JTC17 Kubernetes Security Advanced

Lab0 - Lab information

With Kubernetes rapidly taking over the IT world it is key that the Kubernetes operator and developer knows about basic Kubernetes security concepts.

This Lab provides you with a hands-on with several of those topics.

Lab sources

All the source code for the lab is available here:

https://github.com/niklaushirt/training

Lab overview

In this Lab you will learn about some basic Kubernetes security paradigms.

1. Role Based Acces Control (RBAC)
2. Service Accounts
3. Security Tooling
4. Image scanning

Lab0 - Lab semantics

Nomenclatures

Shell Commands

The commands that you are going to execute to progress the Labs will look like this:

THIS IS AN EXAMPLE - DO NOT EXECUTE THIS!

```bash
kubectl create -f redis-slave-service.yaml

 > Output Line 1
 > Output Line 2
 > Output Line 3
 ...
```

> **IMPORTANT NOTE:** The example output of a command is prefixed by ">" in order to make it more distinguishable.
>
> So in the above example you would only enter/copy-paste `kubectl create -f redis-slave-service.yaml` and the output from the command is "Output Line 1" to "Output Line 3"

Code Examples

Code examples are presented like this:

```yaml
apiVersion: lab.ibm.com/v1beta1
kind: MyResource
metadata:
  name: example
spec:
  size: 3
  image: busybox
```

This is only for illustration and is not being actively used in the Labs.

Lab 0 - Prepare the Lab environment

Before starting the Labs, let's make sure that we have the latest source code from the GitHub repository:

https://github.com/niklaushirt/training

1. Open a Terminal window by clicking on the Termnial icon in the left sidebar - we will use this extensively later as well

2. Execute the following commands to initialize your Training Environment

   ```bash
   ./welcome.sh
   ```

   This will

   - pull the latest example code from my GitHub repository
   - start minikube if not already running
   - installs the registry
   - installs the Network Plugin (Cilium)
   - starts the Personal Training Environment

   > During this you will have to provide a name (your name) that will be used to show your progress in the Instructor Dashboard in order to better assist you.

3. Start the demo application

   ```bash
   kubectl create -f ~/training/deployment/demoapp.yaml
   kubectl create -f ~/training/deployment/demoapp-service.yaml
   kubectl create -f ~/training/deployment/demoapp-backend.yaml
   kubectl create -f ~/training/deployment/demoapp-backend-service.yaml
   ```

4. Wait for the demo application to be available (the status must be 1/1)

   ```bash
   kubectl get pods

   > NAME                                READY   STATUS    RESTARTS   AGE
   > k8sdemo-backend-5b779f567f-2rbgj   1/1     Running   0          21s
   > k8sdemo-backend-5b779f567f-p6j76   1/1     Running   0          21s
   > k8sdemo-bd6bbd548-jcb6r            1/1     Running   0          21s
   ```

5. Open the demo application in the browser

```bash
minikube service k8sdemo-service

> |-----------|-----------------|-------------|-------------------------|
> | NAMESPACE |      NAME       | TARGET PORT |           URL           |
> |-----------|-----------------|-------------|-------------------------|
> | default   | k8sdemo-service |             | http://10.0.2.15:32123 |
> |-----------|-----------------|-------------|-------------------------|
> 🎉  Opening kubernetes service  default/k8sdemo-service in default browser...
```

> If you get the following error
>
> ```yaml
> 💣  Error getting machine status: load: filestore "minikube": open /home/training/.minikube/machines/minikube/config.json: permission denied
>
> 😿  Sorry that minikube crashed. If this was unexpected, we would love to hear from you:
> 👉  https://github.com/kubernetes/minikube/issues/new/choose
> ```
>
> Please execute:
>
> ```
> ~/training/tools/own.sh
> ```

Lab 1 - RBAC

RBAC policies are vital for the correct management of your cluster, as they allow you to specify which types of actions are permitted depending on the user and their role in your organization. Examples include:

Secure your cluster by granting privileged operations (accessing secrets, for example) only to admin users. Force user authentication in your cluster. Limit resource creation (such as pods, persistent volumes, deployments) to specific namespaces. You can also use quotas to ensure that resource usage is limited and under control. Have a user only see resources in their authorized namespace. This allows you to isolate resources within your organization (for example, between departments).

RBAC Roles

Rbac Roles are composed of

- **RBAC API objects**
    - Pods
    - PersistentVolumes
    - ConfigMaps
    - Deployments
    - Nodes
    - Secrets
    - Namespaces
- **Possible operations** over these resources are:
    - create
    - get
    - delete
    - list
    - update
    - edit
    - watch
    - exec

RBAC Elements

- **Rules**: A rule is a set of operations (verbs) that can be carried out on a group of resources which belong to different API Groups.

- **Roles and ClusterRoles**: Both consist of rules. The difference between a Role and a ClusterRole is the scope: in a Role, the rules are applicable to a single namespace, whereas a ClusterRole is cluster-wide, so the rules are applicable to more than one namespace. ClusterRoles can define rules for cluster-scoped resources (such as nodes) as well. Both Roles and ClusterRoles are mapped as API Resources inside our cluster.

- **RoleBindings and ClusterRoleBindings**: Just as the names imply, these bind subjects to roles (i.e. the operations a given user can perform). As for Roles and ClusterRoles, the difference lies in the scope: a RoleBinding will make the rules effective inside a namespace, whereas a ClusterRoleBinding will make the rules effective in all namespaces.

- **Subjects**: These correspond to the entity that attempts an operation in the cluster. There are three types of subjects:

  - **User Accounts**: These are global, and meant for humans or processes living outside the cluster. There is no associated resource API Object in the Kubernetes cluster.
  - **Service Accounts**: This kind of account is namespaced and meant for intra-cluster processes running inside pods, which want to authenticate against the API.
  - **Groups**: This is used for referring to multiple accounts. There are some groups created by default such as cluster-admin (explained in later sections).

You can get more detailed information in the official Kubernetes documentation here.

Lab 1 - Users, Roles and RoleBindings

Lab 1 - Create user with limited namespace access

In this example, we will create a user with limited namespace access.

The following User Account will be created:

- Username: demo
- Group: demogroup

We will add the necessary RBAC policies so this user can fully manage deployments (i.e. use `kubectl run` command) only inside the `rbactest` namespace. At the end, we will test the policies to make sure they work as expected.

