Role-based Access Control (RBAC) Authorization in Kubernetes

Garsha Rostami

Galaxy Consulting L.L.C., Minnesota, USA
E-mail: Galaxy-Consulting-LLC@outlook.com

Received 08 August 2022; Accepted 08 June 2023;
Publication 11 September 2023

Abstract

In computer systems security, role-based access control (RBAC) or role-based security is an approach to restricting system access to authorized users [1]. This paper will describe how the Kubernetes RBAC authorization sub-system works, how to leverage it to secure access to resources in the cluster, and how to validate the set policies through impersonation to ensure users and service accounts are granted the intended rights.

Keywords: Kubernetes Role-based Access Control, RBAC, Kubernetes Role, Kubernetes RoleBinding, Kubernetes ClusterRole, Kubernetes ClusterRoleBinding, Kubernetes authorization, Kubernetes API groups, Kubernetes aggregationRule, Kubernetes impersonation.

1 Introduction

Kubernetes is a very complex system. Managing and securing it requires a lot of planning and expertise. Role-based Access Control (RBAC) is just one aspect of overall Kubernetes security system and as such a brief discussion of overall Kubernetes security is provided in this section.

Journal of ICT Standardization, Vol. 11_3, 237–260.
doi: 10.13052/jicts2245-800X.1132

Kubernetes has a layered architecture. Each layer has its own security requirements and settings. This layered architecture provides defence in depth which is highly effective and desirable, provided that these layers are properly protected. [Figure 1] shows an overview of this architecture. Below provides a brief discussion of these layers from a security standpoint:

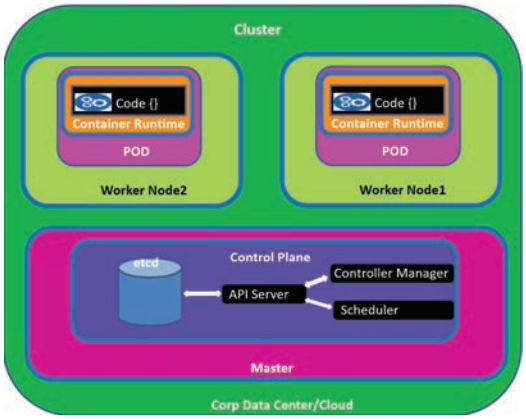

Figure 1 Kubernetes layered architecture.

1. A Kubernetes cluster is set up either in a corporate data center or hosted on a cloud provider's infrastructure. Either environment will need to secure the perimeter, servers, and networks through traditional firewalls, IPsec, etc.
2. Within the Kubernetes cluster, access to the master and worker nodes must be restricted. Kubelets on nodes use client certificates to communicate with the API server. If a node is compromised, a skilled attacker may use those certificates to gain access to the control plain.
3. The control plain is at the heart of a Kubernetes cluster. "etcd" is the datastore for all Kubernetes information and if compromised, you should assume that the cluster is compromised. It is recommended to setup strong credentials (mTLS) from the API server to the etcd and keep the servers hosting the etcd behind a firewall and allow access in only to the API server.
4. Access to Kubernetes resources such as Pods, Services, Secrets, etc. must be regulated and tightly controlled. People and service accounts must be given enough permissions to perform their function but not more. This is done through RBAC which is the subject of this article.

5. Pods running with unconstrained security privileges such as running as root will pose serious security threats to the worker nodes and potentially the entire cluster. Kubernetes new Pod Security Policies and OPA (Open Policy Agent) are some the tools that can be leveraged to mitigate Pod related security misconfigurations.
6. Within the cluster, Pod to Pod communications should regulated through network policies so that if a Pod is compromised, an attacker is not provided with a large attack surface area. It is also highly recommended to encrypt Pod to Pod communications.

2 Kubernetes Authentication and Authorization Process

Every request that arrives at the Kubernetes API server must first be validated to ensure that the requester is a valid Kubernetes user and he/she/it (in case the request is coming from a Kubernetes object) is allowed to make that request. If the user passes the authentication and authorization process and the request will need to persist a new object (e.g., a new Pod), then the request is passed to the "Admission Controllers" to ensure the request is not violating any existing policies [2].

