Chapter 16

Security

16.1 Labs

Exercise 16.1: Working with TLS

Overview

We have learned that the flow of access to a cluster begins with TLS connectivity, then authentication followed by authorization, finally an admission control plug-in allows advanced features prior to the request being fulfilled. The use of `Initializers` allows the flexibility of a shell-script to dynamically modify the request. As security is an important, ongoing concern there may be multiple configurations used depending on the needs of the cluster.

Every process making API requests to the cluster must authenticate or be treated as an anonymous user.

Working with TLS

While one can have multiple cluster root Certificate Authorities (CA) by default each cluster uses their own, intended for intra-cluster communication. The CA certificate bundle is distributed to each node and as a secret to default service accounts. The **kubelet** is a local agent which ensures local containers are running and healthy.

1. View the **kubelet** on both the master and secondary nodes. The **kube-apiserver** also shows security information such as certificates and authorization mode.

```
student@lfs458-node-1a0a:~$ ps -ef |grep kubelet
<output_omitted>
--cluster-domain=cluster.local --authorization-mode=Webhook
--client-ca-file=/etc/kubernetes/pki/ca.crt --cadvisor-port=0
--rotate-certificates=true --cert-dir=/var/lib/kubelet/pki
<output_omitted>
--service-account-key-file=/etc/kubernetes/pki/sa.pub --secure-port=6443
--requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt
--proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
```

```
<output_omitted>
--authorization-mode=Node,RBAC
--etcd-servers=http://127.0.0.1:2379


student@lfs458-node-2b2b:~$ ps -ef |grep kubelet
<output_omitted>
--authorization-mode=Webhook --client-ca-file=/etc/kubernetes/pki/ca.crt
--cadvisor-port=0 --rotate-certificates=true
--cert-dir=/var/lib/kubelet/pki
```

2. View the configuration file where these settings are made.

```
student@lfs458-node-1a0a:~$ ls /etc/kubernetes/manifests/
etcd.yaml            kube-controller-manager.yaml
kube-apiserver.yaml  kube-scheduler.yaml

student@lfs458-node-1a0a:~$ sudo less /etc/kubernetes/manifests/kube-controller-manager.yaml
<output_omitted>
```

3. Note the certificate used is the same as that from `~/.kube/config`.

```
student@lfs458-node-1a0a:~$ kubectl get csr -o yaml
<output_omitted>
  status:
    certificate: LS0tLS1CRUdJTiBDRVJUSUZJ......
<output_omitted>

student@lfs458-node-1a0a:~$ grep cert ~/.kube/config
<output_omitted>
```

4. The **kubectl config** command can also be used to update parameters, which could avoid a typo removing access to the cluster. View the current configuration settings.

```
student@lfs458-node-1a0a:~$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
<output_omitted>
```

5. View the options, such as setting a password for the admin instead of a key. Read through the examples and options.

```
student@lfs458-node-1a0a:~$ kubectl config set-credentials -h
Sets a user entry in kubeconfig
<output_omitted>
```

6. Make a copy of your access configuration file. Later steps will update this file and we can view the differences.

```
student@lfs458-node-1a0a:~$ cp ~/.kube/config ~/cluster-api-config
```

7. Explore working with cluster and security configurations both using **kubectl** and **kubeadm**. Among other values, find the name of your cluster. You will need to become `root` to work with **kubeadm**.

```
student@lfs458-node-1a0a:~$ kubectl config <Tab><Tab>
current-context  get-contexts     set-context      view
delete-cluster   rename-context   set-credentials
delete-context   set              unset
get-clusters     set-cluster      use-context

student@lfs458-node-1a0a:~$ sudo -i

root@lfs458-node-1a0a:~# kubeadm token -h
<output_omitted>
```

```
root@lfs458-node-1a0a:~# kubeadm config -h
<output_omitted>
```

Exercise 16.2: Authentication and Authorization

Kubernetes clusters have to types of users `service accounts` and `normal users`, but `normal users` are assumed to be managed by an outside service. There are no objects to represent them and they cannot be added via an API call, but `service accounts` can be added.

We will use **RBAC** to configure access to actions within a `namespace` for a new contractor, `Developer Dan` who will be working on a new project.

1. Create two namespaces, one for `production` and the other for `development`.

   ```
   student@lfs458-node-1a0a:~$ kubectl create ns development
   namespace "development" created

   student@lfs458-node-1a0a:~$ kubectl create ns production
   namespace "production" created
   ```

2. View the current clusters and context available. The context allows you to configure the cluster to use, namespace and user for **kubectl** commands in an easy and consistent manner.