Create the rbactest namespace

- Execute the `kubectl create` command to create the namespace (as the admin user):

  ```bash
  kubectl create namespace rbactest
  ```

Lab 1 - Create the user credentials

Kubernetes does not have API Objects for User Accounts. Of the available ways to manage authentication (see Kubernetes official documentation for a complete list), we will use OpenSSL certificates for their simplicity.

Lab 1 - Create the certificate

1. Create a private key for your user. In this example, we will name the file `demo.key` .

   Go to a temporary working directory in your terminal.

   ```bash
   mkdir rbacdemo
   cd rbacdemo
   ```

   ```bash
   openssl genrsa -out demo.key 2048

   > Generating RSA private key, 2048 bit long modulus (2 primes)
   ...
   ```

   The private key file `demo.key` has been created:

   ```bash
   ls

   > demo.key
   ```

2. Create a certificate sign request `demo.csr` using the private key you just created (`demo.key` in this example). Make sure you specify your username and group in the *-subj* section (CN is for the username and O for the group). As previously mentioned, we will use `demo` as the name and `demogroup` as the group

   ```bash
   openssl req -new -key demo.key -out demo.csr -subj "/CN=demo/O=demogroup"
   ```

   The certificate sign request `demo.csr` has been created:

   ```bash
   ls

   > demo.csr  demo.key
   ```

3. We will use the Kubernetes cluster certificate authority (CA) for approving the request and generating

the necessary certificate to access the cluster API.

Its location usually is `/etc/kubernetes/pki/`. But in our case (for Minikube), it would be `~/.minikube/`.

4. Generate the final certificate `demo.crt` by approving the certificate sign request, `demo.csr`, you made earlier. In this example, the certificate will be valid for 500 days:

```bash
openssl x509 -req -in demo.csr -CA ~/.minikube/ca.crt -CAkey ~/.minikube/ca.key -CAcreateserial -out demo.crt -days 500

> key -CAcreateserial -out demo.crt -days 500
> Signature ok
> subject=CN = demo, O = demogroup
> Getting CA Private Key
```

The final certificate `demo.crt` has been created:

```bash
ls

> demo.crt  demo.csr  demo.key
```

5. In a real world example you would now save both `demo.crt` and `demo.key` in a safe location.

Lab 1 - Create the context

Add a new context with the new credentials for your Kubernetes cluster. This example is for a Minikube cluster but it should be similar for others:

1. Set credentials

```bash
kubectl config set-credentials demo --client-certificate=./demo.crt --client-key=./demo.key

> User "demo" set.
```

2. Create context

```bash
kubectl config set-context demo-context --cluster=minikube --namespace=rbactest --user=demo

> Context "demo-context" created.
```

3. Check configuration

```bash
kubectl config view
```

We can see that we now have a context `demo-context` and a user `demo` in our configuration that will be used to access the Kubernetes API via `kubectl`.

You should get an access denied error when using the `kubectl` CLI with this configuration file. This is expected as we have not defined any permitted operations for this user.

```bash
kubectl --context=demo-context get pods

> No resources found.
> Error from server (Forbidden): pods is forbidden: User "demo" cannot list resource "pods" in API group "" in the namespace "rbactest"
```

1) Create the role for viewing deployments

We are creating the `Rule` that allows a user to execute several `Read Only` operations on Deployments, Pods and ReplicaSets, which belong to the `core` (expressed by “” in the `yaml` file), `apps`, and `extensions` API Groups:

YAML

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: deployment-viewer
  namespace: rbactest
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list"] # You can also use ["*"]
```

Create the `Role` in the cluster using:

Bash

```bash
kubectl create -f ~/training/rbac/deployment-viewer.yaml
```

2) Bind the viewer role to the demo user

In this step we are creating the `RuleBinding` that binds the `deployment-viewer` `Role` to the User Account `demo` inside the `rbactest` namespace:

YAML

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: deployment-viewer-binding
  namespace: rbactest
subjects:
- kind: User
  name: demo
  apiGroup: ""
roleRef:
  kind: Role
  name: deployment-viewer
  apiGroup: ""
```

Create the Role in the cluster using:

```bash
kubectl create -f ~/training/rbac/deployment-viewer-binding.yaml
```

Lab 1 - Test the RBAC viewer rule

Now you should be able to execute the following command without any issues:

```bash
kubectl --context=demo-context get pods

> No resources found.
```

`No resources found.` simply means that we don't have any `Pods` deployed in this `Namespace`.

If you run the same command for the default namespace with the `--namespace=default` argument, it will fail, as the `demo` user does not have access to this namespace.

```bash
kubectl --context=demo-context get pods --namespace=default

> No resources found.
> Error from server (Forbidden): pods is forbidden: User "demo" cannot list resource "pods" in API group "" in the namespace "default"
```

Also you still don't have the rights to create or delete `Deployments`:

```bash
kubectl --context=demo-context run --image alpine alpine


> Error from server (Forbidden): deployments.apps is forbidden: User "demo" cannot create resource "deployments" in API group "apps" in the namespace "rbactest"
```

Now you have created a user with limited Read Only permissions in your cluster.

1) Create the role for managing deployments

We are creating the `Rule` that allows a user to execute several `Read and Write` operations on Deployments, Pods and ReplicaSets, which belong to the `core` (expressed by “” in the `yaml` file), `apps`, and `extensions` API Groups:

```YAML
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: deployment-manager
  namespace: rbactest
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You can also use ["*"]
```

Create the `Role` in the cluster using:

```Bash
kubectl create -f ~/training/rbac/deployment-manager.yaml
```

2) Bind the Manager role to the demo user

In this step we are creating the `RuleBinding` that binds the `deployment-manager` `Role` to the User Account `demo` inside the `rbactest` namespace:

```YAML
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: deployment-manager-binding
  namespace: rbactest
subjects:
- kind: User
  name: demo
  apiGroup: ""
roleRef:
  kind: Role
  name: deployment-manager
  apiGroup: ""
```

Create the Role in the cluster using:

```bash
kubectl create -f ~/training/rbac/deployment-manager-binding.yaml
```

Lab 1 - Test the RBAC manager rule

Now you should be able to execute the following commands without any issues:

```bash
kubectl --context=demo-context run --image alpine alpine


> deployment.apps/alpine created
```

Check that the alpine Pod is running

```bash
kubectl --context=demo-context get pods

> NAME                          READY     STATUS      RESTARTS    AGE
> alpine-7f866557df-dkmks       0/1       Completed   0           6s
```

However if you run the same command for the default namespace with the `--namespace=default` argument, it will still fail, as the `demo` user still does not have access to this namespace.

```bash
kubectl --context=demo-context get pods --namespace=default

> No resources found.
> Error from server (Forbidden): pods is forbidden: User "demo" cannot list resource "pods" in API group "" in the namespace "default"
```

Now you have created a user with limited permissions in your cluster but with full Management rights for Deployments in the `rbactest` namespace.