In Kubernetes, both authentication and authorization methods are pluggable and configurable. A cluster admin can configure which authentication module(s) are used for authentication and authorization; this is demonstrated in [Figure 2] below:

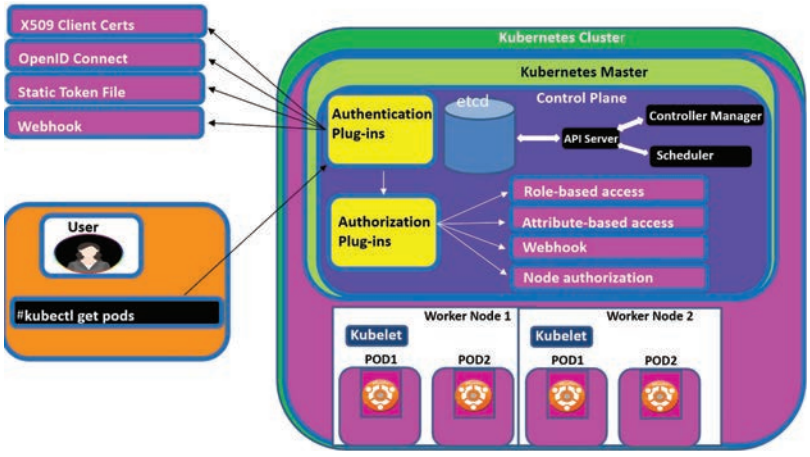

Figure 2 An illustration of Kubernetes pluggable authentication and authorization modules.

In the above picture, when our hypothetical user "Alice" issues "kubectl get pods" command, the kubectl utility sends Alice's credential along with the command to the API server. Which credential is sent, depends which authentication module the administrator has configured the API server to use. For instance, if "X509 Client Certificate" authentication is enabled then Alice's client cert is sent. If "OpenID connect" is enabled then kubectl sends her "id token" (which she got when she was authenticated by an OpenID Connect provider), in a header called "Authorization" to the API server [3].

Likewise, depending on which authorization module a Kubernetes administrator has enabled (Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC)), that particular module will decide if Alice is authorized to list pods. Since this is a read-only request, the request will not be evaluated by the Admission Controllers.

Notes:

1. "Node authorization" authorizes Kubelets API requests. It is not used for authorizing people or service accounts.
2. As alluded earlier, ABAC is another and older method of Kubernetes authorization. ABAC provides finer grain authorization than RBAC but it is more complex and generally it is recommended to use RBAC if possible.
3. Webhook [4] is a custom authorization enabler where authorization requests are intercepted and posted to a web service that performs the actual authorization and notifies Kubernetes to allow/disallow the request.

3 USERS, GROUPS, and Service Accounts in Kubernetes

Before diving into RBAC which is the main topic of this paper, it may be helpful to briefly discuss Kubernetes users, groups, and service accounts and how they are created.

- There are two types of users in Kubernetes: regular users (people), and service accounts.
- Service accounts are created and maintained by Kubernetes. Regular users, on the other hand, are managed by authentication providers, outside the Kubernetes environment. In Kubernetes, there is neither a user object to represent a regular user, nor an API to create/delete them.

3.1 Service Accounts

A service account is a type of non-human account that, in Kubernetes, provides a distinct identity for a service in a Kubernetes cluster. Application Pods, system components, and entities inside and outside the cluster can use specific ServiceAccount credentials to identify themselves as that ServiceAccount. This identity is useful in various situations, including authenticating to the API server or implementing identity-based security policies [5].

Service accounts can be created through yaml or command line:

```
#Yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: myservice-svc-acct

#Command line
kubectl create serviceaccount myservice-svc-acct
```

