Achieving Least-Privilege Access in Kubernetes

Eran Leib
Co-Founder, VP Product Management
apolicy

Daniel Pacak
Open Source Engineer
Aqua Security

September 2020

Agenda

- RBAC & Kubernetes
- Common Pitfalls
- Best Practices
- The Road To Least Privilege
- Demo

About Us

Eran Leib

Co-Founder, VP Product Management

Co-Founder & VP of Product Management at Apolicy and has more than 20 years of experience in security, identity, access and policies.

Prior to Apolicy, I co-founded Whitebox Security, a Data Access Governance platform that was acquired by SailPoint Technologies (NYSE: SAIL).

I enjoy hiking, movies, snowboarding, Lego (but not stepping on them!), cooking (without burning stuff) and traveling for fun.

Daniel Pacak

Open Source Engineer

Daniel Pacak is an Open Source Engineer at Aqua Security. He works on Kubernetes and container security related projects, while also taking part in maintaining the CNCF's project, Harbor.

When he isn't at work, he enjoys taking walks in the woods with his family.

"Role Based Access Control (RBAC) is complex and takes more time and effort than most organizations realize in order to get it right."

Gartner 2017

Role Based Access Control (RBAC)

- RBAC is a simple method of managing access in large systems
- In RBAC we group permissions and assign them to groups of users
- RBAC is a compromise between operations and security
- Security wants users with exact permissions per user
- Operations wants manageability
- It is important to find the balance, meaning, a good amount of roles

WHAT IF I TOLD YOU THAT
ROLES ARE NOT JOBS
imgflip.com

Roles, Roles, Roles & ClusterRoles

- A role is a group of rules
- Rule = a permission set
- Roles rules are scoped to a namespace
- ClusterRole rules scope is the cluster
- ClusterRoles can also be scoped down to a namespace with the right configuration

Defining a role does not provide access

Role

Scope: Namespace

Label(s)

Rule(s)

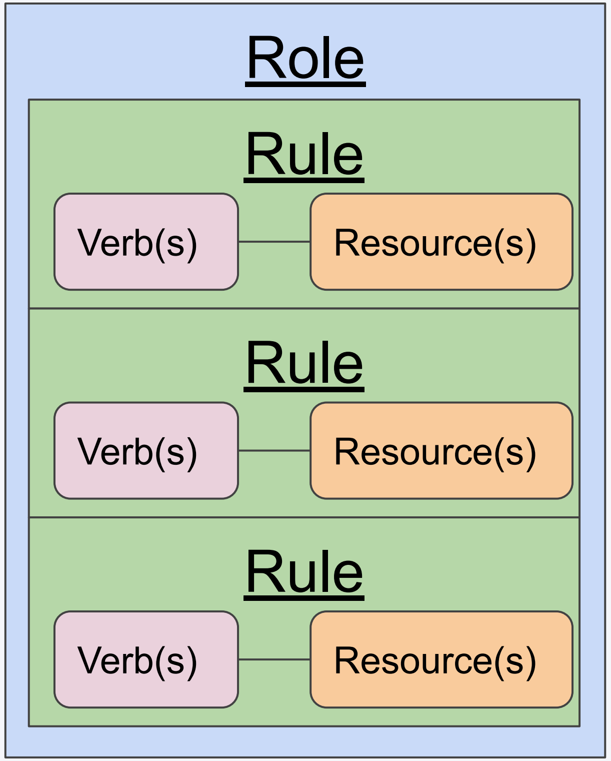
ClusterRole

Label(s)

Rule(s)