Congratulations!!!

This concludes Lab 1 on RBAC and Roles/RoleBindings.

Lab 2 - Service Accounts

Lab 2 - Create a ServiceAccount for a Deployment

In this chapter we will start this `Pod` with a limited `ServiceAccount`.

Create the resources

To create the `ServiceAccount`:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: service-account-1
  labels:
    app: tools-rbac
```

Run the following command:

```bash
kubectl apply -f ~/training/rbac/service-accounts.yaml

> serviceaccount "service-account-1"
```

Now we will create a Deployment that runs under the `ServiceAccount` that we have just created. The `Pod` contains the `kubectl` executable, so that we can test the access rights from withing this `Pod`.

To create the `Deployment`:

```YAML
kind: Deployment
metadata:
  name: kubectl
  labels:
    rbac: service-account-1
spec:
  replicas: 1
 ...
  template:
    metadata:
      labels:
        rbac: service-account-1
    spec:
      serviceAccountName: service-account-1
      containers:
        - name: kubectl
          image: "niklaushirt/kubectl:1.14"
...
```

Run the following command:

```Bash
kubectl apply -f ~/training/rbac/service-account-kubectl.yaml

> deployment.apps/kubectl configured
```

Great, now lets see how our pod is doing:

````Bash
kubectl get pods

> NAME                                     READY     STATUS         RESTARTS   AG
E
> alpine                                   1/1       Running       0          3h4
8m
> k8sdemo-7d46f69d68-d5dwm                 1/1       Running       0          4h6
m
> k8sdemo-backend-9c777544b-knnth          1/1       Running       0          4h2
m
> k8sdemo-backend-9c777544b-tztr8          1/1       Running       0          4h2
m
> k8sdemo-nok-7b4c444454-h6w6r             1/1       Running       0          3h3
0m
> kubectl-f8977f5d9-4mm69                  1/1       Running       0          25s
> tools-service-account-7c4c798b7-x7rkv    1/1       Running       0          28m
```
````
````

Test Access

Now test the access from inside the Pod (**you will have to replace the Pod name**):

```bash
kubectl exec kubectl-f8977f5d9-4mm69 kubectl get services

> Error from server (Forbidden): services is forbidden: User "system:serviceaccount:default:service-account-1" cannot list resource "services" in API group "" in the namespace "default"
> command terminated with exit code 1
```

So the access is forbidden for the Pod running under the `ServiceAccount` `service-account-1`, which makes sense, because the `ServiceAccount` has no rights assigned as of now.

Lab 2 - Add Role and RoleBinding for Service Account

We now are running the `kubectl` `Pod` under the `ServiceAccount` `service-account-1` .

The following configuration will create a `Role` and a `RoleBinding` for just this service account.

Create Role and RoleBinding

1. Create Role

   ```yaml
   kind: Role
   apiVersion: rbac.authorization.k8s.io/v1
   metadata:
     name: api-role
     namespace: default
     labels:
       app: tools-rbac
   rules:
   - apiGroups: [""]
     resources: ["services"]
     verbs: ["get", "list"]
   - apiGroups: [""]
     resources: ["configmaps"]
     verbs: ["create"]
   - apiGroups: [""]
     resources: ["configmaps"]
     resourceNames: ["mqtt-pub-address"]
     verbs: ["update", "delete"]
   ```

   The `Role` has the rights to list the `Services` .

   ```bash
   kubectl create -f ~/training/rbac/service-accounts-role.yaml

   > role.rbac.authorization.k8s.io/api-role configured
   ```

2. Now lets bind the Role to the `ServiceAccount` `service-account-1`

```YAML
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-account-rolebinding
  namespace: default
  labels:
    app: tools-rbac
subjects:
- kind: ServiceAccount
  name: service-account-1
roleRef:
  kind: Role
  name: api-role
  apiGroup: ""
```

Create the RoleBinding

```Bash
kubectl create -f ~/training/rbac/service-accounts-role-binding.yaml

> rolebinding.rbac.authorization.k8s.io/service-account-rolebinding created
```

Now the `ServiceAccount` `service-account-1` should have the rights to list the `Services`.

Test Access

Let's try this again:

```Bash
kubectl exec kubectl-f8977f5d9-4mm69 kubectl get services

> NAME                      TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
     AGE
> k8sdemo-backend-service   NodePort    10.109.88.37   <none>        3000:30601/T
CP    4h9m
> k8sdemo-service           NodePort    10.99.195.7    <none>        3000:30456/T
CP    4h11m
> kubernetes                ClusterIP   10.96.0.1      <none>        443/TCP
     5h20m
```

We can see, that the Pod running under the `ServiceAccount` `service-account-1` can now access the list of `Services` because it is bound to a `Role` that allows for listing them.

However this is still a **ReadOnly** access (the allowed `verbs` for Services being "get" and "list").

When trying to **modify** (delete) a `Service` we still get an error:

```bash
kubectl exec kubectl-f8977f5d9-4mm69 kubectl delete services k8sdemo-service