3.2 Regular Users and Groups

As stated earlier, Kubernetes supports a number of authentication providers such as X509 client certificate, OpenID Connect, and Webhooks. Each authentication provider manages its users and groups differently. X509 client certificate is the default authentication provider and is configured out of the box, so in this paper we will focus on how a Kubernetes admin can leverage client certificates to manage users and groups. Note that although X509 client certificate authentication is adequate for small to medium Kubernetes sites, for large companies, OpenID Connect is recommended [6].

```
#Generte a private key
Openssl genrsa -out john.doe.key 2048

#Generate a Certificate Signing Request (CSR)
#/CN (Common Name) is the username, and /O (Organization) is the group(s) the user belongs to. In this example, John Doe is our new users who will belong to “marketing-dev” and “hr-dev” groups.
openssl req -new -key john.doe.key -out john.doe.csr -subj
"/CN=john.doe/O=marketing-dev/O=hr-dev"

#Encode CSR (base64)
#And also have the header and trailer pulled out.
cat john.doe.csr |base64 |tr -d "\n" > john.doe.base64.csr
```

```
#Submit the CertificateSigningRequest to the API Server
#Key elements, name, request, signerName and usages (must be client auth)
cat <<EOF |kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: john.doe
spec:
  groups:
  - system:authenticated
  request: $(cat john.doe.base64.csr)
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - client auth
EOF

#View the CSRs, it will show as "pending" status
kubectl get certificatesigningrequests

#Approve the CSR (Note, you'll have one hour to approve it, otherwise it will
be garbage collected!)
kubectl certificate approve john.doe

#View the cert request and its status
kubectl get certificatesigningrequests john.doe

# Retrieve the certificate from the CSR object (it's base64 encoded), decode
it and save it to a file

kubectl get certificatesigningrequests john.doe \
-o jsonpath='{ .status.certificate }' |base64 --decode > john.doe.crt

#Once the client certificate is generated, the Kubernetes admin will use the
certificate to generate a config file for our new user "john.doe".

#Start by creating the "cluster" section of the config file
kubectl config set-cluster cluster1 \
  --server=https://<$MASTER_IP_ADDRESS>:6443 \
  --certificate-authority=/etc/kubernetes/pki/ca.crt \
  --embed-certs=true \
  --kubeconfig=john.doe.conf

#Add our "user" section of the config file
kubectl config set-credentials john.doe \
```

```
 --client-key=john.doe.key \
 --client-certificate=john.doe.crt \
 --embed-certs=true \
 --kubeconfig=john.doe.conf

#Create the context section
kubectl config set-context john.doe@cluster1 \
 --cluster=cluster1\
 --user=john.doe \
 --kubeconfig=john.doe.conf

#View the completed config for our user "john.do"
kubectl config view --kubeconfig=john.doe.conf
```

Finally, the admin will send the generated config file which has the client certificated embedded in it to user "John Doe". The user will copy the file to his home directory under the "./kube" folder. The embedded client certificate serves as the user's credentials and his user id and groups he belongs to will be available for authorization purposes.

4 RBAC Overview

RBAC is an essential part of Kubernetes security to safeguard its resources. As the [Figure 3] illustrates, it allows Kubernetes administrators to grant access to Kubernetes resource(s) to subjects(users), to perform select opeartions on those resource(s):

- Subjects: These include people, service accounts and other Kuberenetes objects such as kubelets idendified by their X509 client certificates.
- Kubernetes resources: Some resources such as "Persistentvolumes, Nodes, Storageclasses, Certificatesigningrequests" are cluster scoped, where as other resources such as "Pods, Services, Deployments, Daemonsets" are name space scoped.
- Opeartions/Verbs: These represent permisions such as "get, list, watch, create, patch, update, delete, deletecollection".

In order to create RBAC rules, we need to:

1. Determine where the resource is located within Kubernetes "API Groups" (we will cover API groups this in the next section).
2. Decide the scope of the rule (namespace/cluster). Kubernetes defines a set of objects to facilitate that, namely "Role" and "RoleBinding"

and "ClusterRole" and "Cluster RoleBinding". These will be covered in upcoming sections.

3. Determine appropriate set of permissions a subject will need on a given resource.

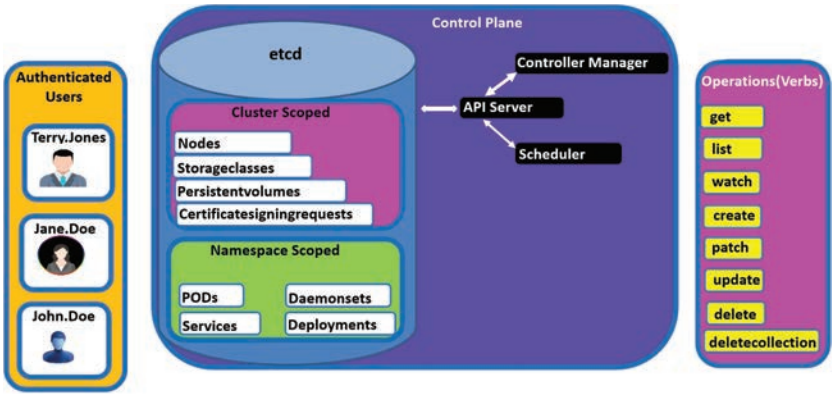

Figure 3 Overview of elements involved in RBAC decision making: Users, Resources, and Permissions.