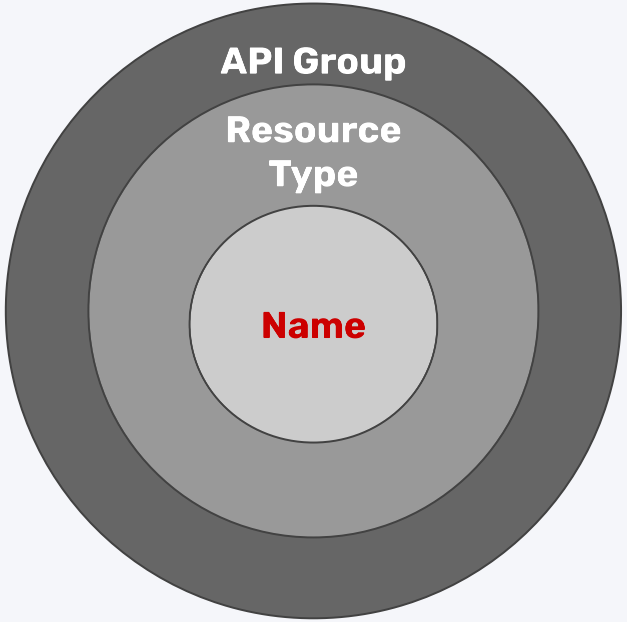
Rules

- Rules are the building blocks of roles
- Each rule addresses a key question: *What is allowed?*
- A verb is an action, for example, *get, create*
- Kubernetes is an API first platform, hence, resources are also defined as APIs and API groups
- All rules are always ALLOWing access



Resources - Permissions Targets

- Resources are defined top-down
- Each layer defined, narrows down the applied resources
- Leaving the resources blank, translates to “All resources in the API Group”
- A specific (resource) name would narrow the selection to the specific resource



