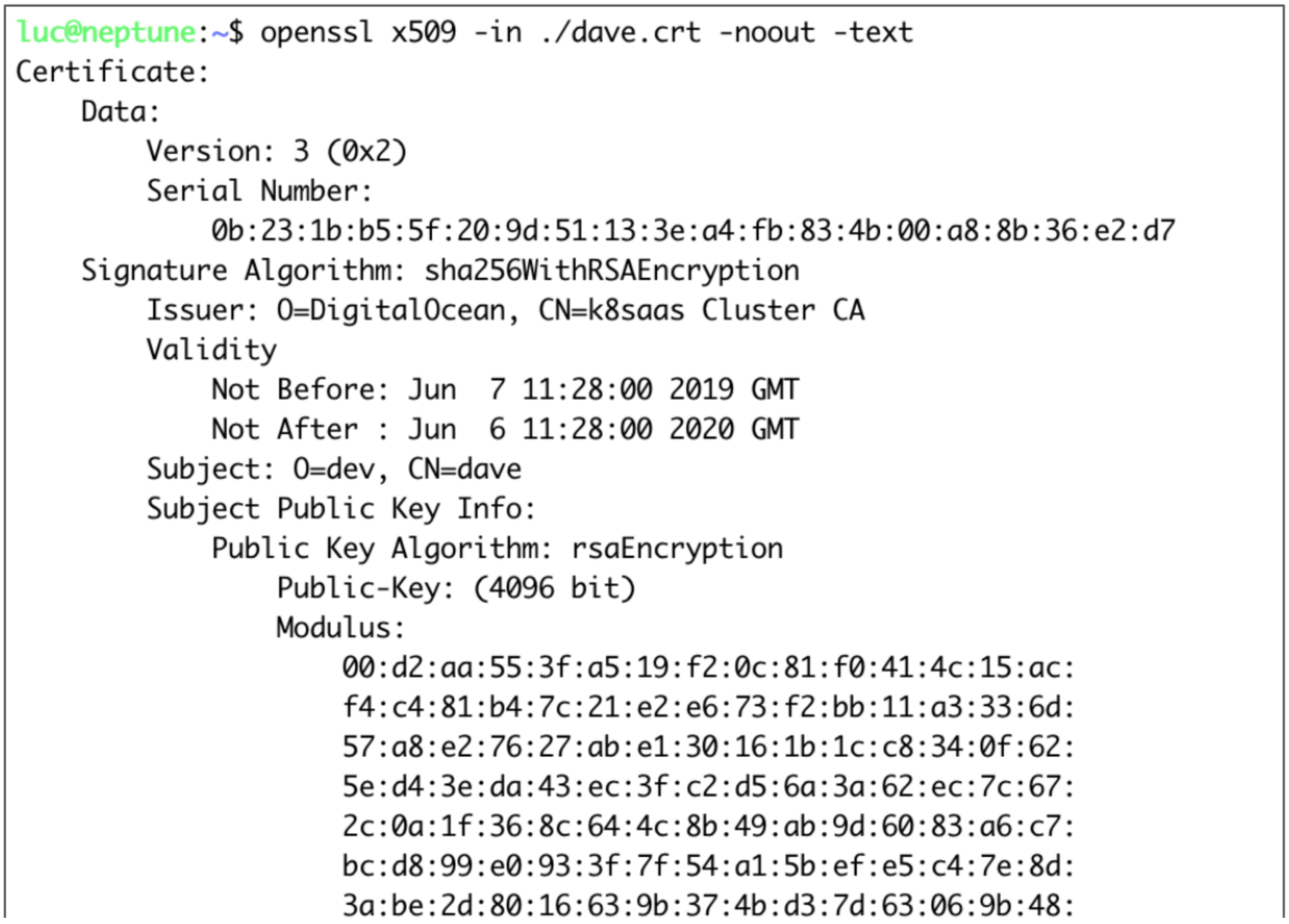
Kubernetes Tips: Give Access To Your Cluster With A Client Certificate

A simple guide to giving users access to the new Kubernetes cluster, including authentification setup and RoleBinding

