls and policies aimed at ensuring pods run with appropriate security contexts, minimizing risk from misconfigurations or vulnerabilities within workloads.

Key Concepts:

- Pod Security Standards (PSS): Define baseline, restricted, and privileged security levels.
- SecurityContext: Pod or container-level settings specifying privileges, user IDs, capabilities, etc.
- Pod Security Admission Controller: Enforces PSS policies on pod creation/update.
- Linux Capabilities: Fine-grained control over privileges granted to containers.
- Seccomp Profiles: Kernel syscall filtering for pods.
- SELinux/AppArmor: Mandatory access control mechanisms to restrict pod behavior.

Best Practices:

- Enforce Pod Security Standards appropriate for your workload sensitivity.
- Run containers as non-root users unless absolutely required.

- Drop unnecessary Linux capabilities to reduce attack surface.
- Use seccomp and SELinux/AppArmor profiles for runtime confinement.
- Avoid privileged containers and host network/IPC unless necessary.
- Regularly audit pod security contexts and update policies.

Use Cases:

- Running multi-tenant clusters with strict pod isolation requirements.
- Hardening workloads handling sensitive data.
- Preventing privilege escalation within pods.
- Enforcing compliance with industry security standards (e.g., PCI-DSS, HIPAA).
- Applying consistent security baseline across dev, staging, and production.

Important Considerations:

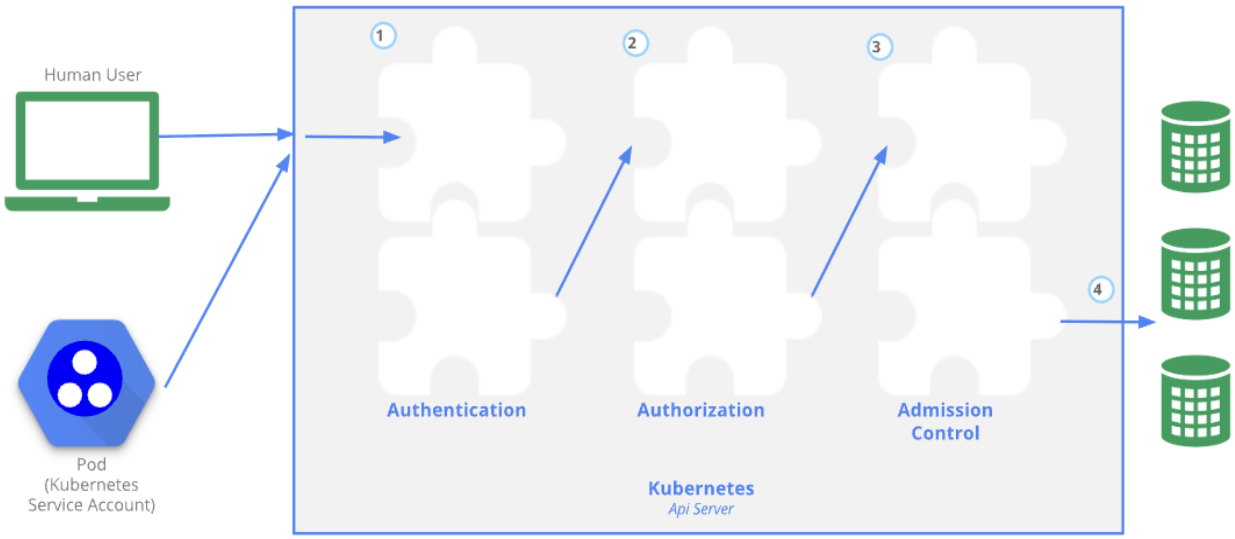
- Pod Security Admission is enforced at API server; policies can block non-compliant pod creation.
- Some legacy workloads may require policy exceptions or gradual remediation.
- Overly restrictive policies can cause deployment failures or operational friction.
- Combine pod security with network policies and RBAC for layered defense.
- Monitoring pod security violations helps detect risky workloads.

Real World Insights:

- FANG scale clusters rely on automated pod security admission combined with policy-as-code tooling.
- Gradual enforcement strategies ease migration to stricter pod security postures.
- Runtime security tools complement static pod security controls by detecting anomalous behaviors.
- Pod security policies significantly reduce risks of container escape and privilege escalation attacks.
- Integrating pod security with CI/CD gates ensures only compliant workloads reach production.

TLS and API Access

TLS and API Access mechanisms protect communication between clients and the Kubernetes API server, ensuring confidentiality, integrity, and authentication.



Key Concepts:

- TLS Encryption: Secures all API traffic between clients and the Kubernetes API server.
- Client Certificates: Used for mutual TLS authentication of users or service accounts.
- Bearer Tokens: Common authentication method using service account JWT tokens.
- Authentication Plugins: Support multiple auth mechanisms (OIDC, webhook token auth, etc.).
- API Server Authorization: RBAC or ABAC enforce access control after authentication.
- API Server Endpoints: Exposed via secure IPs or load balancers, often behind network firewalls.

Best Practices:

- Always enable TLS for all Kubernetes API server endpoints.
- Use short-lived client certificates or tokens for authentication.
- Integrate with enterprise identity providers (OIDC, LDAP) for user authentication.
- Rotate certificates and tokens periodically.
- Limit API server exposure with firewall rules and private networking.
- Audit API server access logs regularly.

Use Cases:

- Users accessing Kubernetes dashboard or kubectl with client certificates or tokens.
- CI/CD pipelines authenticating to the API server via service accounts and tokens.
- External systems integrating via OIDC or webhook token authentication.
- Internal cluster components communicating securely with the API server.
- Enforcing fine-grained access control using RBAC post authentication.

Important Considerations:

- Improper TLS setup risks man-in-the-middle attacks.
- Stale or compromised tokens/certificates can lead to unauthorized access.
- API server endpoint exposure must be minimized for security.
- Logging and monitoring API server authentication attempts help detect breaches.
- Integration with centralized auth systems improves manageability.

Real World Insights:

- FANG-scale clusters tightly control API access through zero-trust networking and short-lived credentials.
- Mutual TLS authentication is common for highly secure environments.
- Identity federation simplifies user and system access management at scale.
- Automated certificate and token rotation reduces risk of credential leaks.
- Combining strong TLS with RBAC and audit logging is foundational to cluster security.