Beyond `cluster-admin`: Getting Started with Kubernetes Users and Permissions

Tiffany Jernigan
Developer Advocate
VMware
tiffanyfayj

AUTHENTICATION & AUTHORIZATION

- AUTHN (authentication): who are you?

- AUTHZ (authorization): what are you allowed to do?

k8s.io/docs/reference/access-authn-authz/

AUTHENTICATION & USER PROVISIONING

TIFFANYFAYJ

TIFFANYFAY@MASTODON.ONLINE

PROVISIONING USERS

At least three possibilities:

- **certificates**
  - can use your own CA (e.g. Vault), or Kubernetes'
  - warning: Kubernetes API server doesn't support revocation, so you need short-lived certs
- **OIDC tokens**
  - can use an auth provider of your choice (e.g. okta, keycloak…) or something linked to your cloud's IAM
- (ab)use **serviceaccounts** to provision users
  (a service account is really just a user named `system:serviceaccount:<namespace>:<serviceaccountname>`)

PROVISIONING USERS

- **Humans**: TLS, OIDC, Service Account/Client Certs, etc.
- **Robots**: use Service Accounts

CERTIFICATES

- Example creation with OpenSSL:

```
# Generate key and CSR for our user
openssl genrsa 4096 > user.key
openssl req -new -key user.key \
    -subj /CN=ada.lovelace/O=devs/O=ops > user.csr
```

- After that, transfer the CSR to the CA

CERTIFICATES – SELF-HOSTED

```
# Copy the CSR to the CA (for instance, a kubeadm-deployed control
plane node)
# Then generate the cert:
sudo openssl x509 -req \
 -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key \
 -in user.csr -days 1 -set_serial 1234 > user.crt
# Copy certificate (user.crt) back to the user!
```

CERTIFICATES – THROUGH THE CSR API (1)

- Or we can use Kubernetes CA through the CSR API
- The Kubernetes cluster admin can submit the CSR like this:

```
kubectl apply -f - <<EOF
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: user=ada.lovelace
spec:
  #expirationSeconds: 3600
  request: $(base64 -w0 < user.csr)
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF
```

CERTIFICATES – THROUGH THE CSR API (2)

- Then approve it:

```
kubectl certificate approve user=ada.lovelace
```

- And retrieve the certificate like this:

```
kubectl get csr user=ada.lovelace -o
jsonpath={.status.certificate} | base64 -d > user.crt
```

- Now give back the user.crt file to the user!

ONCE WE HAVE OUR CERTIFICATE...

```
# Add the key and cert to our kubeconfig file like this:
kubectl config set-credentials ada.lovelace \
 --client-key=user.key --client-certificate=user.crt
```

TOKENS: OIDC

- OIDC is conceptually similar to TLS (but different set of protocols)
- On self-hosted clusters, you'd need to add a few command-line flags to API server:
  - `--oidc-issuer-url` → URL of the OpenID provider
  - `--oidc-client-id` → OpenID app requesting the authentication (=our cluster)
- More details on k8s.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens
- On managed clusters, there may be ways to achieve the same results, e.g. on EKS: docs.aws.amazon.com/eks/latest/userguide/authenticate-oidc-identity-provider.html

TOKENS: SERVICE ACCOUNT

- A Service Account is just a user with a funny name: "`system:serviceaccount:<namespace>:<serviceaccountname>`"
- "Service Account Tokens" are JWT generated by the Kubernetes control plane
- By default, in each container, Kubernetes will automatically place a token in that file: `/var/run/secrets/kubernetes.io/serviceaccount/token`
- That token is a token for the Service Account of the Pod that the container belongs to
- Kubernetes client libraries know to automatically detect and use that token
- We're going to see that in practice!

ROLE-BASED ACCESS CONTROL (RBAC)

TIFFANYFAYJ

TIFFANYFAY@MASTODON.ONLINE

RBAC

High level idea on Kubernetes:

1. Define a ROLE (or ClusterRole), which is a collection of permissions ("things that can be done")
    a. e.g. list pods

1. Bind the ROLE to a USER or GROUP or SERVICEACCOUNT (with a RoleBinding or ClusterRoleBinding)

RBAC

- `kubectl get --raw /api/v1`
  (core resources with apiVersion: v1)
- `kubectl get --raw /apis/<group>/<version>`
  (for other resources)

Example Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: cnsc
```

Example Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: get-pods
  namespace: cnsc
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
```