Luc Juggery Follow
Jun 8, 2019 · 11 min read ★



```
luc@neptune:~$ openssl x509 -in ./dave.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0b:23:1b:b5:5f:20:9d:51:13:3e:a4:fb:83:4b:00:a8:8b:36:e2:d7
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: O=DigitalOcean, CN=k8saas Cluster CA
        Validity
            Not Before: Jun  7 11:28:00 2019 GMT
            Not After : Jun  6 11:28:00 2020 GMT
        Subject: O=dev, CN=dave
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:d2:aa:55:3f:a5:19:f2:0c:81:f0:41:4c:15:ac:
                    f4:c4:81:b4:7c:21:e2:e6:73:f2:bb:11:a3:33:6d:
                    57:a8:e2:76:27:ab:e1:30:16:1b:1c:c8:34:0f:62:
                    5e:d4:3e:da:43:ec:3f:c2:d5:6a:3a:62:ec:7c:67:
                    2c:0a:1f:36:8c:64:4c:8b:49:ab:9d:60:83:a6:c7:
                    bc:d8:99:e0:93:3f:7f:54:a1:5b:ef:e5:c4:7e:8d:
                    3a:be:2d:80:16:63:9b:37:4b:d3:7d:63:06:9b:48:
```

We have just set up a brand new Kubernetes cluster (congrats! 🎉). It will be used across our company soon and we already have a colleague, Dave from the *development* team, who wants to start playing with the beast and deploy and test his brand new microservices application on it. What are the simple steps we can do to get him access? That's what this post is about (hint: we will use an x509 client certificate).

. . .

User Management in Kubernetes

To manage a Kubernetes cluster and the applications running on it, the kubectl binary or the Web UI are usually used. Behind the hood those tools call the API Server: the HTTP Rest API exposing all the endpoints of the cluster's control plane.

(The documentation of this HTTP API is great, just check it out: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.14)

When a request is sent to the API Server, it first needs to be authenticated (to make sure the requestor is known by the system) before it's authorized (to make sure the requestor is allowed to perform the action requested).

The authentication step is done through the use of authentication plugins. There are several plugins as different authentication mechanisms can be used:

- Client certificates (the one we will talk about in this post)
- Bearer tokens
- Authenticating proxy
- HTTP basic auth

Depending upon the authentication mechanism used, the corresponding plugin expects

There is no user nor group resources inside a Kubernetes cluster. This should be handled outside of the cluster and provided with each request sent to the API Server.

Don't worry if things are not crystal clear yet, we will illustrate it below.

. . .

Some Considerations and Assumptions

- The cluster will be used by several teams/clients (multi-tenants approach), as the workload of each tenant needs to be isolated. We will start by creating a namespace named *development* dedicated to the development team (the one Dave belongs to).
- Dave needs to deploy standard Kubernetes resources. He will then be provided the right to create, list, update, get and delete Deployments and Services resources only. Additional rights could be provided later on if needed. We will ensure those rights are limited to the *development* namespace.
- Chances are that other members of Dave's team need to have the same level of access later on. We will thus consider a group named *dev* and provide rights at the group's level as well.
- Dave needs to have *kubectl* installed (that's probably already the case as he may have played with *Minikube* on this local machine), and he also needs *openssl* as he will generate a private key and a certificate sign-in request.

. . .

Creation of a Private Key and a Certificate Signing Request (CSR)

Dave first needs to generate a private rsa key and a CSR. The private key can easily be

The CSR is a little bit more complicated. Dave needs to make sure he:

- Uses his name in the Common Name (CN) field: this will be used to identify him against the API Server.
- Uses the group name in the Organisation (O) field: this will be used to identify the group against the API Server.

Below is the configuration file Dave will use to generate the CSR:

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn

[ dn ]
CN = dave
O = dev

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
```

Note: the clientAuth entry in the extendedKeyUsage field is important as the certificate will be used to identify the client.

Using the above configuration file (saved in csr.cnf), the CSR can be created using the following command:

```
$ openssl req -config ./csr.cnf -new -key dave.key -nodes -out
dave.csr
```

Signature of the CSR

The signature of the .csr file will result in the creation of a certificate. This one will be used to authenticate each request Dave will send to the API Server.

We will start by creating a Kubernetes CertificateSigninRequest resource.

Note: we might have set up a managed cluster (they are plenty out there: DigitalOcean, Google's GKE, Microsoft Azure AKS, …), or created our own cluster (with kubeadm, kubespray, …). The process is the same.

We will use the following specification and save it in *csr.yaml*.