5 Kubernetes API Groups and Resources

5.1 Kubernetes API Groups

The API server is implemented as a REST web service. You can examine the service's hierarchy by interacting with it through "kubectl proxy" which creates a proxy server between localhost and the API server:

#Establish the proxy. By default, it uses local port 8081 to interact with the API server.
kubectl proxy &

```
#View overall API hierarchy
$ curl http://localhost:8001/
  {
    "paths": [
      "/.well-known/openid-configuration",
      "/api",
      "/api/v1",
      "/apis",
      "/apis/",
```

```
    "/apis/apps",
    "/apis/apps/v1",
    "/apis/storage.k8s.io",
    "/apis/storage.k8s.io/v1",
    "/apis/storage.k8s.io/v1beta1",
    ------------------
  ]
}
```

(Listing 5.1.1)

As you can see from the above result (Listing 5.1.1), the API service is organized into different groups based on functionality and version. This is just a sample to illustrate the point, there are many other API groups that are not shown here due to space constraint.

The "core" API group "/api/v1" is the oldest group and contains many resources such as pods, namespaces, services, etc. If we execute the command below, we'll see what resources are available under the core API group and for each resource what verbs and other attributes are supported. For brevity, only the Pod resource listed below (Listing 5.1.2) but there are many other resources available in the core API group:

$ curl http://localhost:8001/api/v1

```
{
  "name": "pods",
  "singularName": "",
  "namespaced": true,
  "kind": "Pod",
  "verbs": [
   "create",
   "delete",
   "deletecollection",
   "get",
   "list",
   "patch",
   "update",
   "watch"
  ],
  "shortNames": [
   "po"
```

```
    ],
    "categories": [
     "all"
    ],
    "storageVersionHash": "xPOwRZ+Yhw8="
  }
```

(Listing 5.1.2)

Similarly, you can examine other API groups such as "/apis/apps/v1", etc.

5.2 Kubernetes Resources

To view all resource types and in which API groups they reside, we can use "kubectl api-resources" command:

$ kubectl api-resources -o wide #The "-o wide" switch provides more info

[Table 1] shows select output when the above command is executed. Only a few common resources shown for brevity and results are shown in a table format for better formatting:

Table 1

Name	Short Names	API Version	Name Spaced	Kind	Verbs
pods	po	v1	true	Pod	[create delete deletecollection get list patch update watch]
secrets		v1	true	Secrets	[create delete deletecollection get list patch update watch]
serviceaccounts	sa	v1	true	Service Account	[create delete deletecollection get list patch update watch]
services	svc	v1	true	Service	[create delete deletecollection get list patch update watch]

As we can see, it provides great info such as under which API group the resource lives, its scope namespace/cluster, and what verbs it supports.

Running the "kubectl api-resources" returns all resources but if say we are for looking for a specific resource, say "daemonsets", we could do something like:

kubectl api-resources -o wide |grep daemonsets

Table 2

Name	Short Names	API Version	Name Spaced	Kind	Verbs
daemonsets	ds	apps/v1	true	DaemonSet	[create delete deletecollection get list patch update watch]

#Result:

As it can be seen result shown in [Table 2], "daemonsets" resides under the "apps/v1" API group.

6 Kubernetes Role and RoleBinding

Let's say we want to assign a new user "Jane Doe" read-only access to all the Pods in the "Marketing" namespace. First, we need to define a "Role" in "marketing" namespace and assign Jane to that Role.

A Role is a namespaced object that represents a set of permissions assigned to one or more Kubernetes resource(s) that reside in one or more API group(s) as depicted in [Figure 4]:

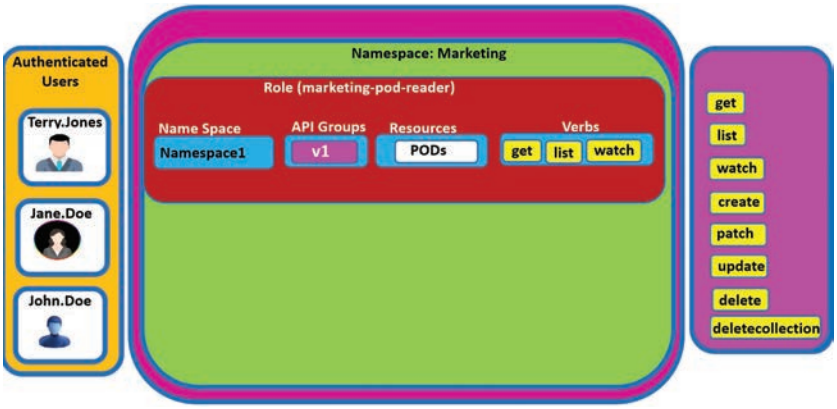

Figure 4 Role.

