Integrate Azure Key Vault With Azure Kubernetes Service

Making secrets from Key Vault available to your application as environment variables

Noah Ispas
Dec 22, 2020 · 10 min read

Photo by Bradford Nicolas on Unsplash

If you are using Kubernetes to deploy your microservices, at some point you will have to think about how to handle your secrets. With secrets, I mean for example credentials for your data services or certificates that your services need to gain access to other

services. You don't want to hard code secrets in your code because it restricts you in terms of flexibility and it is an anti-pattern to add them to your source control.

In the cloud world of volatile microservices, it is a good practice to inject any kind of config through environment variables according to the 12 Factor App.

Kubernetes itself provides a mechanism to handle secrets and make them available either by mounting them into pods as a volume or injecting them into environment variables. One downside of that is, that they are stored as base64 encoded strings. Another downside is that you have to create them using the `kubectl` command-line tool, or defining a manifest for it.

That's why a centralized place for managing secrets like the Azure Key Vault comes in handy. It has some solid encryption for secrets and provides better usability. You can either use the user interface or the Azure CLI. That way, also people that are not that familiar with using `kubectl` can add or change secrets.

We will go through the process of utilizing the best of all involved technologies in order to provide a secret injection from Azure Key Vault into services, that run in Azure Kubernetes Service (AKS), through environment variables.

Prerequisites

Here are some prerequisites that you will need to walk through the whole process:

- Azure CLI installed
- Get an Azure Subscription where you have a global admin role
- kubectl installed
- helm installed
- jq installed if you are working on a Linux system. It is not a strict prerequisite, but the article contains some commands based on a Linux shell.

Create an AKS instance

The first thing we are going to create is the Kubernetes Cluster. Therefore you first have to log in to your Azure account. The following command will open a browser with a Single Sign-on site where you can enter your credentials.

```
az login
```

After logging in successfully, the CLI will list all available subscriptions for your account. Select your subscription.

```
az account set --subscription <subscription id>
```

Now we are good to go and create a simply configured Kubernetes cluster.

```
az group create --name aks2akvrg --location eastus
az aks create --resource-group aks2akvrg --name myk8s --node-count 1 --generate-ssh-keys --enabl
az aks get-credentials --resource-group aks2akvrg --name myk8s
```

create-aks-cluster.sh hosted with ♡ by GitHub view raw

The listed commands will create a resource group named **aks2akvrg** in the **eastus** region and a Kubernetes cluster inside. The Kubernetes cluster is named **myk8s** and contains 1 worker node. The last command will prepare your `kubectl` to be connected to the freshly created **myk8s** Kubernetes cluster.

Create an Azure Key Vault

Next, we will create an Azure Key Vault and already add a secret. Therefore we need the following commands.

```
az keyvault create --name "myk8skv" --resource-group "aks2akvrg" --location eastus
az keyvault secret set --vault-name "myk8skv" --name "ExamplePassword" --value "hVFkk965BuUv"
```

create-azure-key-vault.sh hosted with ♡ by GitHub view raw

After running those commands, you should have an Azure Key Vault named **myk8skv** in our **aks2akvrg** resource group as well as your first secret inside the Key Vault named **ExamplePassword**. This was the simple part.

Allow AKS to access Azure Key Vault

The more complicated part, as we have our Kubernetes Cluster and the Key Vault in place now, is to connect both. This basically means we have to allow Kubernetes to access the Key Vault.

CSI Secret Store Provider for Azure

The first step to connect the Kubernetes Cluster with the Key Vault is to add the CSI Secret Store Provider. The provider consists of two components. One is responsible to get the secrets out of the Key Vault while the other one is responsible for mounting the secrets into Kubernetes pods. Mounting secrets into Kubernetes pods refers to the build-in way of secret handling.

```
1  helm repo add csi-secrets-store-provider-azure https://raw.githubusercontent.com/Azure/secrets-s
2  helm install csi-secrets-store-provider-azure/csi-secrets-store-provider-azure --generate-name
```

install-azure-csi.sh hosted with ♡ by GitHub view raw

With these commands we have prepared the connection between Kubernetes and Key Vault, now we need to care about authentication. Just check if the CSI provider was added properly by invoking the following command:

```
kubectl get pods
```