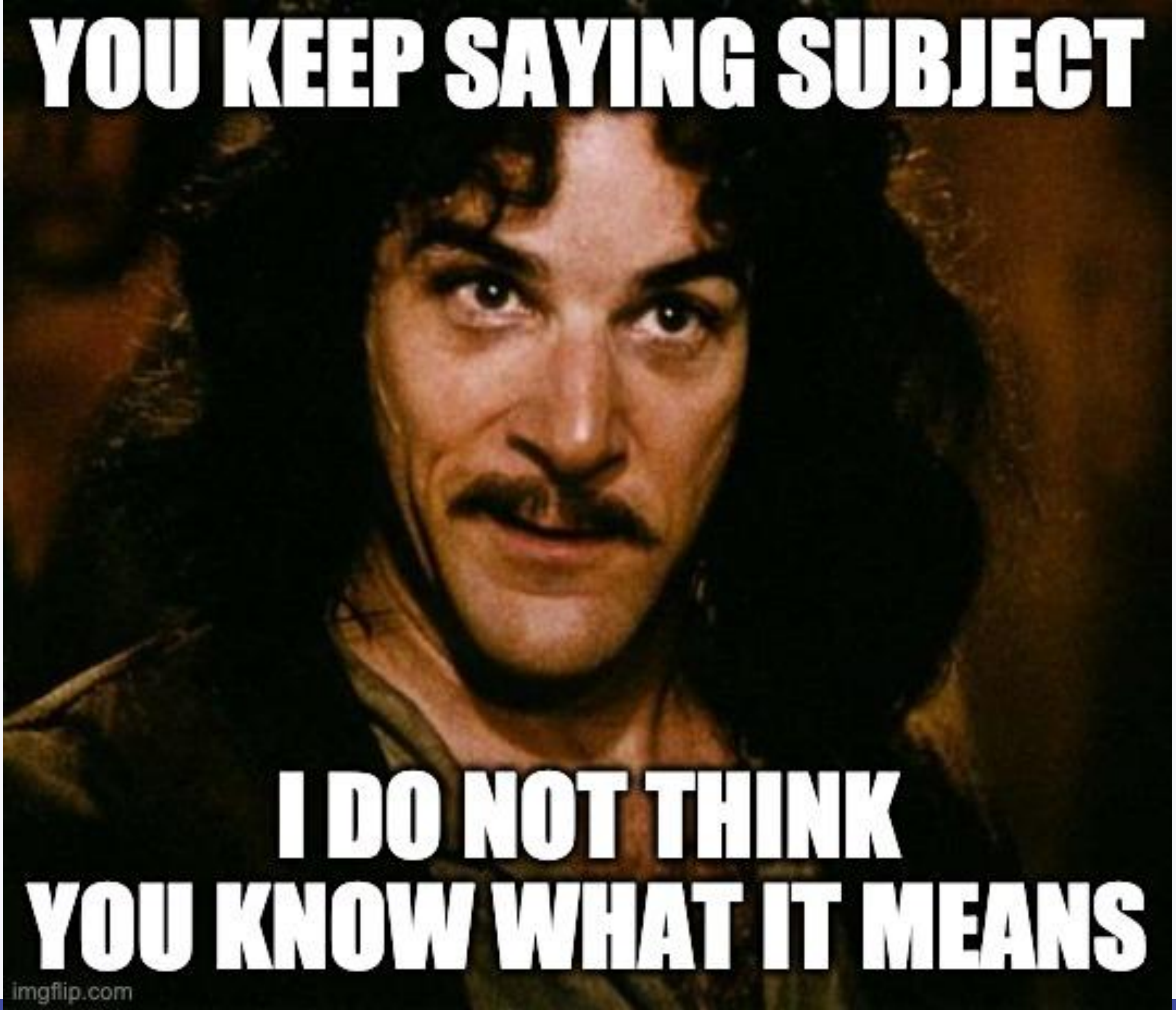
NonResourceURL

- There are resources that are not API resources.
  They are simply URLs. For example: /healthz

- Their access is controlled using Kubernetes permissions

- NonResourceURLs are defined as Cluster level objects
  (even though they're not actually cluster objects)

- To use them, they must be part of a ClusterRole and associated with ClusterRoleBinding


YOU KEEP SAYING SUBJECT
I DO NOT THINK
YOU KNOW WHAT IT MEANS
imgflip.com

Subjects

- Users, groups and service accounts are subjects in Kubernetes
- Associating roles with subjects assign access to subjects
- Access is granted directly or indirectly (through groups)
- The complete set of access is calculated and the result is the actual access list for the user
- Kubernetes does not have DENY rules, therefore, all access is augmented