> Error from server (Forbidden): services "k8sdemo-service" is forbidden: User "system:serviceaccount:default:service-account-1" cannot delete resource "services" in API group "" in the namespace "default"
> command terminated with exit code 1
```

We have now created a RBAC Policy that gives a `ServiceAccount` specific rights (in this case ReadOnly) to certain ressources.

Congratulations!!!

This concludes Lab 3 on RBAC and Service Accounts.

Lab 3 - RBAC Tooling

Rakkess

Have you ever wondered what access rights you have on a provided kubernetes cluster? For single resources you can use kubectl auth can-i list deployments, but maybe you are looking for a complete overview? This is what rakkess is for. It lists access rights for the current user and all server resources.

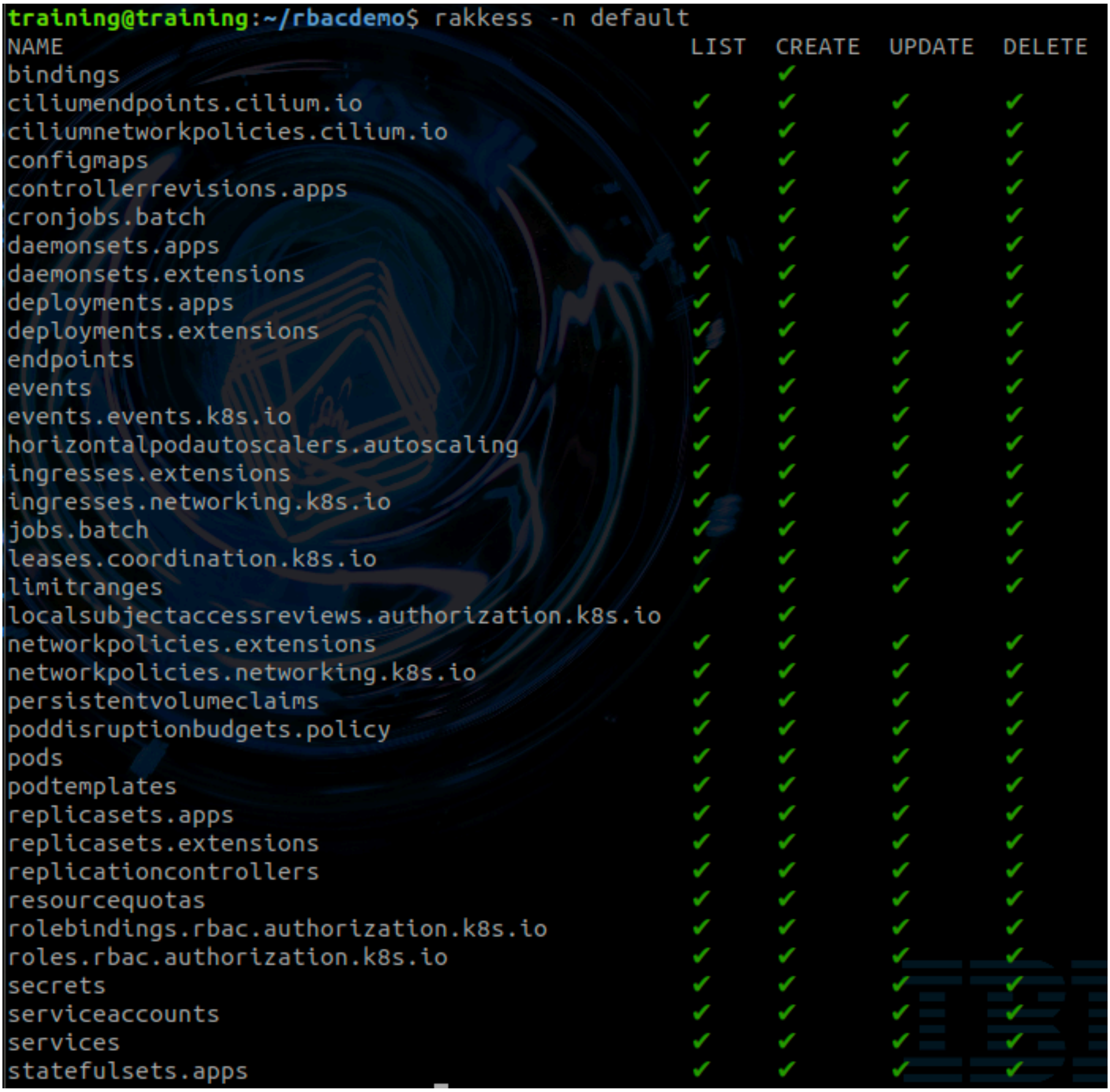
You can get more details here.

1. Install Rakkess

```bash
curl -LO https://github.com/corneliusweig/rakkess/releases/download/v0.4.4/rakkess-amd64-linux.tar.gz \
  && tar xf rakkess-amd64-linux.tar.gz rakkess-amd64-linux \
  && chmod +x rakkess-amd64-linux \
  && sudo mv -i rakkess-amd64-linux $GOPATH/bin/rakkess
```

2. Let's examine the RBAC for the default Namespace:

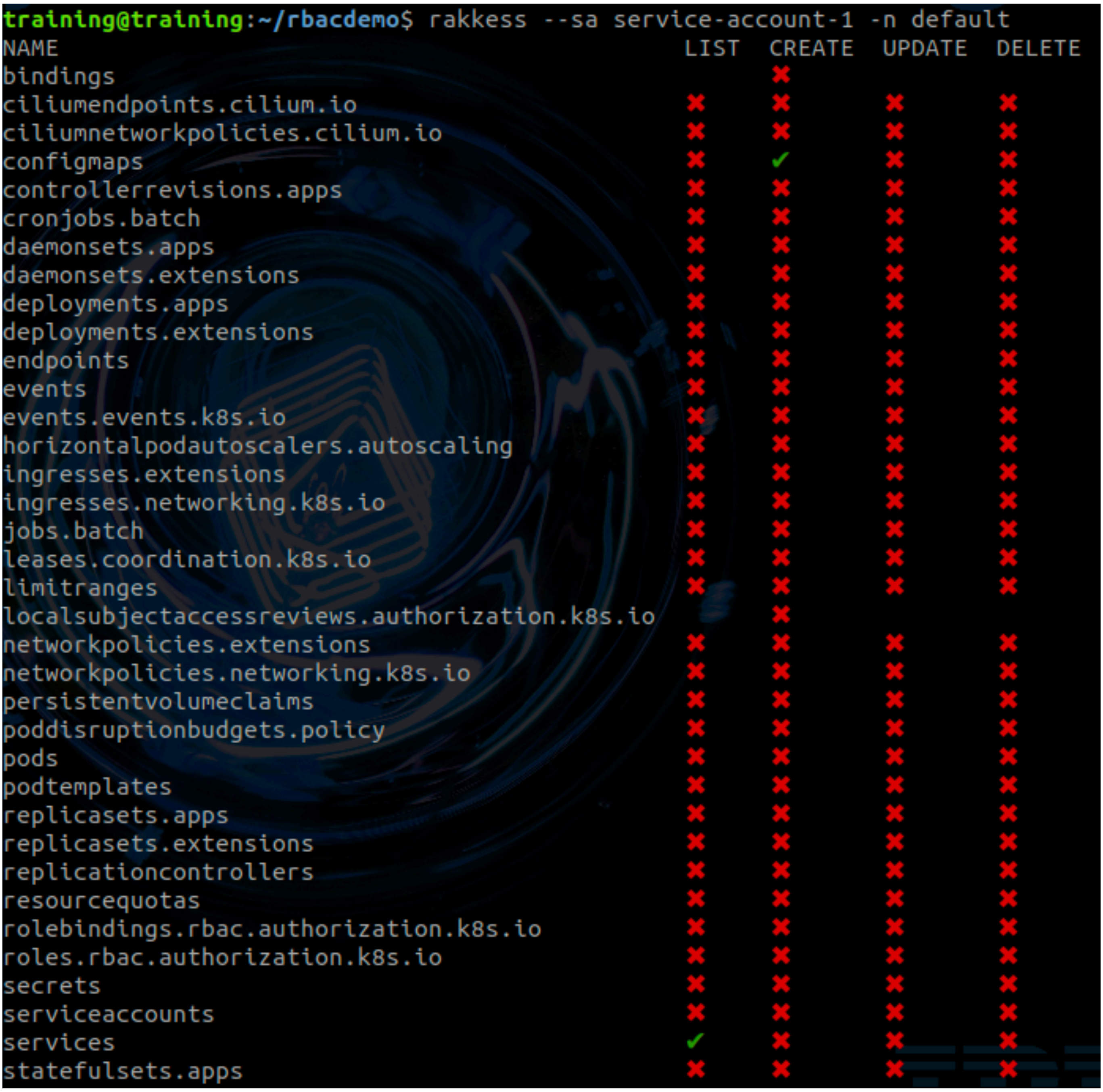
```bash
rakkess -n default
```



