Introducing Azure Key Vault to Kubernetes

Jon Arild Tørresdal
Mar 11, 2019 · 5 min read

The bank vault in NoMad Downtown LA. Image by Benoit Linero.

We created the Azure Key Vault to Kubernetes project as a way for us in Sparebanken Vest (Norwegian bank) to handle Azure Key Vault secrets securely in Kubernetes. It made perfect sense to us to open-source this project, as it is not our core business. Hopefully others can find it useful too.

Azure Key Vault to Kubernetes has two components that can be installed and run in the same cluster, or installed as single entities:

- Azure Key Vault Controller —synchronize Azure Key Vault secrets into native Kubernetes secrets, and keeps them updated
- Azure Key Vault Env Injector — transparently inject Azure Key Vault Secrets into applications running in Kubernetes containers, without revealing its content to Kubernetes resources, Etcd or its users

The recommendation is to have both installed, enabling native Kubernetes secrets when needed and transparently injecting environment variables for all other cases.

Another recommendation is to have a dedicated Azure Key Vault per Kubernetes cluster, and to store all secrets there and not in Kubernetes.

Authentication and Authorization

Having a dedicated Azure Key Vault per Kubernetes cluster also aligns with how authentication works with Azure Key Vault. By default it uses the same Service Principal that Kubernetes use when provisioning resources in Azure, like Load Balancers and VM's. Explicit authorization for Azure Key Vault must still be configured before the components will be able to access secrets. For details and other options for authentication, see the project documentation.

Why not use Azure Key Vault directly from the Application?

It is definitely possible and unfortunately quite common. The downside is a direct dependency to Azure Key Vault from the application, which locks the application to an environment where Azure Key Vault is available.

Adhering to the 12 Factor App, secrets is config, and should be injected into applications using environment variables. They are a language- and OS-agnostic standard. (ref 12 Factor App: https://12factor.net/config)

Trying out the Azure Key Vault Controller

By using the Controller, Azure Key Vault secrets gets synchronized to Kubernetes as native Kubernetes secrets.

In the example below, a Azure Key Vault certificate with an exportable key, gets synchronized as a Kubernetes TLS secret by defining a `AzureKeyVaultSecret` resource like this:

```yaml
apiVersion: spv.no/v1alpha1
kind: AzureKeyVaultSecret
metadata:
  name: my-first-azure-keyvault-certificate
  namespace: default
spec:
  vault:
    name: my-kv
    object:
      type: certificate
      name: test-cert
  output:
    secret:
      name: keyvault-certificate
      type: kubernetes.io/tls
```

2019-03-09-azure-keyvault-certificate.yaml hosted with ♡ by GitHub view raw

Things to note:

1. The Azure Key Vault object type is `certificate`
2. The Kubernetes output type is `kubernetes.io/tls`

When applying the above resource to Kubernetes, a native Kubernetes secret gets created by the Controller:

```yaml
apiVersion: v1
data:
  tls.crt: ...
  tls.key: ...
kind: Secret
metadata:
  name: keyvault-certificate
```

```
    name: keyvault-certificate
    namespace: default
  type: kubernetes.io/tls
```

2019-03-09-kubernetes-certificate.yaml hosted with ♡ by GitHub view raw

Trying out the Azure Key Vault Env Injector

> Working for a Norwegian bank, security is top priority. Certain secrets are so sensitive that no-one should be able to read its content. This is problematic in Kubernetes, as all secrets are plain text (base64 encoded).
>
> Even though locking down and hiding secrets from users is possible in Kubernetes, it adds complexity and reduces agility. We've found that using Azure Key Vault together with the Env Injector allows us to completely lock down certain secrets only allowing the application access.