   ```
   student@lfs458-node-1a0a:~$ kubectl config get-contexts
   CURRENT   NAME               CLUSTER      AUTHINFO        NAMESPACE
   *         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
   ```

3. Create a new user `DevDan` and assign a password of `lfs458`.

   ```
   student@lfs458-node-1a0a:~$ sudo useradd -s /bin/bash DevDan
   student@lfs458-node-1a0a:~$ sudo passwd DevDan
   Enter new UNIX password:  lfs458
   Retype new UNIX password:  lfs458
   passwd: password updated successfully
   ```

4. Generate a private key then Certificate Signing Request (CSR) for `DevDan`.

   ```
   student@lfs458-node-1a0a:~$ openssl genrsa -out DevDan.key 2048
   Generating RSA private key, 2048 bit long modulus
   ......+++
   .........+++
   e is 65537 (0x10001)

   student@lfs458-node-1a0a:~$ openssl req -new -key DevDan.key \
         -out DevDan.csr -subj "/CN=DevDan/O=development"
   ```

5. Using thew newly created request generate a self-signed certificate using the x509 protocol. Use the CA keys for the Kubernetes cluster and set a 45 day expiration. You'll need to use **sudo** to access to the inbound files.

   ```
   student@lfs458-node-1a0a:~$ sudo openssl x509 -req -in DevDan.csr \
        -CA /etc/kubernetes/pki/ca.crt \
        -CAkey /etc/kubernetes/pki/ca.key \
        -CAcreateserial \
        -out DevDan.crt -days 45
   Signature ok
   subject=/CN=DevDan/O=development
   Getting CA Private Key
   ```

6. Update the access config file to reference the new key and certificate. Normally we would move them to a safe directory instead of a non-root user's home.

```
student@lfs458-node-1a0a:~$ kubectl config set-credentials DevDan \
      --client-certificate=/home/student/DevDan.crt \
      --client-key=/home/student/DevDan.key
User "DevDan" set.
```

7. View the update to your credentials file. Use **diff** to compare against the copy we made earlier.

```
student@lfs458-node-1a0a:~$ diff cluster-api-config .kube/config
9a10,14
>     namespace: development
>     user: DevDan
>   name: DevDan-context
> - context:
>     cluster: kubernetes
15a21,25
> - name: DevDan
>   user:
>     as-user-extra: {}
>     client-certificate: /home/student/DevDan.crt
>     client-key: /home/student/DevDan.key
```

8. We will now create a context. For this we will need the name of the cluster, namespace and CN of the user we set or saw in previous steps.

```
student@lfs458-node-1a0a:~$ kubectl config set-context DevDan-context \
       --cluster=kubernetes \
       --namespace=development \
       --user=DevDan
Context "DevDan-context" created.
```

9. Attempt to view the `Pods` inside the `DevDan-context`. Be aware you will get an error.

```
student@lfs458-node-1a0a:~$ kubectl --context=DevDan-context get pods
Error from server (Forbidden): pods is forbidden: User "DevDan"
cannot list pods in the namespace "development"
```

10. Verify the context has been properly set.

```
student@lfs458-node-1a0a:~$ kubectl config get-contexts
CURRENT   NAME                 CLUSTER      AUTHINFO           NAMESPACE
          DevDan-context       kubernetes   DevDan             development
*         kubernetes-admin@kubernetes kubernetes kubernetes-admin
```

11. Again check the recent changes to the cluster access config file.

```
student@lfs458-node-1a0a:~$ diff cluster-api-config .kube/config
9a10,14
>     namespace: development
>     user: DevDan
>   name: DevDan-context
> - context:
>     cluster: kubernetes
15a21,25
> - name: DevDan
>   user:
<output_omitted>
```

12. We will now create a YAML file to associate RBAC rights to a particular namespace and `Role`.

```
student@lfs458-node-1a0a:~$ vim role-dev.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: development
  name: developer
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["list", "get", "watch", "create", "update", "patch", "delete"]
# You can use ["*"] for all verbs
```

13. Create the object. Check white space and for typos if you encounter errors.

```
student@lfs458-node-1a0a:~$ kubectl create -f role-dev.yaml
role "developer" created
```