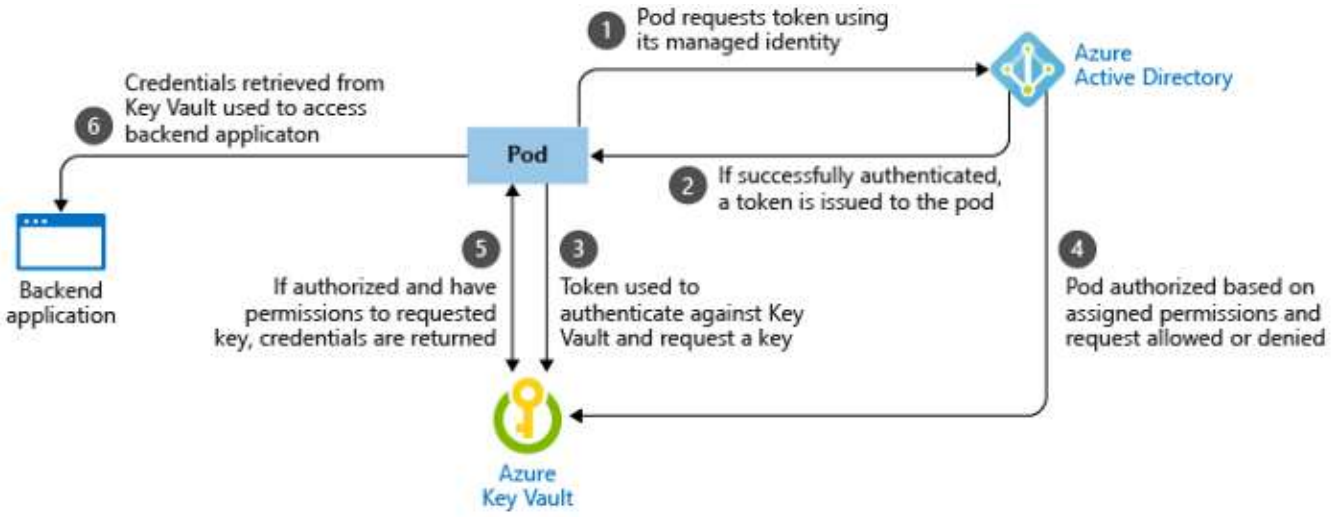
This should result in an output like this

```
NAME                                                READY   STATUS    RESTARTS   AGE
csi-secrets-store-provider-azure-1605956347-s745r   1/1     Running   0          38s
csi-secrets-store-provider-azure-1605956347-secrets-store-ccxbp   3/3     Running   0          37s
```

Authenticate Kubernetes against Key Vault

Access to Azure Key Vault is managed by the Azure Active Directory. In order to allow Kubernetes to get secrets out of the Key Vault, it has to authorize against the Key Vault through the Active Directory.

You can use either managed identities or a service principal to achieve that. We will focus on the method that uses a managed identity. If you are interested in the method that uses a service principal, have a look at this article. Both methods are described there.



https://docs.microsoft.com/de-de/azure/aks/developer-best-practices-pod-security

The picture shows the flow of a Kubernetes pod obtaining a secret from the Key Vault. The following steps will walk you through the preparation of everything that is needed for the flow.

Identity Permissions for Kubernetes

First, we need to grant Kubernetes and it's nodes access to managed identities of Azure. Therefore we first need to get the **clientId** and the **resource-group** of the Kubernetes nodes. After that, we can create the respective role assignments. Therefore run the following commands.

```
clientId=`az aks show --name myk8s --resource-group aks2akvrg |jq -r .identityProfile.kubeletide
nodeResourceGroup=`az aks show --name myk8s --resource-group aks2akvrg |jq -r .nodeResourceGroup
subId=`az account show | jq -r .id`
az role assignment create --role "Managed Identity Operator" --assignee $clientId --scope /subsc
az role assignment create --role "Managed Identity Operator" --assignee $clientId --scope /subsc
az role assignment create --role "Virtual Machine Contributor" --assignee $clientId --scope /sub
```

create-role-assigments.sh hosted with ♡ by GitHub view raw

AAD Pod Identity

Now, we can install the **aad pod identity**, which will allow a Kubernetes pod to make use of a Managed Identity in order to authenticate against the Key Vault.

```
helm repo add aad-pod-identity https://raw.githubusercontent.com/Azure/aad-pod-identity/master/c
helm install pod-identity aad-pod-identity/aad-pod-identity
```

create-aad-pod-identity.sh hosted with ♡ by GitHub view raw

Again, you should check if the respective pods are being deployed by running `kubectl get pods` which should result in a similar output:

Finally, create the Azure Managed Identity

```
az identity create -g aks2akvrg -n aks2kvIdentity
```