Three things must be present in Kubernetes to transparently inject environment variables from Azure Key Vault into applications:

1. The Azure Key Vault Env Injector
2. A `AzureKeyVaultSecret` resource definition
3. An environment variable with a placeholder pointing to the `AzureKeyVaultSecret` resource above

Using the same example as with the Controller where the Azure Key Vault secret is a certificate with an exportable private key. The `AzureKeyVaultSecret` will be reused, but the `output` section is removed, since this is not needed by the Env Injector:

```yaml
apiVersion: spv.no/v1alpha1
kind: AzureKeyVaultSecret
metadata:
  name: my-first-azure-keyvault-certificate
  namespace: default
spec:
  vault:
    name: my-kv
    object:
      type: certificate
      name: test-cert
```

2019-03-09-azure-keyvault-certificate-no-output.yaml hosted with ♡ by GitHub view raw

Reference the `AzureKeyVaultSecret` using its name, from an environment variable inside the container spec:

```yaml
...
containers:
- name: alpine
  env:
  - name: MY_PUBLIC_KEY
    value: my-first-azure-keyvault-certificate@azurekeyvault?tls.crt
  - name: MY_PRIVATE_KEY
    value: my-first-azure-keyvault-certificate@azurekeyvault?tls.key
...
```

2019-03-09-pod-with-env-injection.yaml hosted with ♡ by GitHub view raw

Note how a "query string" is used to retrieve the private/public keys from the Azure Key Vault secret.

That's it! The content of the secret in Azure Key Vault will now be injected into the application through the environment variables `MY_PUBLIC_KEY` and `MY_PRIVATE_KEY` .

> The really cool thing about this solution is that Kubernetes (and its users) can only see the

placeholder for the secret, not the secret value. The secret value is ONLY available INSIDE the application.

Chasing the secret inside Kubernetes

Is the secret only available to the application? Are you sure about that? How about the `env` section of the container spec after the Pod is provisioned?

```
$ kubectl get pods my-pod-77cb7d647-gx68n -o yaml
```

```yaml
...
spec:
  containers:
    env:
    - name: MY_SECRET
      value: my-secret-from-azure@azurekeyvault
...
```

2019-03-09-pod-with-env-placeholder.yaml hosted with ♡ by GitHub view raw

Only the environment placeholder is visible

How about listing environment variables after `exec` into the container?

```
$ kubectl exec -it my-pod-77cb7d647-gx68n /bin/sh
```

```
# env
KUBERNETES_PORT=443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=my-pod-77cb7d647-9zftg
…
MY_SECRET=my-secret-from-azure@azurekeyvault
…
```

2019-03-09-exec-with-env-placeholder.sh hosted with ♡ by GitHub view raw

Only the environment placeholder is visible

So yes, the secret is ONLY visible to the application.

What is going on under the hood?

Diving straight to code, this is the magic sauce:

```
syscall.Exec(binary, os.Args[1:], environ)
```

The code above is from an application that is part of the Azure-Key-Vault-to-Kubernetes project, called `azure-keyvault-env` , and resposible for these key tasks:

1. Extract any environment variables containing the value `<something>@azurekeyvault`
2. Look up `AzureKeyVaultSecret` resources identified in 1.
3. Download secrets from Azure Key Vault using information from 2.
4. Add secrets as environment variables to the `environ` map in the code above
5. Execute the original application found in the container, injecting environment variables from 4.

For this to work the `azure-keyvault-env` application gets injected into the container and takes the original application's place. The big question is then: How does the `azure-keyvault-env` application get injected into the container?

The short answer is through a Kubernetes init-container after being configured in the Pod by a Kubernetes Mutating Admission Webhook. For further details, read how in the documentation and check out the code for these two components: `azure-keyvault-env` and `azure-keyvault-secrets-webhook` .

Summary

By providing these two tools to the community, secret management just got easier, more convenient and secure, by combining Kubernetes with Azure Key Vault.

If native Kubernetes secrets is needed, the Azure Key Vault Controller elegantly synchronize the secrets and add nice features like automatically convert Azure Key Vault certificates to TLS secrets in Kubernetes. At the same time, users can take advantage of all the features in Azure Key Vault, and feel confident that all secrets will sync nicely to Kubernetes if needed.

When it comes to the Env Injector, it is a pretty cool solution to transparently inject Azure Key Vault secrets to applications that way. Even cooler is the fact that the application is the only one that can see the secret!

Kubernetes Azure Vault Keyvault Azure Key Vault