Security

Vasilis

August 9, 2025

Contents

1 Authentication

1.1 Basic Authentication Using Static Password Files

Start the API server with the following parameter:

```
kube-apiserver --basic-auth-file=user-details.csv
```

Below is an example of a basic authentication CSV file (`user-details.csv`):

```
password123,user1,u0001
password123,user2,u0002
password123,user3,u0003
password123,user4,u0004
password123,user5,u0005
```

1.2 Token-Based Authentication Using Static Token Files

Example token authentication CSV file (user-token-details.csv):

```
KpjCVbI7cFAHYPkByTIzRb7gulcUc4B,user10,u0010,group1
rJjncHmvtXHc6MlWQddhtvNyyhgTdxSC,user11,u0011,group1
mjpoFTEiFOkL9toikaRNTt59ePtczZSq,user12,u0012,group2
PG41IXhs7QjqWkmBkvGT9gclOyUqZj,user13,u0013,group2


# Start the API server with the token file:
kube-apiserver --token-auth-file=user-token-details.csv

# When making API requests, include the token as an authorization bearer token
curl -v -k https://master-node-ip:6443/api/v1/pods --header "Authorization: Bearer Kpj
```

1.3 Note

- This is not a recommended authentication mechanism
- Consider volume mount while providing the auth file in a kubeadm setup
- Setup Role Based Authorization for the new users

2 TLS Basics

2.1 SSH access using key pairs

```
# Generate SSH key pair
ssh-keygen
# Files generated: id_rsa (private key) and id_rsa.pub (public key)

# review the authorized keys on the server
cat ~/.ssh/authorized_keys

# connect securely to the server using your key pair
ssh -i id_rsa user1@server1
```

2.2 Asymmetric encryption

Asymmetric encryption addresses the risk if intercepted by securely transferring the symmetric key. Here's how the process works for a web server using HTTPS:

1. The server generates a key pair (private and public keys).
2. Upon a user's initial HTTPS request, the server sends its public key embedded within a certificate.
3. The client's browser encrypts a newly generated symmetric key using the server's public key.
4. The encrypted symmetric key is sent back to the server.
5. The server decrypts the symmetric key using its private key.
6. All subsequent communications are encrypted with this symmetric key.

```
# Generate a private key
openssl genrsa -out my-bank.key 1024

# Extract the public key
openssl rsa -in my-bank.key -pubout > mybank.pem
```

A certificate contains essential details that help verify its authenticity:

- Identity of the issuing authority
- The server's public key
- Domain and other related information

Below is an example excerpt from a certificate:

```
Certificate:
Data:
  Serial Number: 420327018966204255
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: CN=kubernetes
  Validity
    Not After : Feb  9 13:41:28 2020 GMT
  Subject: CN=my-bank.com
  X509v3 Subject Alternative Name:
    DNS:mybank.com, DNS:i-bank.com,
    DNS:we-bank.com,
  Subject Public Key Info:
    00:b9:b0:55:24:fb:a4:ef:77:73:7c:9b
```

3 TLS in Kubernetes

3.1 Naming Conventions

Certificate files follow specific naming conventions. Public key certificates typically have a .crt or .pem extension (e.g., server.crt, server.pem, client.crt, client.pem). In contrast, private keys usually include the word "key" in their file name or extension (e.g., server.key or server-key.pem). If a file name lacks "key," it is almost certainly a public key certificate.

3.2 Kubernetes Components and Their Certificates

1. Kube API Server
2. ETCD Server
3. Kubelet on Worker Nodes

3.3 Client Components and Their Certificates

1. Administrator (kubectl/REST API)
2. Scheduler
3. Kube Controller Manager
4. Kube Proxy

4 TLS in Kubernetes Certificate Creation

4.1 1. Generating CA Certificates

The process involves generating a private key, creating a Certificate Signing Request (CSR) that includes the CA's common name, and finally signing the CSR with the private key to produce the CA certificate. The completed process provides the CA with its private key (ca.key) and root certificate (ca.crt), which are essential for subsequently signing other certificates.

```
openssl genrsa -out ca.key 2048
openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr
openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt
```

4.2 2. Creating Client Certificates

4.2.1 2.1 Admin User Certificate

To generate a certificate for the admin user:

1. Create a private key for the admin.
2. Generate a CSR for the admin user specifying the common name (CN) and organizational unit (OU) to reflect group membership (e.g., `system:masters`). This consistency ensures that the admin identity is properly logged in audit trails and recognized in `kubectl` commands.
3. Sign the admin CSR with the CA certificate to produce the final admin certificate.

```
openssl genrsa -out admin.key 2048
openssl req -new -key admin.key -subj "/CN=kube-admin/O=system:masters" -out admi
openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt
```

The resulting `admin.crt` file functions as a secure credential, akin to a username and password pair, for authenticating the admin user with the Kubernetes API server.