```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: mycsr
spec:
  groups:
  - system:authenticated
  request: ${BASE64_CSR}
  usages:
  - digital signature
  - key encipherment
  - server auth
  - client auth
```

As we can see, the value of the *request* key is the content of the *BASE64_CSR* environment variable. The first step is to get the base64 encoding of the .csr file generated by Dave and then use the *envsubst* binary to substitute the value of this variable before creating the resource.

```
# Encoding the .csr file in base64
$ export BASE64_CSR=$(cat ./dave.csr | base64 | tr -d '\n')

# Substitution of the BASE64 CSR env variable and creation of the
# Checking the status of the newly created CSR
$ kubectl get csr
NAME         AGE   REQUESTOR             CONDITION
mycsr        9s    28b93...d73801ee46    Pending
```

We can then approve this CSR with this command:

```
$ kubectl certificate approve mycsr
```

Checking the status of the CSR once again, we can see it's now approved.

```
$ kubectl get csr
NAME         AGE   REQUESTOR             CONDITION
mycsr        9s    28b93...d73801ee46    Approved,Issued
```

The certificate is created. Let's just extract it from the CSR resource and save it in a file named *dave.crt* to check what's inside.

```
$ kubectl get csr mycsr -o jsonpath='{.status.certificate}' \
  | base64 --decode > dave.crt
```

The following openssl command shows the certificate has been signed by the DigitalOcean's cluster CA (Issuer part), the subject contains *dave* in the CN (CommonName) field and *dev* in the O (Organisation) field as Dave specified when creating the .csr file.

```
$ openssl x509 -in ./dave.crt -noout -text
Certificate:
    Data:
            Not Before: Jun  3 07:56:00 2019 GMT
            Not After : Jun  2 07:56:00 2020 GMT
        Subject: O=dev, CN=dave
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
...
```

Note: the cluster used in this example is a managed Kubernetes cluster created on DigitalOcean. We can see it from the cluster Certificate Authority.

. . .

Creation of a Namespace

We start by creating a namespace, named *development*, so all the resources Dave and his team will deploy are isolated from the other workload of the cluster. It can be created with a simple command:

```
$ kubectl create ns development
```

or with this *dev-ns.yaml* file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

followed by the usual:

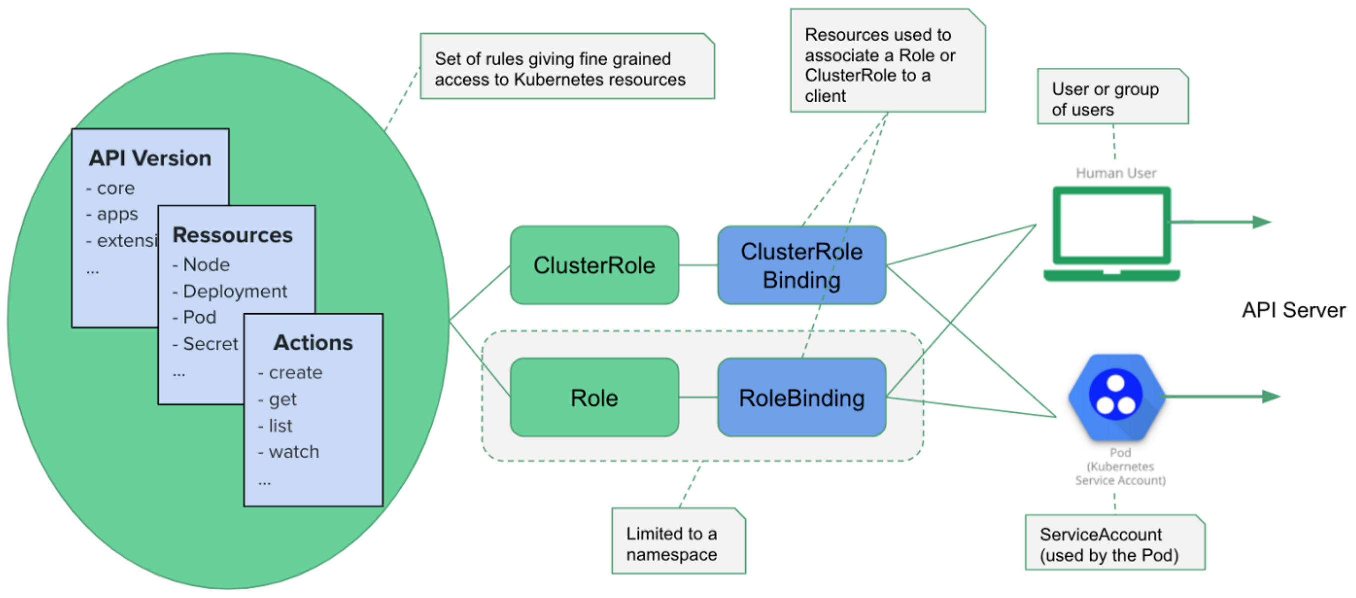
Note: a best practice is to create a ResourceQuota resource and link it to the namespace in order to limit the amount of cpu and RAM that can be used within the namespace, but that's for another article.

. . .

Setting Up RBAC Rules

By creating a certificate, we allow Dave to authenticate against the API Server, but we did not specify any rights so he will not be able to do many things… We will change that and give him the rights to create, get, update, list and delete Deployment and Service resources in the *dev* namespace.

The following picture shows the resources involved in the Kubernetes Role Base Access Control (RBAC).



Overview of the resource involved

In a nutshell: A Role (the same applies to a ClusterRole) contains a list of rules. Each

. . .

Creation of a Role

Let’s first create a Role resource with the following specification:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 namespace: development
 name: dev
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["create", "get", "update", "list", "delete"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["create", "get", "update", "list", "delete"]
```

Pods and Services resources belongs to the *core* API group (value of the apiGroups key is the empty string), whereas Deployments resources belongs to the *apps* API group. For those 2 apiGroups, we defined the list of resources and the actions that should be authorized on those ones.

Assuming the content above is in *role.yaml*, the creation of the role is done with the following command:

```
$ kubectl apply -f role.yaml
```

. . .

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: dev
 namespace: development
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: dev
 apiGroup: rbac.authorization.k8s.io
```

This RoleBinding links:

- A subject: our user Dave.
- A role: the one named *dev* that allows to create/get/update/list/delete the Deployment and Service resources that we defined above.

Note: as Dave belongs to the *dev* group, we could use the following RoleBinding in order to bind the previous Role with the group instead of with an individual user. Remember: the group information is provided in the Organisation (O) field within the certificate that is sent with each request.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: dev
 namespace: development
subjects:
- kind: Group
  name: dev
  apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: dev
 apiGroup: rbac.authorization.k8s.io
```

```
$ kubectl apply -f role-binding.yaml
```

. . .

Building a Kube Config for Dave

Everything is set up. We now have to send Dave the information he needs to configure his local *kubectl* client to communicate with our cluster.

We'll first create a *kubeconfig.tpl* file, with the following content, that we'll use as a template.

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: ${CLUSTER_CA}
    server: ${CLUSTER_ENDPOINT}
  name: ${CLUSTER_NAME}
users:
- name: ${USER}
  user:
    client-certificate-data: ${CLIENT_CERTIFICATE_DATA}
contexts:
- context:
    cluster: ${CLUSTER_NAME}
    user: dave
  name: ${USER}-${CLUSTER_NAME}
current-context: ${USER}-${CLUSTER_NAME}
```

To build a base kube config from this template, we first need to set all the needed environment variables:

```
# User identifier
# Client certificate
$ export CLIENT_CERTIFICATE_DATA=$(kubectl get csr mycsr -o
jsonpath='{.status.certificate}')