Read Permissions for Key Vault

The new Managed Identity will need to have read access on the Key Vault as well as the permission to grab secrets. Therefore run the following commands:

```
clientId=`az identity show --name aks2kvIdentity --resource-group aks2akvrg |jq -r .clientId`
principalId=`az identity show --name aks2kvIdentity --resource-group aks2akvrg |jq -r .principal
subId=`az account show | jq -r .id`
az role assignment create --role "Reader" --assignee $principalId --scope /subscriptions/$subId/
az keyvault set-policy -n myk8skv --secret-permissions get --spn $clientId
az keyvault set-policy -n myk8skv --key-permissions get --spn $clientId
```

set-key-vault-permissions.sh hosted with ♡ by GitHub view raw

Create Azure Identity Binding

We have the AAD pod identity helm chart installed and the Managed Identity created with the access rules it needs. Now we need to connect them by creating an Azure Identity Binding. Create a YAML file with the following content and deploy it with `kubectl apply -f filename.yaml` . You can get the **subscriptionId** with `echo $subId`

and the **clientId** with `echo $clientId` if you followed the last commands showed in the gists.

```yaml
apiVersion: aadpodidentity.k8s.io/v1
kind: AzureIdentity
metadata:
  name: "aks-kv-identity"
spec:
  type: 0
  resourceID: /subscriptions/<subscription id>/resourcegroups/aks2akvrg/providers/Microsoft.Mar
  clientID: "<clientId>"
---
apiVersion: aadpodidentity.k8s.io/v1
kind: AzureIdentityBinding
metadata:
  name: azure-pod-identity-binding
spec:
  azureIdentity: "aks-kv-identity"
  selector: azure-pod-identity-binding-selector
```

identity-binding.yaml hosted with ♡ by GitHub view raw

You can see that within the Azure Identity Binding manifest we reference our Managed Identity and we specify a selector. The selector will be used for pods that should be able to actually use the Managed Identity in order to authenticate against the Key Vault through the AAD.

Inject Secrets as Environment Variables

If you followed all the steps and everything went fine, congratulations on completing the most tricky part. The next steps will be much easier and more straight forward.

Create Custom Secret Provider

We created all the required components for the integration of Key Vault with Kubernetes. Now it's time to actually grab secrets from the Key Vault. We have to be explicit here and create a Secret Provider Class, where we define all the secrets we need to import from the Key Vault.

Please create a YAML file with the following content. Note, that the `<tenant id>` should reference the **tenantId** of the Key Vault. You can retrieve it using this command: `az keyvault show -n myk8skv | grep tenantId`

```yaml
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: spc-myk8skv
spec:
  provider: azure
  secretObjects:
  - secretName: test-secret
    data:
    - key: key
      objectName: ExamplePassword
    type: Opaque
  parameters:
    usePodIdentity: "true"
    useVMManagedIdentity: "false"
    userAssignedIdentityID: ""
    keyvaultName: "myk8skv"
    cloudName: ""
    objects:  |
      array:
        - |
          objectName: ExamplePassword
          objectType: secret
          objectVersion: ""
    resourceGroup: "aks2akvrg"
    subscriptionId: "<subscription id>"
    tenantId: "<tenant id>"
```

SecretProviderClass.yaml hosted with ♡ by GitHub view raw

There are two things of interest here.

1. In the **spec:parameter:objects** section, we define which secrets we want to actually grab from the key vault.

2. In the **spec:secrets** section we can specify which of the grabbed secrets should be created as Kubernetes Secrets.

Don't forget to deploy the Secret Provider Class with `kubectl apply -f filename.yaml`.

Mount Secrets into Application

By deploying the Secret Provider Class, the secrets will **not** be created in Kubernetes yet. This will only happen with the first pod, which mounts a volume utilizing CSI and

referencing our Secret Provider Class. Also, the pod must make use of the selector that we specified in the Azure Identity Binding.

Have a look at the following pod definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: inject-secrets-from-akv
  labels:
    aadpodidbinding: azure-pod-identity-binding-selector