Example RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: get-pods
  namespace: cnsc
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: get-pods
subjects:
- kind: ServiceAccount
  name: default
  namespace: cnsc
```

RBAC AUDITING

After setting permissions, audit them:

- `kubectl auth can-i --list`
- `kubectl who-can` / kubectl-who-can by Aqua Security
- `kubectl access-matrix` / Rakkess (Review Access) by Cornelius Weig
- `kubectl rbac-lookup` / RBAC Lookup by FairwindsOps
- `kubectl rbac-tool` / RBAC Tool by insightCloudSec

DEMO

DEMO P0

WINDOW 1

```
k create ns cnsc
kubens cnsc

k create deploy nginx --image=nginx
k create deploy web --image=nginx

k get pods
```

WINDOW 2

```
k run -it tester --rm --
image=nixery.dev/shell/kubectl/curl/jq -
- sh

#check that if we "kubectl get pods" in
the pod (it won't work)
kubectl get pods
kubectl auth can-i --list
```

DEMO: USING DEFAULT NS SA

WINDOW 1

```
#create a role that can get pods
k create role get-pods \
     --verb=get --verb=list \
     --resource=pods

#bind role (create RoleBinding) to the NS default SA
k create rolebinding get-pods --
role=get-pods --
serviceaccount=cnsc:default
```

WINDOW 2

```
kubectl get pods

#now "kubectl get pods -v6" so we see the req URL
kubectl get pods -v6

#-k, --insecure Allow insecure server connections
when using SSL
Basically I don't care about the cert shown to me by
the Kubernetes API server. I trust that I am talking
to my cluster and not some impersonator
curl
https://$IP:443/api/v1/namespaces/cnsc/p
ods -k
```

DEMO: USING DEFAULT NS SA

WINDOW 2

```
cat /var/run/secrets/kubernetes.io/serviceaccount/token
#copy it and paste it in jwt.io to see what it shows
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)

We can also find the HOST:PORT in env as $KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT
env

curl https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/api/v1/namespaces/cnsc/pods
-k -H "Authorization: Bearer $TOKEN" | jq .items[].metadata.name
#Gets API server certificate
curl https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/api/v1/namespaces/cnsc/pods
-H "Authorization: Bearer $TOKEN" --cacert
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt | jq .items[].metadata.name
```

DEMO: BOUND SERVICE ACCOUNT TOKENS

WINDOW 1

```
k run -it pirate --rm --image=nixery.dev/shell/kubectl/curl/jq -- sh

TOKEN="<ctrl-v>"

#Gets API server certificate
curl https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/api/v1/namespaces/cnsc/pods
-H "Authorization: Bearer $TOKEN" --cacert
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt | jq .items[].metadata.name

#In window 2 type exit and hit enter in the pod and wait for it to be completely gone
#since the token is gone this should fail
curl https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/api/v1/namespaces/cnsc/pods
-H "Authorization: Bearer $TOKEN" --cacert
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt | jq

exit
```

DEMO: CREATING A NEW SA

WINDOW 1

```
#create new sa called scaler
k create sa scaler
```

WINDOW 2

```
#create pod using this new service account
k run -it scaler --rm --image=nixery.dev/shell/kubectl/curl/jq --
overrides='{ "spec": { "serviceAccount": "scaler" } }' -- sh
```

DEMO: FACTORY ROLES

WINDOW 1

```
#add ability to view resources using an existing cluster role
k create clusterrolebinding scaler-view --clusterrole=view --
serviceaccount=cnsc:scaler
```

WINDOW 2

```
#verify it works
kubectl get all
#you only have view, so this should fail
kubectl delete deployment/nginx
```

DEMO: SCALER

WINDOW 1

```
#create a role that can scale deployments
k create role scaler --verb=patch --resource=deployments/scale --resource-name=nginx

#bind role (create RoleBinding) to the scaler SA
k create rolebinding scaler --role=scaler --serviceaccount=cnsc:scaler
```

WINDOW 2

```
kubectl scale deployment nginx --replicas=2

#will fail since it is tied to nginx
kubectl scale deployment web --replicas=2

kubectl delete deployment nginx
```

DEMO: SCALER

WINDOW 1

```
#remove scaler-view clusterrolebinding
k delete clusterrolebinding scaler-view
```

WINDOW 2

```
#will fail since you can't get the deployment anymore
kubectl scale deployment nginx --replicas=2
```

DEMO: SCALER

WINDOW 1

```
#edit scaler role to add get to the nginx deployment
k edit role scaler
#Add another apigroup under the rules like this:
rules:
- apiGroups:
  - apps
  resourceNames:
  - nginx
  resources:
  - deployments
  verbs:
  - get
```

WINDOW 2

```
kubectl scale deploy nginx --replicas=1

#will fail since you can't get any other resources
kubectl get deploy

kubectl get deploy nginx

exit

k delete ns cnsc
```

THANK YOU!

- tiffanyfayj
  tiffanyfay@mastodon.online

Special thanks to:
Jérôme Petazzoni