# Cluster Certificate Authority
$ export CLUSTER_CA=$(kubectl config view --raw -o json | jq -r
'.clusters[] | select(.name == "'$(kubectl config current-context)'")
| .cluster."certificate-authority-data"')

# API Server endpoint
$ export CLUSTER_ENDPOINT=$(kubectl config view --raw -o json | jq -r
'.clusters[] | select(.name == "'$(kubectl config current-context)'")
| .cluster."server"')
```

and substitute them using, once again, the convenient envsubst utility:

```
$ cat kubeconfig.tpl | envsubst > kubeconfig
```

We can now send this *kubeconfig* file to Dave who will just need to add his private key inside of it and he will be fine to communicate with the cluster.

. . .

Use of the Context

In order to use the *kubeconfig,* Dave can set the KUBECONFIG environment variable with the path towards the file.

```
$ export KUBECONFIG=$PWD/kubeconfig
```

Note: there are different ways to use a Kubernetes configuration: setting the *KUBECONFIG* environment variable, adding a new entry in the default

```
$ kubectl config set-credentials dave \
  --client-key=$PWD/dave.key \
  --embed-certs=true
```

It will create the key *client-key-data* within the user entry of the *kubeconfig* file and set the base64 encoding of *dave.key* as the value.

If everything is fine, Dave should be able to check the version of the server (and the client) with the following command:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"14",
GitVersion:"v1.14.2",
GitCommit:"66049e3b21efe110454d67df4fa62b08ea79a19b",
GitTreeState:"clean", BuildDate:"2019-05-16T16:23:09Z",
GoVersion:"go1.12.5", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"14",
GitVersion:"v1.14.2",
GitCommit:"66049e3b21efe110454d67df4fa62b08ea79a19b",
GitTreeState:"clean", BuildDate:"2019-05-16T16:14:56Z",
GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"}
```

Let's go one step further and check if the current Role associated to Dave allows him to list the nodes of the cluster.

```
$ kubectl get nodes
Error from server (Forbidden): nodes is forbidden: User "dave" cannot
list resource "nodes" in API group "" at the cluster scope
```

Of course not! But Dave should now be able to deploy stuff on the cluster—well, at least in the namespace named *development*. Let's check this with this sample yml file defining a Deployment based on the nginx image and a Service to expose it.

```
metadata:
  name: www
  namespace: development
spec:
  replicas: 3
  selector:
    matchLabels:
      app: www
  template:
    metadata:
      labels:
        app: www
    spec:
      containers:
      - name: nginx
        image: nginx:1.14-alpine
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: www
  namespace: development
spec:
  selector:
    app: vote
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 80
```

We can see from the following command that Dave can create those resources in the cluster:

```
$ kubectl apply -f www.yaml
deployment.apps/www created
service/www created
```

Dave is limited to the *development* namespace.We can confirm it from the error message he gets when trying to list all the Pods within the *default* namespace:

Also, he will not be able to create other resources than the one we granted him access to. For example, we can consider the following specification of a resource of type Secret:

```
# credentials.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
  namespace: development
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Let's see Dave try to create it:

```
$ kubectl apply -f credentials.yaml
Error from server (Forbidden): error when retrieving current
configuration of:
Resource: "/v1, Resource=secrets", GroupVersionKind: "/v1,
Kind=Secret"
Name: "mysecret", Namespace: "development"
Object: &{map["apiVersion":"v1"
"data":map["password":"MWYyZDFlMmU2N2Rm" "username":"YWRtaW4="]
"kind":"Secret"
"metadata":map["annotations":map["kubectl.kubernetes.io/last-applied-
configuration":""] "name":"mysecret" "namespace":"development"]]}
from server for: "credentials.yaml": secrets "mysecret" is forbidden:
User "dave" cannot get resource "secrets" in API group "" in the
namespace "development"
```

. . .

Summary

Once the authentication was set up, we used a Role to define some rights limited to a namespace and bind it to the user with a RoleBinding. In case we need to provide Cluster-wide rights, we could use ClusterRole and ClusterRoleBinding resources.

. . .

Good luck! If you gave this a try, I'd love to hear what you think in the comments below.

Kubernetes Programming DevOps Software Development