Top Five Kubernetes RBAC Mistakes to Avoid

If you run workloads in Kubernetes, you know how much important data is accessible through the Kubernetes API—from details of deployments to persistent storage configurations to secrets. The Kubernetes community has delivered a number of impactful security features in 2017 and 2018, including Role-Based Access Control (RBAC) for the Kubernetes API. RBAC is a key security feature that protects your cluster by allowing you to control who can access specific API resources. Because the feature is relatively new, your organization might have configured RBAC in a manner that leaves you unintentionally exposed. To achieve least privilege without leaving unintentional weaknesses, be sure you haven't made any of the following five configuration mistakes.

The most important advice we can give regarding RBAC is: “Use it!” Different Kubernetes distributions and platforms have enabled RBAC by default at different times, and newly upgraded older clusters may still not enforce RBAC because the legacy Attribute-Based Access Control (ABAC) controller is still active. If you’re using a cloud provider, this setting is typically visible in the cloud console or using the provider’s command-line tool. For instance, on Google Kubernetes Engine, you can check this setting on all of your clusters using `gcloud`:

```
$ gcloud container clusters list --
format='table[box](name,legacyAbac.enabled)'
┌────────────┬─────────┐
│ NAME       │ ENABLED │
├────────────┼─────────┤
│ with-rbac  │         │
│ with-abac  │ True    │
└────────────┴─────────┘
```

Once you know that RBAC is enabled, you’ll want to check that you haven’t made any of the top five configuration mistakes. But first, let’s go over the main concepts in *the Kubernetes RBAC system*.

Your cluster's RBAC configuration controls which **subjects** can execute which **verbs** on which **resource types** in which **namespaces**. For example, a configuration might grant **user alice** access to **view** resources of type **pod** in the namespace **external-api**. (Resources are also scoped inside of API groups.)

These access privileges are synthesized from definitions of:

- **Roles**, which define lists of **rules**. Each rule is a combination of verbs, resource types, and namespace selectors. (A related noun, Cluster Role, can be used to refer to resources that aren't namespace-specific, such as nodes.)
- **Role Bindings**, which connect ("bind") roles to subjects (users, groups, and service accounts). (A related noun, **Cluster Role Binding**, grants access across all namespaces.)

In Kubernetes 1.9 and later, Cluster Roles can be extended to include new rules using the *Aggregated ClusterRoles* feature.

This design enables fine-grained access limits, but, as in any powerful system, even knowledgeable and attentive administrators can make mistakes. Our experiences with customers have revealed the following five most common mistakes to look for in your RBAC configuration settings.

Configuration Mistake 1
Cluster Administrator Role Granted Unnecessarily

The built-in `cluster-admin` role grants effectively unlimited access to the cluster. During the transition from the legacy ABAC controller to RBAC, some administrators and users may have replicated ABAC's permissive configuration by granting `cluster-admin` widely, neglecting the warnings in *the relevant documentation*. If users or groups are routinely granted `cluster-admin`, account compromises or mistakes can have dangerously broad effects. Service accounts typically also do not need this type of access. In both cases, a more tailored Role or Cluster Role should be created and granted only to the specific users that need it.

Configuration Mistake 2
Improper Use of Role Aggregation

In Kubernetes 1.9 and later, *Role Aggregation* can be used to simplify privilege grants by allowing new privileges to be combined into existing roles. However, if these aggregations are not carefully reviewed, they can change the intended use of a role; for instance, the `system:view` role could improperly aggregate rules with verbs other than view, violating the intention that subjects granted `system:view` can never modify the cluster.

Configuration Mistake 3
Duplicated Role Grant

Role definitions may overlap with each other, giving subjects the same access in more than one way. Administrators sometimes intend for this overlap to happen, but this configuration can make it more difficult to understand which subjects are granted which accesses. And, this situation can make access revocation more difficult if an administrator does not realize that multiple role bindings grant the same privileges.

Configuration Mistake 4
Unused Role

Roles that are created but not granted to any subject can increase the complexity of RBAC management. Similarly, roles that are granted only to subjects that do not exist (such as service accounts in deleted namespaces or users who have left the organization) can make it difficult to see the configurations that *do* matter. Removing these unused or inactive roles is typically safe and will focus attention on the active roles.

Configuration Mistake 5
Grant of Missing Roles

Role bindings can reference roles that do not exist. If the same role name is reused for a different purpose in the future, these inactive role bindings can suddenly and unexpectedly grant privileges to subjects other than the ones the new role creator intends.

Summary

Kubernetes RBAC configuration is a critical control for the security of your containerized workloads. Properly configuring your cluster RBAC roles and bindings helps minimize the impact of application compromises, user account takeovers, application bugs, or simple human mistakes. Check your clusters today—have you made any of these configuration mistakes?