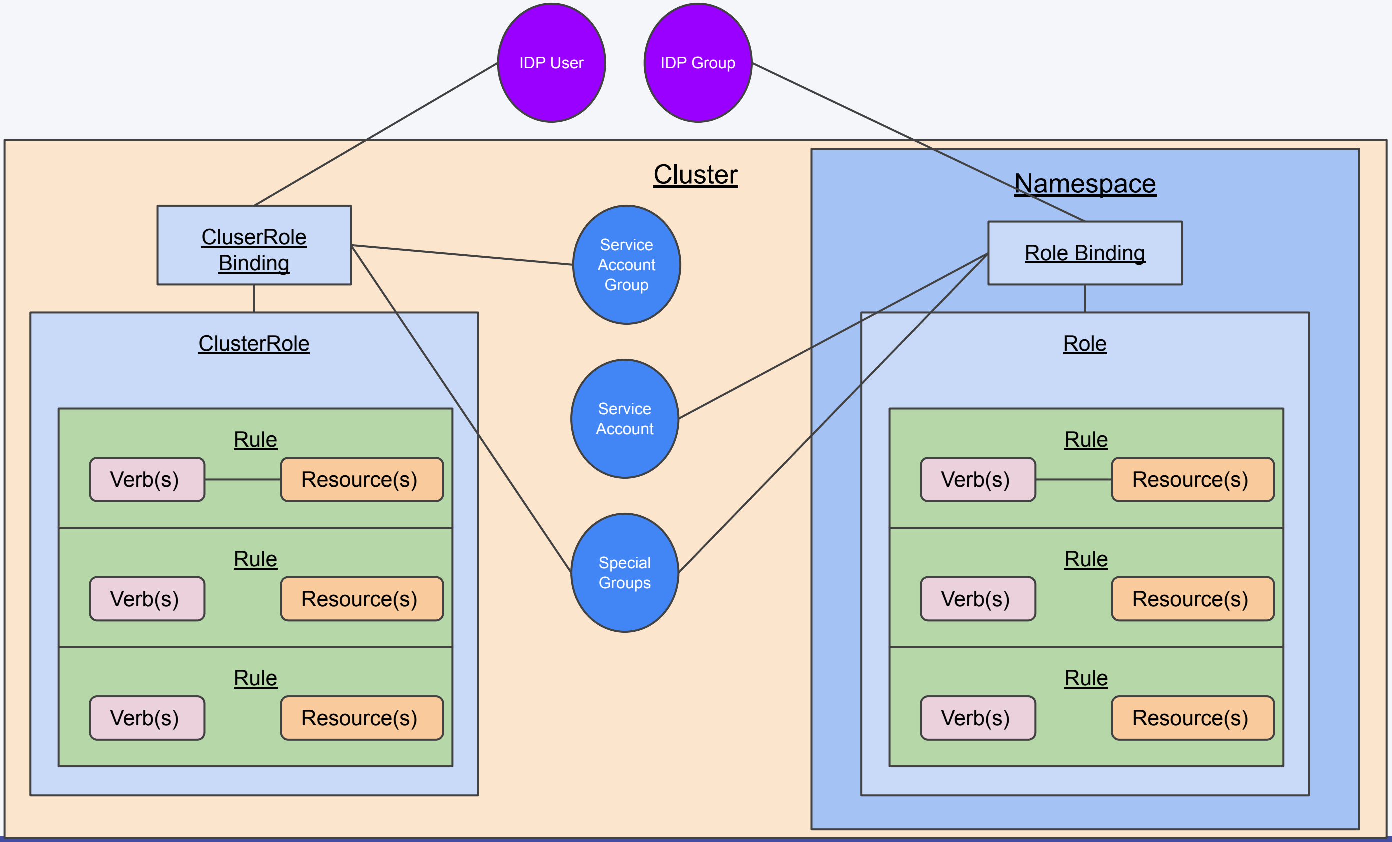
Subject Types

Special Groups	Service Account Group	Service Account	IDP User	IDP Group
Automatically populated groups prefixed with `system:` like: `authenticated` `Unauthenticated` `masters`	Automatically populated groups for service accounts. One for every namespace and one global	Users defined internally in Kubernetes to run workloads	Users authenticated by the identity provider	Groups from the identity provider consists users and other groups

Connecting The Dots

- Role, rules, subjects, what's next?
  Binding roles and subjects with (Cluster)RoleBinding!
- RoleBinding and the binded Role are in the same namespace.
- When you bind to a (Cluster)Role you bind to all of its rules
- Bind service accounts from any namespace to roles in the role binding namespace
- ***Yes…*** *a role binding is for a single role*
  Yes… *you cannot change the role binded, you have to re-create*

Complete structure visual



What Can I Do?

- By now you probably understood that the to know if a user can or cannot do something can be tricky
- Kubectl has a command called *kubectl can-i* to help us understand if we can do something or to list all our permissions
- It is also very useful for cluster admins to figure out if a service account has the permissions they think he has
- To use this there's the ability to impersonate *--as <user>*

Knowing Effective Access Is A Must

- Compliance requirements, incidents investigations and operations all requires understanding who can do what.
- Analyzing the effective access for subjects is challenging:
  - Direct vs indirect access
  - Resource definition is cumbersome and scoping makes it more complex
  - Built in groups, general hygiene are not helping either
- Large scale clusters, multi cloud and hybrid doesn’t make it easier
- All of the above combined is what you probably have

