What are Kubernetes Secrets and Service Accounts?

⋮ 2021/02/24

Brian McClain & Tiffany Jernigan

In software, there's often data that you want to keep separate from your build process. These could be simple configuration properties, such as URLs or IP addresses, or more sensitive data, such as usernames and passwords, OAuth tokens or TLS certificates. In Kubernetes, these are referred to as Secrets.

Secrets

It's worth noting that while the name "secret" may imply "secure", there are some qualifiers. By default, all secrets are stored unencrypted in `etcd`. As of Kubernetes 1.13 though, operators are given the option of encrypting data at rest in etcd. Additionally, you can integrate with an external Key Management Service, such as Google Cloud KMS or HashiCorp Vault. This guide doesn't cover these topics, but the above links are a great start to learn more.

All examples used in this guide can be found on GitHub.

Before you can get started using secrets, you first need to create a secret. As you may expect, this can be done by defining an object of kind `Secret`:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: bXl1c2VybmFtZQo= #Base64 encoded value of "myusername"
  password: bXlwYXNzd29yZAo= #Base64 encoded value of "mypassword"
```

Secrets in Kubernetes are, at their most basic form, a collection of keys and values. The above example creates a secret named `mysecret` with two keys: `username` and `password`. There's one very important thing to note though, which is that the values of these key/value pairs are encoded as base64. Remember that base64 is an encoding algorithm, not an encryption algorithm. This is done to help facilitate data that may not be entirely alpha-numeric, and instead could include binary data, non-ASCII data, etc. You apply can this YAML as you would if you were creating any other Kubernetes object:

```
kubectl apply -f https://raw.githubusercontent.com/BrianMMcClain/k8s-secrets-and-sa/main/secret-base64.yaml
```

Once applied, you can see that while you can get the secret with `kubectl`, it avoids printing the values of each key by default:

```
$ kubectl describe secret mysecret

Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations:
Type:         Opaque

Data
====
password:  11 bytes
username:  11 bytes
```

Of course, if you want to see the base64-encoded contents of the secret, you can still fetch them with a slightly different command:

```
$ kubectl get secret mysecret -o yaml

apiVersion: v1
data:
  password: bXlwYXNzd29yZAo=
  username: bXl1c2VybmFtZQo=
kind: Secret
...
```

Great! With your secret created, it's time to start creating pods to use it! You're faced with another decision, however, since Kubernetes provides a couple of methods for presenting secrets to a pod. The first example you'll look at is mounting them as files in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-as-file
spec:
```

```
  containers:
  - name: secret-as-file
    image: nginx
    volumeMounts:
    - name: mysecretvol
      mountPath: "/etc/mysecret"
      readOnly: true
  volumes:
  - name: mysecretvol
    secret:
      secretName: mysecret
```

Here, a new pod named `secret-as-file` is created from the NGINX Docker image.

NOTE: The nginx container image is used here simply because it's an easily accessible long-running process, this would look the same for your own container image.

There are two sections to point out, the first being the `volumes` section, which defines a new volume. Kubernetes has many different types of volumes to choose from, but for this case you're specifically interested in creating a volume of type `secret`. These volumes are backed by `tmpfs`, a RAM-based file system, rather than written to a persistent disk. Secret volumes require you to define the secret to mount (in the `secretName` field), and for each key in your secret, it creates a file that contains the key's value. You can see this in action by applying this YAML and then listing the files at the `mountPath`:

```
kubectl apply -f https://raw.githubusercontent.com/BrianMMcClain/k8s-secrets-and-sa/main/pod-secret-as-file.yaml
```

```
$ kubectl exec secret-as-file -- ls /etc/mysecret

password
username

$ kubectl exec secret-as-file -- cat /etc/mysecret/username

myusername
```

As you can see, there are two files in the volume that was created: `password` and `username`. If you print out the contents of the `username` file, you can see the secret's value of `myusername`.

```
$ kubectl exec secret-as-file -- cat /etc/mysecret/username

myusername
```

Alternatively, secrets can also be presented to your container as environment variables. Consider the following YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-as-env
spec:
  containers:
  - name: secret-as-env
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: password
```

Here, instead of defining volumes that reference your secret, two environment variables are defined and reference the secret name and key name. Applying this YAML allows you to retrieve these environment variables from a shell in the pod:

```
kubectl apply -f https://raw.githubusercontent.com/BrianMMcClain/k8s-secrets-and-sa/main/pod-secret-as-env.yaml
```

```
$ kubectl exec secret-as-env -- sh -c "echo \$SECRET_USERNAME"

myusername
```

As you can see, the value of your secret is stored in the `$SECRET_USERNAME` environment variable as defined!

Service Accounts

When you interact directly with Kubernetes, using `kubectl` for example, you're using a user account. When processes in pods need to interact with Kubernetes though, they use a service account, which describes the set of permissions they have within Kubernetes. The good news is that out of the box, all pods are given the `default` service account. Unless your Kubernetes administrator has changed the `default` service account though, the permissions are limited. If you run `kubectl` in a container on Kubernetes, it will automatically know where to find the cluster that it's running on. You can verify this by standing up a pod and running `kubectl version`, which will show information about the server it's connected to:

```
kubectl run -it kubectl --restart=Never --rm --image=brianmmcclain/kubectl-alpine -- /bin/bash
```

```
$ kubectl version

Client Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.4",
GitCommit:"c96aede7b5205121079932896c4ad89bb93260af", GitTreeState:"clean",
BuildDate:"2020-06-17T11:41:22Z", GoVersion:"go1.13.9", Compiler:"gc",
Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.2",
GitCommit:"52c56ce7a8272c798dbc29846288d7cd9fbae032", GitTreeState:"clean",
BuildDate:"2020-04-30T20:19:45Z", GoVersion:"go1.13.9", Compiler:"gc",
Platform:"linux/amd64"}
```

Notice that you haven't provided any credentials or configuration file. This information is provided by Kubernetes and the `default` service account. However, almost any attempt at interacting with the Kubernetes API will be greeted with denial:

```
$ kubectl get pods

Error from server (Forbidden): pods is forbidden: User
"system:serviceaccount:default:default" cannot list resource "pods" in API group
"" in the namespace "default"

$ exit
```

Note: If you need to check if you have permission to run a command before actually running it, you can use the `kubectl auth can-i` command:

```
$ kubectl auth can-i get pods
no
```

To address this, you can create a new service account with a wider set of permissions. This is demonstrated in the following YAML:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-read-role
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

---
apiVersion: v1
kind: ServiceAccount
```

```
metadata:
  name: pod-read-sa

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pod-read-rolebinding
  namespace: default
subjects:
- kind: ServiceAccount
  name: pod-read-sa
  apiGroup: ""
roleRef:
  kind: Role
  name: pod-read-role
  apiGroup: ""
```

Here, three things are created: a `Role` named “pod-read-role”, a `ServiceAccount`, and a `RoleBinding` to tie them together. Specifically, the `Role` gives access to the “get”, “watch” and “list” actions on the resource “pods”. Described more simply, this role allows you to read information about pods, but not write or delete information about pods. This will allow you to do things like `kubectl get pods`, but not `kubectl delete pod`. You can see this in action by applying this YAML, creating a pod with this service account and running the commands yourself:

```
kubectl apply -f https://raw.githubusercontent.com/BrianMMcClain/k8s-secrets-and-sa/main/role-sa-pod-read.yaml
```

```
kubectl run -it --restart=Never --rm kubectl-with-sa --image=brianmmcclain/kubectl-alpine --serviceaccount=pod-read-sa -- /bin/bash
```

```
$ kubectl get pods

NAME              READY   STATUS    RESTARTS   AGE
kubectl           1/1     Running   1          22s
kubectl-with-sa   1/1     Running   0          6s
secret-as-env     1/1     Running   0          3h40m
secret-as-file    1/1     Running   0          4h10m

$ kubectl delete pod secret-as-file

Error from server (Forbidden): pods "secrets-as-file" is forbidden: User "system:serviceaccount:default:pod-read-sa" cannot delete resource "pods" in API group "" in the namespace "default"

$ exit
```

Finally, the combination of secrets and service accounts can be leveraged to pull container images from private registries by using the `imagePullSecrets` configuration property. You can create a secret from the command line with the following command:

```
kubectl create secret docker-registry myregistrykey --docker-server=DUMMY_SERVER \
        --docker-username=DUMMY_USERNAME --docker-password=DUMMY_DOCKER_PASSWORD \
        --docker-email=DUMMY_DOCKER_EMAIL
```

Note: The secret used here isn’t exposed to the pod in the same way that you’ve seen earlier. The processes inside the containers of the pod don’t have access to this information. Instead, Kubernetes knows that it needs to use these credentials to pull the container images.

Once you create the secret by filling in your registry’s server, username, password, and email, you can create a service account, or edit an existing one, to use this secret when pulling container images. For example, you can add this to the `default` service account. Make note, however, that this will overwrite any `imagePullSecret` previously set:

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
```

Since this adds the `imagePullSecrets` property to the default service account, any pod that you create without specifying a different service account will have these permissions. However, it's worth noting that you can also specify `imagePullSecrets` on an individual pod if it fits your deployment model better.

Cleanup

To remove the resources that you've created, you can use `kubectl delete -f` command and provide the file names used when applying them:

```
kubectl delete -f <insert file>
```

Learn More

As with all things Kubernetes, the best place to go to keep learning is the official documentation, which covers secrets and service accounts in even greater detail. You can also see where these are used in other guides, such as Getting Started with kpack and Getting Started with Tekton.