```
training@training:~/rbacdemo$ rakkess -n default
NAME                                            LIST  CREATE  UPDATE  DELETE
bindings                                              ✓
ciliumendpoints.cilium.io                       ✓     ✓       ✓       ✓
ciliumnetworkpolicies.cilium.io                 ✓     ✓       ✓       ✓
configmaps                                      ✓     ✓       ✓       ✓
controllerrevisions.apps                        ✓     ✓       ✓       ✓
cronjobs.batch                                  ✓     ✓       ✓       ✓
daemonsets.apps                                 ✓     ✓       ✓       ✓
daemonsets.extensions                           ✓     ✓       ✓       ✓
deployments.apps                                ✓     ✓       ✓       ✓
deployments.extensions                          ✓     ✓       ✓       ✓
endpoints                                       ✓     ✓       ✓       ✓
events                                          ✓     ✓       ✓       ✓
events.events.k8s.io                            ✓     ✓       ✓       ✓
horizontalpodautoscalers.autoscaling            ✓     ✓       ✓       ✓
ingresses.extensions                            ✓     ✓       ✓       ✓
ingresses.networking.k8s.io                     ✓     ✓       ✓       ✓
jobs.batch                                      ✓     ✓       ✓       ✓
leases.coordination.k8s.io                      ✓     ✓       ✓       ✓
limitranges                                     ✓     ✓       ✓       ✓
localsubjectaccessreviews.authorization.k8s.io        ✓
networkpolicies.extensions                      ✓     ✓       ✓       ✓
networkpolicies.networking.k8s.io               ✓     ✓       ✓       ✓
persistentvolumeclaims                          ✓     ✓       ✓       ✓
poddisruptionbudgets.policy                     ✓     ✓       ✓       ✓
pods                                            ✓     ✓       ✓       ✓
podtemplates                                    ✓     ✓       ✓       ✓
replicasets.apps                                ✓     ✓       ✓       ✓
replicasets.extensions                          ✓     ✓       ✓       ✓
replicationcontrollers                          ✓     ✓       ✓       ✓
resourcequotas                                  ✓     ✓       ✓       ✓
rolebindings.rbac.authorization.k8s.io          ✓     ✓       ✓       ✓
roles.rbac.authorization.k8s.io                 ✓     ✓       ✓       ✓
secrets                                         ✓     ✓       ✓       ✓
serviceaccounts                                 ✓     ✓       ✓       ✓
services                                        ✓     ✓       ✓       ✓
statefulsets.apps                               ✓     ✓       ✓       ✓
```

3. And now the RBAC for the ServiceAccount that we have created earlier:

```Bash
rakkess --sa service-account-1 -n default
```



```
training@training:~/rbacdemo$ rakkess --sa service-account-1 -n default
NAME                                            LIST  CREATE  UPDATE  DELETE
bindings                                               ✗
ciliumendpoints.cilium.io                        ✗     ✗       ✗       ✗
ciliumnetworkpolicies.cilium.io                  ✗     ✗       ✗       ✗
configmaps                                       ✗     ✓       ✗       ✗
controllerrevisions.apps                         ✗     ✗       ✗       ✗
cronjobs.batch                                   ✗     ✗       ✗       ✗
daemonsets.apps                                  ✗     ✗       ✗       ✗
daemonsets.extensions                            ✗     ✗       ✗       ✗
deployments.apps                                 ✗     ✗       ✗       ✗
deployments.extensions                           ✗     ✗       ✗       ✗
endpoints                                        ✗     ✗       ✗       ✗
events                                           ✗     ✗       ✗       ✗
events.events.k8s.io                             ✗     ✗       ✗       ✗
horizontalpodautoscalers.autoscaling             ✗     ✗       ✗       ✗
ingresses.extensions                             ✗     ✗       ✗       ✗
ingresses.networking.k8s.io                      ✗     ✗       ✗       ✗
jobs.batch                                       ✗     ✗       ✗       ✗
leases.coordination.k8s.io                       ✗     ✗       ✗       ✗
limitranges                                      ✗     ✗       ✗       ✗
localsubjectaccessreviews.authorization.k8s.io         ✗
networkpolicies.extensions                       ✗     ✗       ✗       ✗
networkpolicies.networking.k8s.io                ✗     ✗       ✗       ✗
persistentvolumeclaims                           ✗     ✗       ✗       ✗
poddisruptionbudgets.policy                      ✗     ✗       ✗       ✗
pods                                             ✗     ✗       ✗       ✗
podtemplates                                     ✗     ✗       ✗       ✗
replicasets.apps                                 ✗     ✗       ✗       ✗
replicasets.extensions                           ✗     ✗       ✗       ✗
replicationcontrollers                           ✗     ✗       ✗       ✗
resourcequotas                                   ✗     ✗       ✗       ✗
rolebindings.rbac.authorization.k8s.io           ✗     ✗       ✗       ✗
roles.rbac.authorization.k8s.io                  ✗     ✗       ✗       ✗
secrets                                          ✗     ✗       ✗       ✗
serviceaccounts                                  ✗     ✗       ✗       ✗
services                                         ✓     ✗       ✗       ✗
statefulsets.apps                                ✗     ✗       ✗       ✗
```

We can see that the `ServcieAccount` has the rights to list `Services` and to create `ConfigMaps`. This corresponds to the `api-role` `Role` that we have defined earlier:

```yaml
...
- apiGroups: [""]
  resources: ["services"]
  verbs: ["get", "list"]
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["create"]
...
```

rbac-view

Polaris runs a variety of checks to ensure that Kubernetes pods and controllers are configured using best practices, helping you avoid problems in the future.

You can get more details here.

1. Install rbac-view

```bash
wget https://github.com/jasonrichardsmith/rbac-view/releases/download/v0.2.1/rbac-view.v0.2.1.linux.tar.gz
tar xf rbac-view.v0.2.1.linux.tar.gz
chmod +x ./bin/linux/rbac-view
sudo mv -i ./bin/linux/rbac-view $GOPATH/bin/rbac-view
```

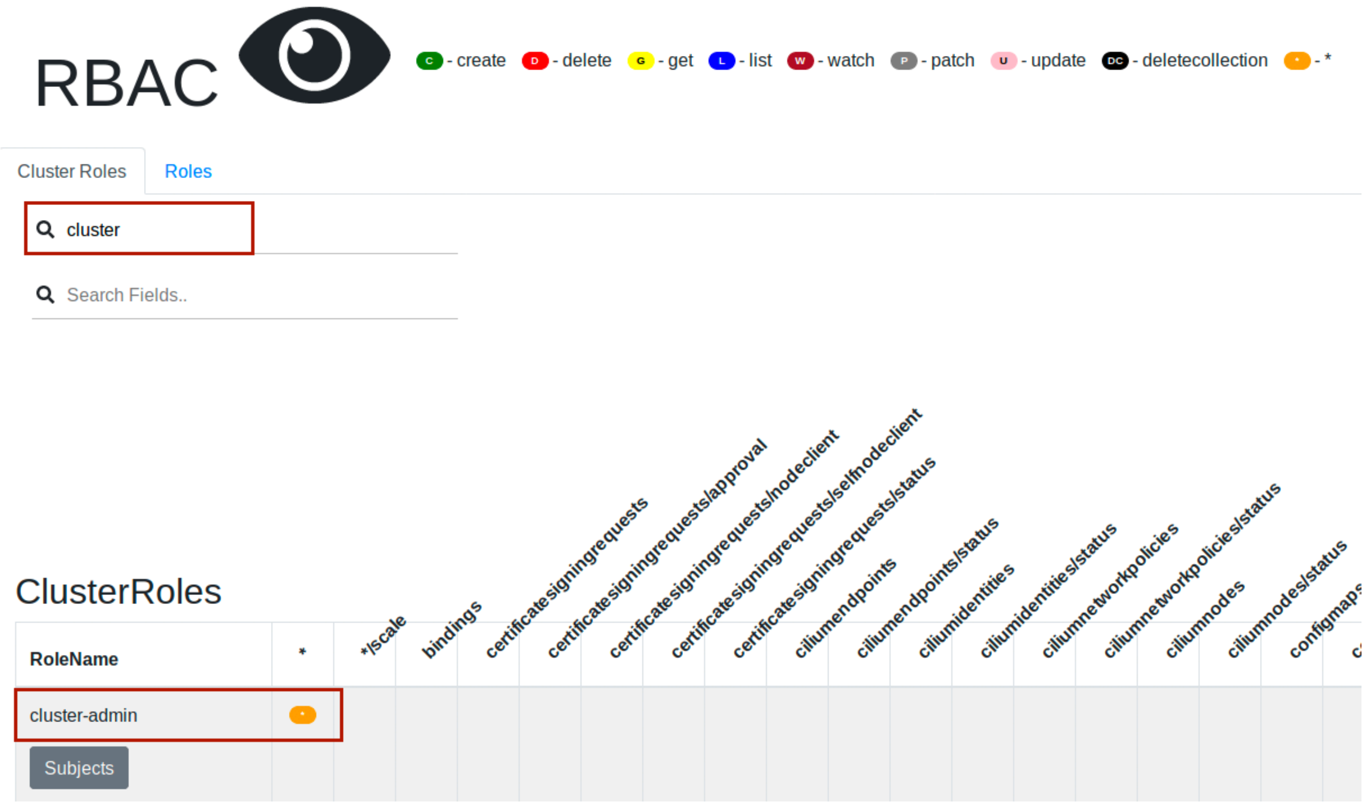
2. Run the following in your Terminal:

```bash
rbac-view

> INFO[0000] Getting K8s client
> INFO[0000] serving RBAC View and http://localhost:8800
> INFO[0039] Building full matrix for json
> INFO[0039] Building Matrix for Roles
> INFO[0039] Retrieving RoleBindings
> INFO[0039] Building Matrix for ClusterRoles
> INFO[0039] Retrieving ClusterRoleBindings
> INFO[0039] Retrieved 49 ClusterRoleBindings
> INFO[0039] Retrieving ClusterRole system:volume-scheduler
> INFO[0039] Retrieving ClusterRole system:controller:horizontal-pod-autoscaler
> INFO[0039] Retrieving ClusterRole system:controller:generic-garbage-collector
> INFO[0039] Retrieving ClusterRole system:controller:job-controller
> INFO[0039] Retrieving ClusterRole cilium
...
> INFO[0048] Retrieving Role system:controller:bootstrap-signer in namespace kube-public
> INFO[0048] Retrieving Role kubernetes-dashboard in namespace kubernetes-dashboard
> INFO[0048] Retrieving Role deployment-manager in namespace rbactest
> INFO[0051] Built Matrix for Roles
> INFO[0051] Matrix for json built
```

This takes some time and when you see Matrix for json built you can start the browser at http://:8800 or directly use the bookmark.

3. Enter `cluster` in the `SearchRoles` field and examine the `cluster-admin` `Role`. The `*` means that it has all access rights to all ressources with all verbs.



RBAC

C - create D - delete G - get L - list W - watch P - patch U - update DC - deletecollection * - *

Cluster Roles Roles

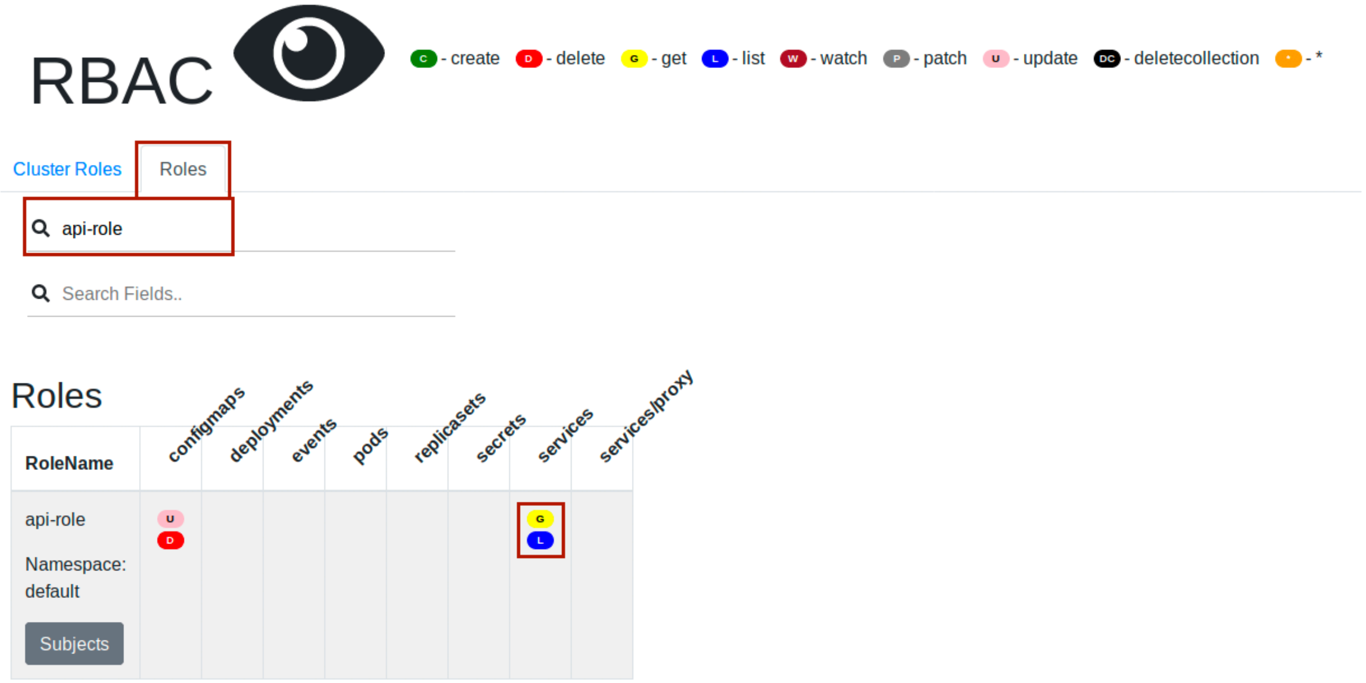
cluster

Search Fields..

ClusterRoles

RoleName	*	*/scale	bindings	certificatesigningrequests	certificatesigningrequests/approval	certificatesigningrequests/nodeclient	certificatesigningrequests/selfnodeclient	certificatesigningrequests/status	ciliumendpoints	ciliumendpoints/status	ciliumidentities	ciliumidentities/status	ciliumnetworkpolicies	ciliumnetworkpolicies/status	ciliumnodes	ciliumnodes/status	configmaps	c
cluster-admin Subjects	*																	

4. Switch to the Roles tab and enter `api-role` in the `SearchRoles` field and examine it. We can see the access rights previously discussed, notably `get` and `list` rights to `Services`.



RBAC

C - create D - delete G - get L - list W - watch P - patch U - update DC - deletecollection * - *

Cluster Roles Roles

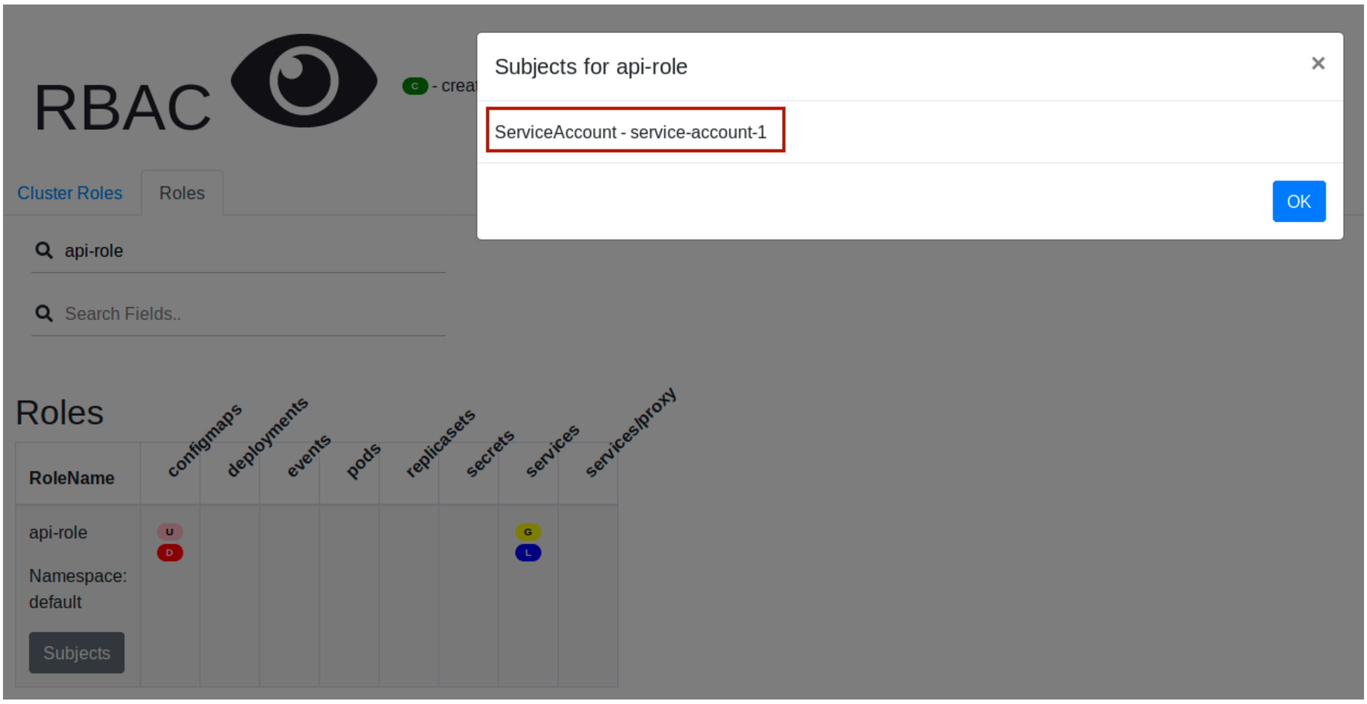
api-role

Search Fields..

Roles

RoleName	configmaps	deployments	events	pods	replicasets	secrets	services	services/proxy
api-role Namespace: default Subjects	U D						G L	

5. When you click on the Subjects button you will get the list of Subjects for this `Role`. In this cas we can see the `ServiceAccount` `service-account-1` that is a subject of the Role.



Subjects for api-role

ServiceAccount - service-account-1

api-role

Roles

RoleName
api-role Namespace: default

Congratulations!!!

This concludes Lab 4 on RBAC Tooling.

Lab 4 - Image Scanning

Lab 4 - Deploy Clair

In this chapter we will deploy the Clair image scanner and scan an example image.

Clair is an open source project for the static analysis of vulnerabilities in application containers (currently including appc and docker).

1. In regular intervals, Clair ingests vulnerability metadata from a configured set of sources and stores it in the database.
2. Clients use the Clair API to index their container images; this creates a list of features present in the image and stores them in the database.
3. Clients use the Clair API to query the database for vulnerabilities of a particular image; correlating vulnerabilities and features is done for each request, avoiding the need to rescan images.
4. When updates to vulnerability metadata occur, a notification can be sent to alert systems that a change has occurred.

To make things easier we will use the Klar command line tool to interact with the Clair engine.

Install the Klar command line tool