Here is the yaml representation:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: marketing
  name: marketing-pod-reader
```

```
rules:
- apiGroups: [""] # "" represents the "core" or "v1" API group.
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

(Listing 6.1)

Notes:

1. This Role (Listing 6.1) is bound to the "Marketing" namespace. In other words, RBAC enforces this rule only in the "Marketing" namespace.
2. We can specify multiple commas separated resources, for example to define a role that includes both "pods" and "daemonsets", we could express "resources" as ["pods"," daemonsets"]. Since "daemonsets" are in the "apps/v1" API group (see Section 5.2), we also need to add "apps" to the "apiGroup":

   ```
   rules:
   - apiGroups: ["",**"apps**"]
     resources: ["pods",**"daemonsets**"]
     verbs: ["get", "watch", "list"]
   ```

3. We can also set RBAC policy to a specific resource, for instance the "marketing-pod-reader" Role (Listing 6.1) gives read-only access to all the Pods in the marketing namespace but we could limit the scope to specific Pod("inventory-pod" in this case):

   ```
   rules:
   - apiGroups: [""]
     resources: ["pods"]
     resourceNames: ["inventory-pod"]
     verbs: ["get", "watch", "list"]
   ```

To assign(bind) user "jane.doe" to the newly created "marketing-pod-reader", we need to create a "RoleBinding" object:

Here is the yaml representation:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: marketing-pod-reader-binding
  namespace: marketing
subjects:
```

```
- kind: User
  name: jane.doe
  apiGroup: rbac.authorization.k8s.io
roleRef: # Signifies the binding to a Role / ClusterRole
  kind: Role # In this case we are binding to a "Role".
  name: marketing-pod-reader #Must match the name of the Role or Cluster-
Role to bind to
  apiGroup: rbac.authorization.k8s.io
```

(Listing 6.2)

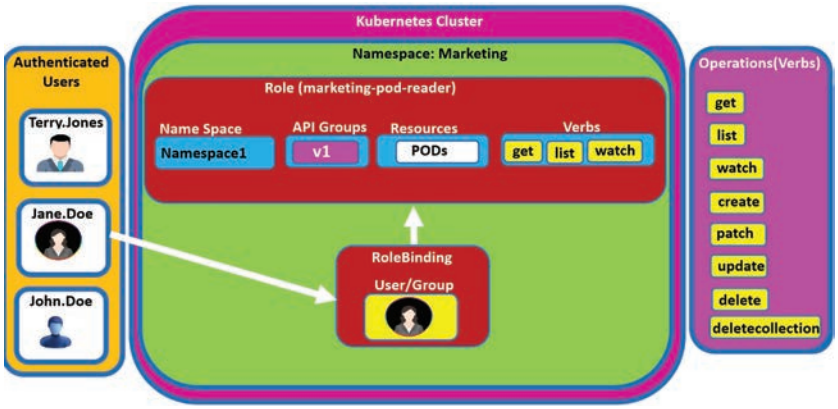

Figure 5 RoleBinding.

Notes:

1. "RoleBinding" is namespace scoped. In this example (Listing 6.2), it is bound to the "marketing" namespace.
2. Under "subjects", we can either specify specific user(s) by setting "-kind: User" or group(s) by specifying "-kind: Group". Groups could be the group specified in the "Subject" of a client certificate (if client certs are used) or the groups the user belongs to in LDAP/Open ID Connect systems.
3. Subjects can also be Kubernetes service accounts/group of service accounts as well, for example:

```
# "marketing-svc1" service account in the marketing name space
subjects:
- kind: ServiceAccount
```

```
  name: marketing-svc1
  namespace: marketing
  apiGroup: rbac.authorization.k8s.io