spec:
  containers:
    - name: nginx
      image: nginx
      env:
      - name: SECRET
        valueFrom:
          secretKeyRef:
            name: test-secret
            key: key
      volumeMounts:
        - name: secrets-store-inline
          mountPath: "/mnt/secrets-store"
          readOnly: true
  volumes:
    - name: secrets-store-inline
      csi:
        driver: secrets-store.csi.k8s.io
        readOnly: true
        volumeAttributes:
          secretProviderClass: spc-myk8skv
```

secret-injection.yaml hosted with ♡ by GitHub view raw

Create a YAML file containing the content and deploy it using `kubectl apply -f filename.yaml`

A successful deployment should result in Kubernetes secrets being created. Also if you connect to the created pod, the **SECRET** environment variable should be filled with `hVFkk965BuUv .`

Troubleshooting

If your secret is not mounted to the pod the first thing I would do is to check if the Kubernetes secret is created by running `kubectl get secrets` . After that, you might want to check the different parts that play a role in the whole process:

1. Describe the pod that mounts secrets volume using `kubectl describe pod/inject-secrets-from-akv` and you will see some errors in case something went wrong.
2. The next place you could watch out for errors is the `aad-pod-identity-nmi-*` pod, where * is a random string. You will find the exact name of the pod by using `kubectl get pods | grep aad-pod-identity-nmi` . Look at the logs with `kubectl logs pod/aad-pod-identity-nmi-*` .
3. If you could not find any hint, also have a look at the logs of the aad-pod-identity-mic-* pods. Again * is a random string that you can find out using `kubectl get pods | grep aad-pod-identity-mic` . You will get 2 results out of that command. Look at the logs of both with the following command: `kubectl logs pod/aad-pod-identity-mic-*` .

Considerations

If you followed all steps of the article with success, you should be able to inject secrets of Key Vault into your applications running inside of AKS. Now, let's talk about some considerations for your future setup.

Volume Mounting

The secrets inside the Key Vault are only created as Kubernetes secrets with the first pod that mounts the volume pointing to the Secret Provider Class. What you can do is to add that volume to every pod that makes use of secrets. I don't know how you think about that, but somehow I don't like the idea that I need to mount a volume containing all the secrets of the Key Vault (at least all of them that are also defined in your Secret Provider Class) only to use a subset of them through environment variables.

The setup I used in my latest deployment scenario is the following. I have only one deployment containing a pod with the volume mounted that points to the Secret Provider Class. I named it `create-secrets-from-akv.yml` and it looks quite similar to the example deployment YAML I showed (named `inject-secrets-from-akv`). Maybe you get it straight away. I am assuming that this deployment is up and running before any other deployment is made which needs secrets. This way you don't need for every

other deployment to explicitly add and mount the volume pointing to the Secret Provider Class and still, Kubernetes secrets are created.

The downside is obvious. If that deployment is not deployed upfront, secrets will not be created inside of Kubernetes.

Add or update Secret in Key Vault

An important question that will arise is what happens if you update a secret inside the Key Vault. Another might be, how to add new secrets.

For new secrets, it's quite obvious that they will not be created inside Kubernetes, as every secret has to be defined explicitly in the Secret Store Provider Class. But at least for already existing secrets, I assumed that they will be updated in Kubernetes as well. Unfortunately, they will **not** be updated out of the box.

How do we handle updated or completely new secrets, then?

New secrets have to be added to Secret Provider Class. After that the procedure is the same for new or updated secrets:

- Redeploy the Secret Provider Class
- Redeploy the deployment that mounts the volume pointing to the Secret Provider Class. In my solution, I would need to completely delete and create again the `create-secrets-from-akv.yml` .

The fact that I need to completely delete the `create-secrets-from-akv.yml` made me rethink my approach. Why? After deleting the `create-secrets-from-akv.yml` deployment, also all the secrets will be removed from Kubernetes. No worries, deployments that reference the secrets, will still work. Somehow as soon as they inject the secrets, they seem to don't really care if they still exist in Kubernetes. At least that was what I observed.

Anyway, I was scared a little bit of the fact that I have to completely delete the `create-secrets-from-akv.yml` and that all the secrets would be gone until I redeploy. What if a

deployment was restarted at that moment, it would fail, right? Also somehow it does not feel right.

As a consequence, I will go the other way and for every deployment that needs secrets from the Key Vault, I will mount the volume that points to the Secret Provider Class. Even though I don't like the fact, that every deployment will then access every single secret defined Secret Store Provider. Even those, that it doesn't need.

Thank you for reading my article! I hope you enjoyed it, and everything went well. I am happy if I could help you. Please let me know if you find some wrong information. Also, I would love to hear if I missed something out or any other feedback. Maybe you have another clever idea of how to solve the challenge of making secrets from Key Vault available to AKS.

Further Reading

- https://docs.microsoft.com/de-de/azure/aks/developer-best-practices-pod-security
- https://medium.com/sela-devops-team/integrate-azure-key-vault-with-aks-using-akv2k8s-part-2-3-ee1d6682bf37
- https://medium.com/faun/how-to-use-secrets-from-azure-key-vault-in-azure-kubernetes-service-704973be5fc1
- https://mrdevops.io/introducing-azure-key-vault-to-kubernetes-931f82364354
- https://docs.microsoft.com/de-de/azure/key-vault/general/key-vault-integrate-kubernetes
- https://kubernetes.io/docs/concepts/configuration/secret/

Kubernetes Azure Azure Key Vault Secrets