```bash
wget https://github.com/optiopay/klar/releases/download/v2.4.0/klar-2.4.0-linux-amd64
sudo chmod +x klar-2.4.0-linux-amd64
sudo mv klar-2.4.0-linux-amd64 /usr/local/bin/klar
```

Deploy Clair into the Kubernetes Cluster

Make sure that you have executed the following commands to pull the latest example code from my GitHub repository

```bash
cd training/
gitrefresh
```

1. Go to the directory with the clair resources

```bash
cd ~/training/clair
```

2. Create the Secret that holds the Clair configuration

```bash
kubectl create secret generic clairsecret --from-file=./config.yaml

> from-file=./config.yaml
> secret/clairsecret created
```

3. Deploy Clair

```bash
kubectl create -f clair-kubernetes.yaml

> service/clairsvc created
> replicationcontroller/clair created
> replicationcontroller/clair-postgres created
> service/postgres created
```

4. Wait for Clair to be running (Ready is 1/1)

```bash
kubectl get pods

> NAME                    READY   STATUS              RESTARTS   AGE
> clair-jcb6r             1/1     Running             0          17s
> clair-postgres-2rbgj    1/1     Running             0          17s
```

Use Klar to scan an Image

Klar is a simple tool to analyze images stored in a private or public Docker registry for security vulnerabilities using Clair. Klar is designed to be used integrated in other CI/CD tools. It's a single binary which requires no dependencies.

Klar returns:

- 0 if the number of detected high severity vulnerabilities in an image is less than or equal to a threshold
- 1 if there were more.
- 2 if an error has prevented the image from being scanned

1. Execute the following command

```bash
CLAIR_ADDR=$(minikube ip):30060 FORMAT_OUTPUT=table CLAIR_OUTPUT=High klar nik
laushirt/k8sdemo:1.0.1

> clair timeout 1m0s
> docker timeout: 1m0s
> no whitelist file
> Analysing 14 layers
> Got results from Clair API v1
> Found 597 vulnerabilities
> Unknown: 176
> Negligible: 284
> Low: 135
> Medium: 2
```

We define several things here:

- `CLAIR_ADDR` : the address of the Clair server
- `FORMAT_OUTPUT` : the format for the output (table in this example)
- `CLAIR_OUTPUT` : output only vulnerabilities above or equal

The output shows that there are no HIGH severity vulerabilities.

> **Initialization of Clair takes some time!** If the output shows a completely different count than in this example you have to wait for Clair to ingest all security rules first.

2. Now let's execute it again with a lower OUTPUT (Medium)

```bash
CLAIR_ADDR=$(minikube ip):30060 FORMAT_OUTPUT=table CLAIR_OUTPUT=Medium klar n
iklaushirt/k8sdemo:1.0.1

> clair timeout 1m0s
> docker timeout: 1m0s
> no whitelist file
> Analysing 14 layers
> Got results from Clair API v1
> Found 597 vulnerabilities
> Unknown: 176
> Negligible: 284
> Low: 135
> Medium: 2

> +----------+---------------+-------------+----------------+---------+-------
------------------------- > +-------------------------------------------------
----------+
> | SEVERITY | NAME          | FEATURENAME | FEATUREVERSION | FIXEDBY | DESCRI
PTION                    | LINK
```

```
        > |
> +----------+---------------+-------------+----------------+----------+-------
------------------------- > +--------------------------------------------------
----------+
> | Medium   | CVE-2009-3546 | libwmf      | 0.2.8.4-10.6   |          | The _g
dGetColors function       | https://security-tracker.debian.org/tracker/CVE-200
9-3546 |
> |          |               |             |                |          | in gd_
gd.c in PHP 5.2.11 and    |
        > |
> |          |               |             |                |          | 5.3.x
before 5.3.1, and the     |
        > |
> |          |               |             |                |          | GD Gra
phics Library 2.x, does   |
        > |
...
> |          |               |             |                |          | from t
hird party information.   |
        > |
> +----------+---------------+-------------+----------------+----------+-------
------------------------- > +--------------------------------------------------
----------+
> | Medium   | CVE-2007-3996 | libwmf      | 0.2.8.4-10.6   |          | Multip
le integer overflows      | https://security-tracker.debian.org/tracker/CVE-200
7-3996 |
> |          |               |             |                |          | in lib
gd in PHP before 5.2.4    |
        > |
> |          |               |             |                |          | allow
remote attackers to       |
        > |
...
> |          |               |             |                |          | functi
on.                       |
        > |
> +----------+---------------+-------------+----------------+----------+-------
------------------------- > +--------------------------------------------------
----------+
```

The output shows the detail for the two vulnerabilities with Medum severity.

You can also output the result to JSON Format to be reused in a CI/CD tool.

Congratulations!!!

This concludes Lab 5 on Image Scanning.