# All service accounts in the marketing name space
subjects:
- kind: Group
  name: system:serviceaccounts:marketing
  apiGroup: rbac.authorization.k8s.io
```

7 ClusterRole and ClusterRoleBinding

RBAC Role and Role Bindings can also be defined at cluster level. They are called "ClusterRole" and "ClusterRoleBinding" respectively. This means these roles and their corresponding role bindings are not bound by namespaces. [Figure 6] illustrates a POD read -only "ClusterRole":

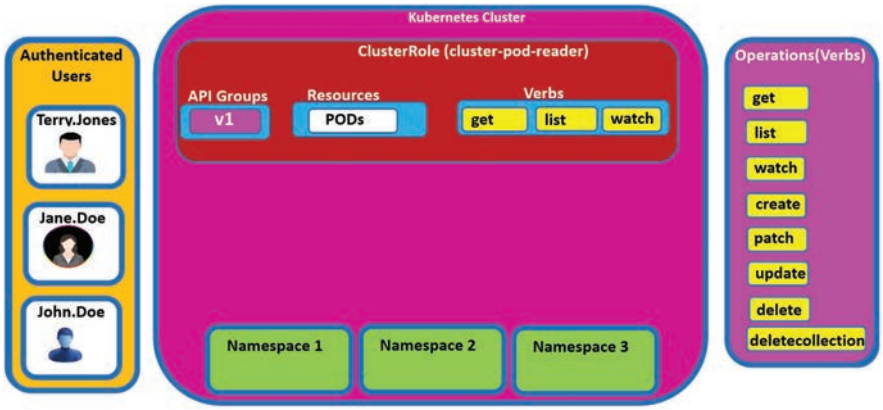

Figure 6 ClusterRoleBinding.

As you can see, unlike "Role", a "ClusterRole" does not have a "namespace" element.

Here is the yaml representation of our read only POD ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
metadata:
  name: cluster-pod-reader
rules:
- apiGroups: [""] # "" represents the "core" or "v1" API group.
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```
(Listing 7.1)

Once a ClusterRole has been created, we can bind users/groups to it. Here is the yaml representation of our ClusterRoleBinding object:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding
subjects:
- kind: User
  name: terry.jones
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-reader #must match the name of the Role or ClusterRole
to bind to
  apiGroup: rbac.authorization.k8s.io
```
(Listing 7.2)

Notes:

1. In this example (Listing 7.2), "Terry.Jones" will have read access to all Pods regardless of the namespaces they are bound to.

8 ClusterRole and Role Binding (Namespaced)

Often times it is desirable to define and reuse generic roles such as "Reader", "Developer," "Team Lead", etc. and bind them to namespace scoped role bindings. In other words, rather than creating a "Marketing Pod Reader" Role, and an "Accounting Pod Reader" Role, then bind them to RoleBinding binding in each namespace, we could create a "Pod Reader" ClusterRole and

bind it to namespace scoped RoleBinding. This is illustrated in [Figure 7] below:

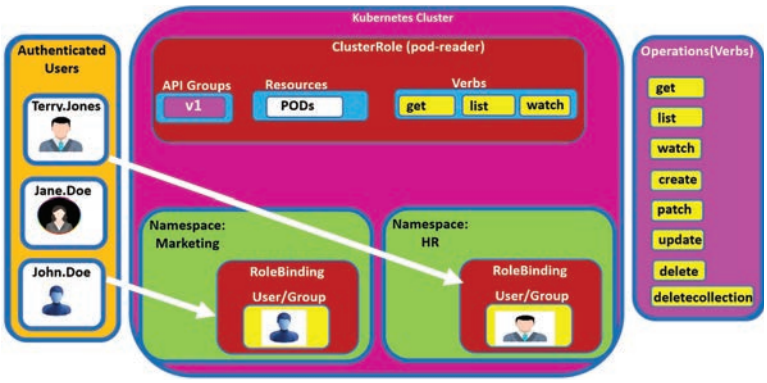

Figure 7 ClusterRole bound to a namespaced scoped Role.

In the above example, the ClusterRole "pod-reader" is defined at the cluster level and is being shared and bound to the "Marketing" and "HR" namespaces. "Terry.Jones" and "John.Doe" have Pod read-only access in the "HR" and "Marketing" namespaces respectively.