Examples

```
kubectl create role view-pods --namespace=webinar \
  --resource=pods --verb get,list,watch \
  --dry-run=client --output=yaml
```

```
kubectl create rolebinding webinar-ns-admin-can-view-pods \
  --namespace=webinar \
  --role=view-pods \
  --serviceaccount=webinar:webinar-ns-admin
```

```
kubectl describe clusterrole view
```

```
kubectl create rolebinding webinar-ns-admin-can-view-any-resource \
  --namespace=webinar \
  --clusterrole=view \
  --serviceaccount=webinar:webinar-ns-admin
```

aqua run free

apolicy

Examples

```
kubectl auth can-i list pods \
  --namespace=webinar \
  --as system:serviceaccount:webinar:webinar-ns-admin
yes
```

```
kubectl auth can-i create pods \
  --namespace=webinar \
  --as system:serviceaccount:webinar:webinar-ns-admin
no
```

```
kubectl auth can-i --list --namespace=webinar \
  --as system:serviceaccount:webinar:webinar-ns-admin
Resources   Non-Resource URLs   Resource Names   Verbs
pods        []                  []               [get list watch]
```

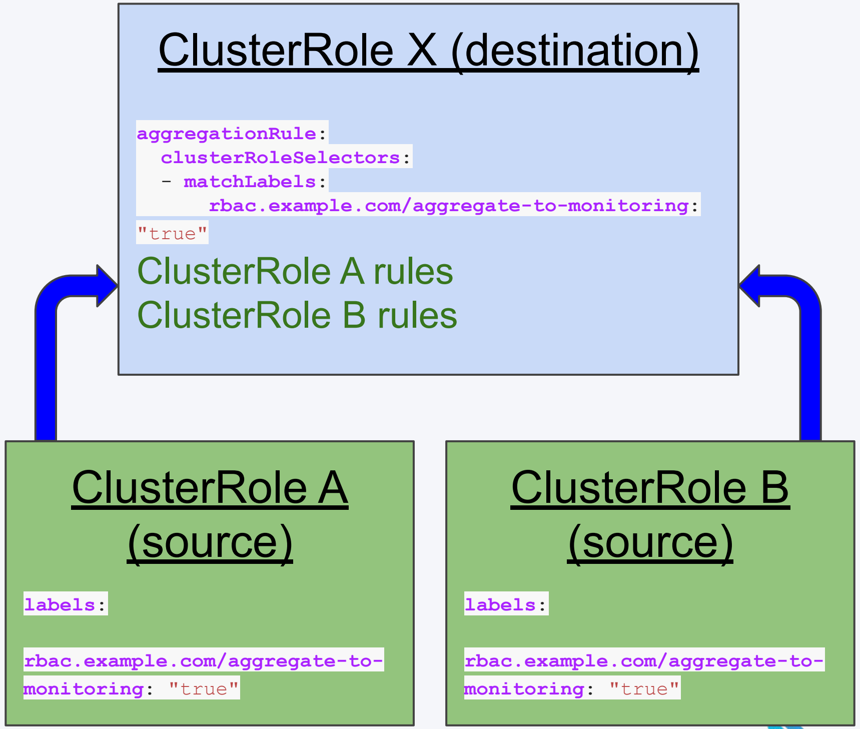
```
kubectl who-can list pods --namespace=webinar --output=wide
ROLEBINDING                    ROLE            NAMESPACE  SUBJECT            TYPE            SA-NAMESPACE
webinar-ns-admin-can-view-pods Role/view-pods  webinar    webinar-ns-admin   ServiceAccount  webinar
```

To Default Or Not To Default

- Kubernetes provides a *default* account per namespace
- If not defined explicitly, workloads will run under that account
  It's simple, easy & … usually WRONG
- This usually causes organizations to add access to <namespace>:default account
- In turn, that account ends up with much more access than needed
  (hint: needed = nothing!)
- It is also harder to trace issues if you're using a default account

Aggregate → ClusterRole

- ClusterRoles have a special mechanism to aggregate their rules to other ClusterRoles
- Kubernetes uses a Controller to aggregate rules using special labels
- This in turn allows extending built in roles with custom resources by aggregating their access



```
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring:
"true"
```

```
labels:
  rbac.example.com/aggregate-to-
monitoring: "true"
```

```
labels:
  rbac.example.com/aggregate-to-
monitoring: "true"
```