14. Now we create a `RoleBinding` to associate the `Role` we just created with a user. Create the object when the file has been created.

```
student@lfs458-node-1a0a:~$ vim rolebind.yaml


kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: developer-role-binding
  namespace: development
subjects:
- kind: User
  name: DevDan
  apiGroup: ""
roleRef:
  kind: Role
  name: developer
  apiGroup: ""


student@lfs458-node-1a0a:~$ kubectl apply -f rolebind.yaml
rolebinding "developer-role-binding" created
```

15. Test the context again. This time it should work. There are no `Pods` running so you should get a response of `No resources found`.

```
student@lfs458-node-1a0a:~$  kubectl --context=DevDan-context get pods
No resources found.
```

16. Create a new pod, verify it exists, then delete it.

```
student@lfs458-node-1a0a:~$ kubectl --context=DevDan-context run nginx --image=nginx
deployment "nginx" created

student@lfs458-node-1a0a:~$ kubectl --context=DevDan-context get pods
NAME                      READY     STATUS              RESTARTS    AGE
nginx-7c87f569d-7gb9k     1/1       Running             0           5s

student@lfs458-node-1a0a:~$ kubectl --context=DevDan-context delete deploy nginx
deployment "nginx" deleted
```

17. We will now create a different context for production systems. The `Role` will only have the ability to view, but not create or delete resources. Begin by copying and editing the `Role` and `RoleBindings` YAML files.

```
student@lfs458-node-1a0a:~$ cp role-dev.yaml role-prod.yaml

student@lfs458-node-1a0a:~$ vim role-prod.yaml

kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: production       #<<- This line
  name: dev-prod              #<<- and this line
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch"] #<<- and this one

student@lfs458-node-1a0a:~$ cp rolebind.yaml rolebindprod.yaml

student@lfs458-node-1a0a:~$ vim rolebindprod.yaml


kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: production-role-binding
  namespace: production
subjects:
- kind: User
  name: DevDan
  apiGroup: ""
roleRef:
  kind: Role
  name: dev-prod
  apiGroup: ""
```

18. Create both new objects.

```
student@lfs458-node-1a0a:~$ kubectl apply -f role-prod.yaml
role "dev-prod" created

student@lfs458-node-1a0a:~$ kubectl apply -f rolebindprod.yaml
rolebinding "production-role-binding" created
```

19. Create the new context for production use.

```
student@lfs458-node-1a0a:~$ kubectl config set-context ProdDan-context \
       --cluster=kubernetes \
       --namespace=production \
       --user=DevDan
Context "ProdDan-context" created.
```

20. Verify that user DevDan can view pods using the new context.

```
student@lfs458-node-1a0a:~$ kubectl --context=ProdDan-context get pods
No resources found.
```

21. Try to create a `Pod` in production. The developer should be `Forbidden`.

```
student@lfs458-node-1a0a:~$ kubectl --context=ProdDan-context run \
      nginx --image=nginx
Error from server (Forbidden): deployments.extensions is forbidden: User "DevDan" cannot create deployments.extension
```

22. View the details of a role.

```
student@lfs458-node-1a0a:~$ kubectl describe role dev-prod -n production
Name:         dev-prod
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration=
{"apiVersion":"rbac.authorization.k8s.io/v1beta1","kind":"Role"
,"metadata":{"annotations":{},"name":"dev-prod","namespace":
"production"},"rules":[{"api...
PolicyRule:
  Resources         Non-Resource URLs  Resource Names  Verbs
  ---------          -----------------  --------------  -----
  deployments        []                 []              [get list watch]
  deployments.apps   []                 []              [get list watch]
<output_omitted>
```

23. Experiment with other subcommands in both contexts. They should match those listed in the respective roles.

Exercise 16.3: Admission Controllers

The last stop before a request is sent to the API server is an admission control plug-in. They interact with features such as setting parameters like a default storage class, checking resource quotas, or security settings. A newer feature (v1.7.x) is dynamic controllers which allow new controllers to be ingested or configured at runtime.

The order of listing matters. When an admission controller matches the plug-ins action is performed, acceptance or mutation of the request, and the request is sent along to the API server. No further plug-ins are checked.

1. View the current admission controller list and order. Note that Initializers, a dynamic controller, is listed first. This allows modification of the object. The use of webhooks could also be used if notification is the only desired goal.

```
student@lfs458-node-1a0a:~$ sudo grep admission \
                 /etc/kubernetes/manifests/kube-apiserver.yaml
    - --admission-control=Initializers,NamespaceLifecycle,
    LimitRanger,ServiceAccount,PersistentVolumeLabel,
    DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,
    ResourceQuota
```