Here is the yaml representation:

#ClusterRole definition

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

(Listing 8.1)

#HR RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: hr-pod-reader-binding
  namespace: hr
subjects:
- kind: User
```

```
  name: terry.jones
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-reader-role
  apiGroup: rbac.authorization.k8s.io
```

(Listing 8.2)

```
#Marketing RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: marketing-pod-reader-binding
  namespace: marketing
subjects:
- kind: User
  name: john.doe
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-reader-role
  apiGroup: rbac.authorization.k8s.io
```

(Listing 8.3)

9 Aggregating ClusterRoles

Kubernetes RBAC allows aggregating two or more ClusterRoles into a single ClusterRole. Let's look at the example below:

The ClusterRole "support" (Listing 9.1) inherits all its rules from "manage-pods", manage-endpoints-services, manage-deployments, and "manage-daemonsets" ClusterRoles. As you can see its "rules:" section is empty. The "aggregationRule" will cause the control plane to automatically populate its "rules" with the rules from other ClusterRoles that have a matching label. As shown below (Listing 9.1), the other ClusterRoles that follow the "support" ClusterRole all have the matching "'acme.com/aggregate-to-support: "true"' label.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
metadata:
  name: support
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      acme.com/aggregate-to-support: "true"
rules: [] # The API Server fills in the rules form other ClusterRoles with matching label.
```

(Listing 9.1)

#If we run the following command, after creating the other ClusterRoles below, we'll see all the inherited rules (Listing 9.2):

kubectl describe ClusterRole support

PolicyRule:

Resources	Non-Resource URLs	Resource Names	Verbs
deployments.apps	[]	[]	[get list watch create update patch delete collection]
endpoints	[]	[]	[get list watch create update patch delete]
pods/logs	[]	[]	[get list watch create update patch delete]
pods	[]	[]	[get list watch create update patch delete]
services	[]	[]	[get list watch create update patch delete]
daemonsets.apps	[]	[]	[get list watch create update patch delete]

(Listing 9.2)

#Below (Listing 9.3 through 9.6) are the yaml declaration of the ClusterRoles that the "support" ClusterRole (Listing 8.1) inherited rules from:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: manage-pods
  labels:
    acme.com/aggregate-to-support: "true"
rules:
- apiGroups: [""]
  resources: ["pods","pods/logs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

(Listing 9.3)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: manage-endpoints-services
  labels:
    acme.com/aggregate-to-support: "true"
rules:
- apiGroups: [""]
  resources: ["endpoints","services"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

(Listing 9.4)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: manage-deployments
  labels:
    acme.com/aggregate-to-support: "true"
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

(Listing 9.5)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: manage-daemonsets
  labels:
    acme.com/aggregate-to-support: "true"
rules:
- apiGroups: ["apps"]
  resources: ["daemonsets"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

(Listing 9.6)

10 User Facing Built-in ClusterRoles

In addition to being able to create custom Role and ClusterRoles, Kubernetes provides some built-in ClusterRoles as described in [Table 3] below. Special Care must be taken when binding users/groups to these ClusterRoles, particularly the “cluster-admin” ClusterRole.

Table 3 Built-in User-facing roles [7].

ClusterRole	Description
cluster-admin	Allows super-user access to perform any action on any resource. When used in a ClusterRoleBinding, it gives full control over every resource in the cluster and in all namespaces. When used in a RoleBinding, it gives full control over every resource in the role binding's namespace, including the namespace itself.
admin	Allows admin access, intended to be granted within a namespace using a RoleBinding. If used in a RoleBinding, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself. This role also does not allow write access to Endpoints in clusters created using Kubernetes v1.22+.
edit	Allows read/write access to most objects in a namespace. This role does not allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. This role also does not allow write access to Endpoints in clusters created using Kubernetes v1.22+.
view	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. This role does not allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation).

11 Validating RBAC Rules Through Impersonation

The principle of least privilege (POLP) requires giving each user, service and application only the permissions needed to perform their work and no more [8].

When granting access to Kubernetes resources, administrators must ensure that the principal of least privilege is observed. This can be achieved through "impersonation". That is an administrator can impersonate the subject he/she is granting access to and validate the outcome.

A user (if granted that privilege), can act as another person through impersonation headers [9]. Impersonation can be achieved through Kubernetes API REST calls or through the "kubectl" utility. For this discussion we will use kubectl.