RoleBinding → ClusterRole

- ClusterRoles are normally associated using ClusterRole bindings
- However, you can still associate a ClusterRole using a namespaced role binding
- One of the purposes of this is to create a SINGLE cluster level role with all around access, whilst scoping the access down using a role binding
- For example, ClusterRole with create POD associated using a role binding in namespace A will allow the subjects to create PODs only in Namespace A

Examples

kubectl create clusterrole **view-nodes** --verb=**get,list,watch** --resource=**node**

kubectl create clusterrolebinding **webinar-ns-admin-can-view-nodes** \
  --namespace=webinar \
  --clusterrole=**view-nodes** \
  --serviceaccount=**webinar:webinar-ns-admin**

kubectl create clusterrole **check-healthz** \
  --verb=**get** --non-resource-url=**/healthz**

kubectl create clusterrolebinidng **webinar-ns-admin-can-check-healthz** \
  --clusterrole=**check-healthz** \
  --serviceaccount=**webinar:webinar-ns-admin**

kubect auth can-i **get /healtz** \
--as system:serviceaccount:**webinar:webinar-ns-admin**
yes

Common Pitfalls

- Validation is a common issue in the RBAC mechanism
- Creating a role doesn’t validate the rules
- Creating a role binding doesn’t validate the roles and subjects
- Changes of course do not trigger validations (modify or delete)
- Keeping your cluster hygiene is important to avoid loopholes

Common Pitfalls (2)

- Built in groups are a great mechanism as long as you use them correctly
- A member of a built in group will always get that group access
- You should consider for every rule assigned to them if it is the right place
- The most important ones are:
`system:authenticated`, `system:unauthenticated`,
`system:serviceaccounts`,
`system:serviceaccounts.<namespace>`

Least Privilege - What Is It Good For?

- The principle of least privilege is the idea that at any user, program, or process should have only the bare minimum privileges necessary to perform its function
- To achieve least privilege we need to have a clear picture of what the user needs to do
- Service Accounts are simpler to keep as least privileges. Their activities are set and would only change if the workload change
- For regular users, we can achieve “close to least privileges” and adapt the access over time

aqua run free

apolicy

Isolating Risky Access

- A great way of managing access on the road to least privilege is to isolate risky access into designated roles
- That way, when needed to remove specific risky access it would be much simpler and will not require breaking down more roles
- While doing so, avoid providing such access to built in groups as their membership is uncontrolled
- The thought behind creating such roles is to group rules that are required together

Least Privilege = Operational Headache?

- Maintaining just the right privileges for every user is challenging
- The trade off is always a tension point between security and operations
- A reasonable approach is:
  - Service Accounts with the tightest least privileged
  - Normal users
    - Risky access in designated isolated roles and keep access to minimum
    - Non risky in a more “relaxed mode”
    - Focus your energy where it’s needed

Audit To The Rescue

- Are they actually using it? Is a question I often hear
- The answer is right in front of us: Audit Trail
- Kubernetes audit trail is very elaborate and helps tracking usage
- It is yet another stepping stone on the road to least privileged
- Keeping tabs and correlating with the effective access is still complex though

Audit Log : Forbidden Anonymous User

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "RequestReceived",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username": "system:anonymous",
    "groups": [
      "system:unauthenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  }
}
```

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username": "system:anonymous",
    "groups": [
      "system:unauthenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  },
  "responseStatus": {
    "metadata": {},
    "status": "Failure",
    "reason": "Forbidden",
    "code": 403
  },
  "annotations": {
    "authorization.k8s.io/decision": "forbid",
    "authorization.k8s.io/reason": ""
  }
}
```