A similar process is followed to generate client certificates for other components such as the scheduler, controller manager, and kube-proxy.

4.3 3. Using Client Certificates in API Requests

The admin certificate can be used to securely communicate with the server by specifying the key, certificate, and CA certificate in the request.

```
curl https://kube-apiserver:6443/api/v1/pods \
  --key admin.key --cert admin.crt --cacert ca.crt
```

The API server will respond with a JSON object listing the pods:

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods"
  },
  "items": []
}
```

4.4 4. Server-Side Certificates

For secure communication, both client and server certificates must trust the same CA root certificate. This certificate is used by both parties to verify the authenticity of the certificate they receive.

4.4.1 4.1 Etcd Server Certificate

If etcd is running as a cluster, remember to generate peer certificates to secure inter-member communications. Once created, the certificates are referenced in the etcd configuration file (commonly, `etcd.yaml`). See the example below:

```
cat etcd.yaml
- --advertise-client-urls=https://127.0.0.1:2379
- --key-file=/path-to-certs/etcdserver.key
```

```
- --cert-file=/path-to-certs/etcdserver.crt
- --client-cert-auth=true
- --data-dir=/var/lib/etcd
- --initial-advertise-peer-urls=https://127.0.0.1:2380
- --initial-cluster=master=https://127.0.0.1:2380
- --listen-client-urls=https://127.0.0.1:2379
- --listen-peer-urls=https://127.0.0.1:2380
- --name=master
- --peer-cert-file=/path-to-certs/etcdpeer1.crt
- --peer-client-cert-auth=true
- --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
- --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
- --snapshot-count=10000
- --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

The `--trusted-ca-file` option ensures that etcd client connections are authenticated using the CA certificate.

4.5 5. Kube API Server Certificates

4.5.1 5.1 Creating the API Server Certificate

Start by generating a CSR for the API server:

```
openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out apiserver.csr
```

Then, create an OpenSSL configuration file (e.g., `openssl.cnf`) to include all necessary SANs:

```
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name


[v3_req]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names


[alt_names]
```

```
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
IP.1 = 10.96.0.1
IP.2 = 172.17.0.87
```

After configuring the CSR with the SANs, sign the certificate using your CA certificate and key. Specify the final certificate parameters in your kube-apiserver configuration, as shown in the configuration snippet below:

```
ExecStart=/usr/local/bin/kube-apiserver \
  --advertise-address=${INTERNAL_IP} \
  --allow-privileged=true \
  --apiserver-count=3 \
  --authorization-mode=Node,RBAC \
  --bind-address=0.0.0.0 \
  --enable-swagger-ui=true \
  --etcd-cafile=/var/lib/kubernetes/ca.pem \
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \
  --etcd-servers=https://127.0.0.1:2379 \
  --event-ttl=1h \
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-kubelet-client.crt \
  --kubelet-client-key=/var/lib/kubernetes/apiserver-kubelet-client.key \
  --kubelet-https=true \
  --runtime-config=api/all \
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \
  --service-cluster-ip-range=10.32.0.0/24 \
  --service-node-port-range=30000-32767 \
  --client-ca-file=/var/lib/kubernetes/ca.pem \
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \
  --v=2
```

5 View Certificate Details

5.1 Understanding Your Cluster Setup

There are several methods for deploying a Kubernetes cluster, and each has its own approach to generating and managing certificates.

- If you deploy a cluster from scratch, you may generate and configure all certificates manually (as explored in a previous lesson).
- If you use an automated provisioning tool like kubeadm, certificate generation and configuration are handled for you. In this case, Kubernetes components are deployed as pods instead of OS services.

5.1.1 Native Service Deployment

When Kubernetes components are deployed as native services, you can review service files to understand the certificate configuration

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=172.17.0.32 \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certfile=/var/lib/kubernetes/kubelet-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/kubelet-client.key \\
  --kubelet-https=true \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/kube-apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/kube-apiserver-key.pem \\
  --v=2
```

5.1.2 Deployment Using kubeadm

When using kubeadm, components such as the kube-apiserver are defined as pods in manifest files

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
    - command:
      - kube-apiserver
      - --authorization-mode=Node,RBAC
      - --advertise-address=172.17.0.32
      - --allow-privileged=true
      - --client-ca-file=/etc/kubernetes/pki/ca.crt
      - --disable-admission-plugins=PersistentVolumeLabel
      - --enable-admission-plugins=NodeRestriction
      - --enable-bootstrap-token-auth=true
      - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
      - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
      - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
      - --insecure-port=0
      - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
      - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
      - --proxy-client-certfile=/etc/kubernetes/pki/apiserver-kubelet-client.crt
      - --proxy-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
      - --request-timeout=30s
```