To see this in action, recall that in Section 6 of this paper we created a ClusterRole named "pod-reader-role" that gives read access to Pods. We also created a RoleBinding named "marketing-pod-reader-binding" in the

"Marketing" namespace with "John.Doe" as the subject. As Kubernetes admins we want to ensure that John has the intended rights which is "John Doe has read-only access to all Pods in the 'Marketing' namespace", nothing more and nothing less:

```
#Rather than literally executing commands on behalf of Terry, we will use the convenient
#"auth can-i" command to assess permissions. '--as=john.doe' impersonates john:

#Can John list Pods in the Marketing namespace? The answer should be yes
kubectl auth can-i list pods -n marketing --as=john.doe
yes # Kubernetes response

#Can John list Pods in the "kube-system" namespace? The answer should no.
#Even though "pod-reader-role" is a ClusterRole, it is bound to a RoleBinding in
#Marketing namespace. This limits the scope of John's rights only to the Marketing
#namespace
kubectl auth can-i list pods -n kube-system --as=john.doe
no # Kubernetes response

#Can John list Pods at the cluster level? The answer should be no, his rights are limited to the Marketing namespace.
kubectl auth can-i list pods --as=john.doe
no # Kubernetes response

#Can John create deployments in the Marketing namespace? The answer should be no
kubectl auth can-i create deployments -n marketing --as=john.doe
no #Kubernetes response
```

12 Conclusion

Kubernetes has become the facto standard for hosting modern cloud-native applications. It provides many benefits such as application load balancing, high availability, ease of deployment, ease of scale up/down based on business demands, etc. It has a layered architecture. Each layer has its own security requirements and settings. This layered architecture provides defence in depth which is highly effective and desirable, provided that these layers are properly protected. The focus of this paper of was protecting Kubernetes

the control plane layer (more specifically the objects in the etcd database) through Role Based Access Control (RBAC). Through RBAC we can exercise fine grain access control on all Kubernetes objects such as PODs, name spaces, secrets, deployments, end points, etc. RBAC enables us to create roles (permission sets on Kubernetes objects) and bind them to Kubernetes users, allowing users to perform specific operations such as list, create, delete, etc. Roles can be defined at the name space (Role) or cluster level (ClusterRole). Users can be bound to Roles at the name space (RobeBinding) level, or to ClusterRoles at the cluster level (ClusterRoleBinding). ClusterRoles can also be bound to users at the name space level through RoleBinding. Using this technique, Kubernetes admins can create generic roles such as "Developers, Team Leads", "Deployment", "Readers", etc. scoped at the cluster and bind them to users at individual name spaces. This will provide great reusability and reduces administrative burden of managing Roles at individual name space. RBAC also provides some built-in ClusterRoles such as "cluster-admin", "admin", "edit", and "view" that Kubernetes admins can leverage as needed (see Table 3). Kubernetes RBAC also provides role aggregation feature where super ClusterRoles can be created to automatically inherit rights from one or more ClusterRoles. Finally, impersonation allows Kubernetes admins to impersonate users to ensure they have just the right level of access to Kubernetes object to do their jobs (no more, no less).

References

[1] Role-based access control (Wiki):
https://en.wikipedia.org/wiki/Role-based_access_control.
[2] Using Admission Controllers (Kubernetes documentation):
https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers.
[3] OpenID Connect Tokens (Kubernetes documentation):
https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens.
[4] Webhook Mode (Kubernetes documentation):
https://kubernetes.io/docs/reference/access-authn-authz/webhook/.
[5] Kubernetes Service Accounts
https://kubernetes.io/docs/concepts/security/service-accounts/.
[6] Configure Kubernetes to use OpenID Connect Authentication
https://youtu.be/M9KABid_sCY.
[7] User-facing roles (Kubernetes documentation):

https://kubernetes.io/docs/reference/access-authn-authz/rbac/#user-facing-roles.
[8] Best Practice Guide to Implementing the Least privilege (Netwrix): https://www.netwrix.com/guide_to_implementing_the_least_privilege_principle.
[9] User impersonation (Kubernetes documentation): https://kubernetes.io/docs/reference/access-authn-authz/authentication/#user-impersonation.

Biography

Garsha Rostami is the CEO of the Galaxy Consulting L.L.C. in Minnesota, USA which provides custom Kubernetes training materials for clients. He also owns and manages the technology focused The Learning Channel on YouTube. He received his bachelor of science in Computer Science from university of New Brunswick in Fredericton, Canada. He has been in the computing business for the past 30 years and has worked for a variety of private and public companies including Target Corporation in Minneapolis where he was a Principal engineer.