Audit Log : Forbidden with Access Token

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "RequestReceived",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username":
"system:serviceaccount:webinar:webinar-ns-admin",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:webinar",
      "system:authenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  }
}
```

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username":
"system:serviceaccount:webinar:webinar-ns-admin",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:webinar",
      "system:authenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  },
  "responseStatus": {
    "metadata": {},
    "status": "Failure",
    "reason": "Forbidden",
    "code": 403
  },
  "annotations": {
    "authorization.k8s.io/decision": "forbid",
    "authorization.k8s.io/reason": ""
  }
}
```

aqua run free

apolicy

Audit Log : Allowed with Access Token

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "RequestReceived",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username":
"system:serviceaccount:webinar:webinar-ns-admin",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:webinar",
      "system:authenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  }
}
```

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/webinar/pods",
  "verb": "list",
  "user": {
    "username":
"system:serviceaccount:webinar:webinar-ns-admin",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:webinar",
      "system:authenticated"
    ]
  },
  "userAgent": "curl/7.64.1",
  "objectRef": {
    "resource": "pods",
    "namespace": "webinar",
    "apiVersion": "v1"
  },
  "responseStatus": {
    "metadata": {},
    "code": 200
  },
  "annotations": {
    "authorization.k8s.io/decision": "allow",
    "authorization.k8s.io/reason": "RBAC: allowed by
RoleBinding \"webinar-ns-admin-view/webinar\" of ClusterRole
\"view\" to ServiceAccount \"webinar-ns-admin/webinar\""
  }
}
```

run free

Step By Step For ServiceAccounts

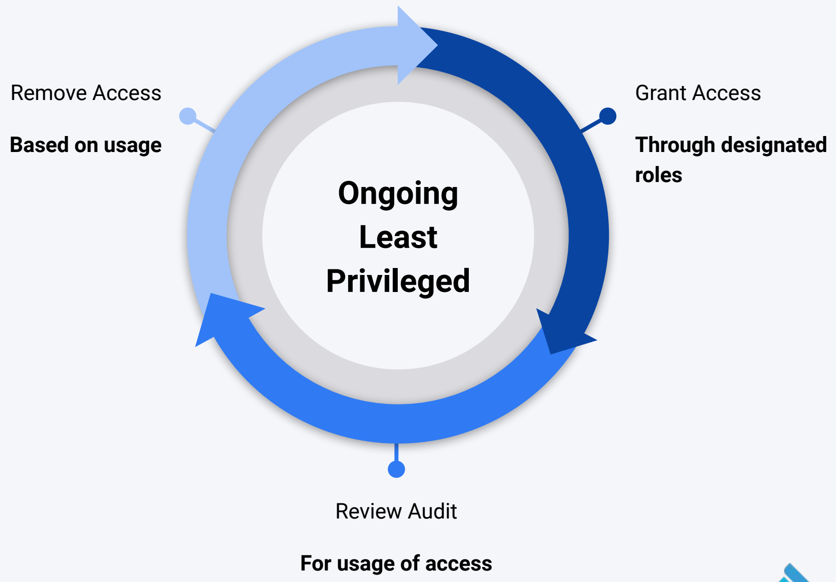
- Use named accounts
- Separate your ServiceAccount with designated roles and bindings
- Use built in audit to track unused access
- Remove unused access over time
- Grant new required access exclusively through the designated roles

Step By Step For ServiceAccounts

Day 0

1	Use A Named Account	• Create an Service Account in the namespace • Set the account in the workload
2	Create Designated Roles & Bindings	• Create roles & bindings in each scope • Associate them to the user
3	Setup Audit	• Make sure the audit is turned on • Configure it to match your needs

Day 1+



aqua run free

apolicy

5 Things To Do Today

- Start using named accounts - even if your account currently doesn’t have any special access
- Disable automount token = THE most least privileged
- Think twice before assigning access to system: subjects that are groups
- Make sure your clusters don’t have loose ends - keep it clean
- Use the audit capabilities to help with visibility

Achieving Least-Privilege Access in Kubernetes

Eran Leib
Co-Founder, VP Product Management
apolicy

Daniel Pacak
Open Source Engineer
Aqua Security

September 2020

It’s time for

Better Kubernetes

Be Risk Smart

Assess workload exposure and prioritize risks for action

Be Declarative

Achieve the workload state you’ve declared

Be Right

Prevent issues before they arise

apolicy