

Quarkus - Working with HashiCorp Vault's Authentication

Working with Vault is typically a 2 step process:

- Logging in, which returns a client token
- Start using Vault using the client token, within the limits of what is allowed by the policies associated with the token

There are several Vault authentication methods supported in Quarkus today, namely:

- [Token](#): whenever you already have a token
- [Userpass](#): authenticate with a username and a password
- [AppRole](#): authenticate with a role id and a secret id (which can be seen as a *Userpass* for automated workflows - machines and services)
- [Kubernetes](#), which is applicable to workloads deployed into Kubernetes

This guide aims at providing examples for each of those authentication methods.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

Prerequisites

To complete this guide, you need:

- to complete the [Vault guide](#)
- roughly 30 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Docker installed
- For the Kubernetes authentication: A Kubernetes distribution to deploy the Quarkus application (Minishift, K3s, Docker Desktop, ...)

Token Authentication

We assume there is a Vault container and a PostgreSQL container running from the [Vault guide](#), and the root token is known.

First, create a new shell, `docker exec` into the container and set the root token:

```
docker exec -it dev-vault sh
/ # export VAULT_TOKEN=s.5VUS8pte13RqekCB2fmMT3u2
```

Create a token for the `vault-quickstart-policy` policy:

```
/ # vault token create -policy=vault-quickstart-policy
Key                               Value
---                               -
token                             s.JqMeE1UEyUb19F6zmMW0SWx6
token_accessor                     q1ynY9T7FDgbMKd3uST7RzLy
token_duration                     768h
token_renewable                   true
token_policies                    ["default" "vault-quickstart-policy"]
identity_policies                 []
policies                          ["default" "vault-quickstart-policy"]
```

Now use the generated client token in the application configuration:

```
quarkus.vault.authentication.client-
token=s.JqMeE1UEyUb19F6zmMW0SWx6
```

Compile and start the application:

```
./mvnw clean install
java -jar target/vault-quickstart-1.0-SNAPSHOT-runner.jar
```

Finally test the application endpoint:

```
curl http://localhost:8080/hello/private-key
```

You should see: `123456`.

Client Token using Response Wrapping

One drawback of this approach is that you expose a secret piece of information (i.e. the token) that

can give access to sensitive data in Vault. This requires ensuring that the application's configuration stays secure at all time. If an intruder was to access the client token, it would be able to start calling Vault on all endpoints permitted by the policy.

This risk can be mitigated using an approach called [Response Wrapping](#) (which used to be known as *Cubbyhole Authentication* in older versions of Vault). This principle is simple: instead of configuring the client token itself, we hide it inside a *Wrapping Token*, which we provide to the application. Upon startup, the application will unwrap the *Wrapping Token*, and fetch the real token from within. The additional level of security comes from the fact that the *Wrapping Token* is short lived (from a few seconds to a few minutes; basically just enough to start and unwrap the token), and can be unwrapped **only once**. If the *Wrapping Token* gets stolen and unwrapped, we will notice immediately because the legitimate application will get an error saying that the token is invalid.

With that in mind, let's create a new token and wrap it inside a *Wrapping Token* with a TTL of 1 minute:

```
/ # vault token create -wrap-ttl=60s -policy=vault-quickstart
-policy
Key                                Value
---                                -
wrapping_token:                    s.2cLMBoKhe1DK6W3uAFT2umXu
wrapping_accessor:                  ojvb0tmLzB5D47SzXGo9b3sR
wrapping_token_ttl:                 1m
wrapping_token_creation_time:       2020-04-14 16:05:20.990240428
+0000 UTC
wrapping_token_creation_path:       auth/token/create
wrapped_accessor:                    a4ITYQNnQtwC0UmV5DJMpCiG
```

Now let's use this wrapping token in the configuration:

```
quarkus.vault.authentication.client-token-wrapping-
token=s.2cLMBoKhe1DK6W3uAFT2umXu
```

Compile and run the application **without the tests**, you should be able now to curl the private key **123456** as before.

Stop the application, and execute tests with `./mvnw test`. They should fail with the following error:

```
ERROR: Failed to start application
io.quarkus.vault.VaultException: wrapping token is not valid or
does not exist; this means that the token has already expired (if
so you can increase the TTL on the wrapping token) or has been
consumed by somebody else (potentially indicating that the wrapping
token has been stolen)
```

Userpass Authentication

Normally the `userpass` auth method should already be enabled from the [Vault guide](#). If not, execute the following commands now:

```
vault auth enable userpass
vault write auth/userpass/users/bob password=sinclair
policies=vault-quickstart-policy
```

And simply specify the username and password for this user in the application configuration:

```
quarkus.vault.authentication.userpass.username=bob
quarkus.vault.authentication.userpass.password=sinclair
```

Test the application endpoint after compiling and starting the application again:

```
curl http://localhost:8080/hello/private-key
```

You should see: `123456`.



Userpass supports response wrapping as well for the `password` property, although it is more unusual to use this approach as response wrapping typically involves a technical workflow, which is better suited for `aprole`.

Approle Authentication

Approle is an authentication method suited for technical workflows. It relies on 2 pieces of information:

- role id can be compared to the user name in *Userpass*
- secret id plays the role of the `password`

To set up *Approle* you need to enable the `aprole` auth method, create an app role, and generate a role id and secret id:

```

/ # vault auth enable approle
/ # vault write auth/approle/role/myapprole policies=vault-quickstart-policy

/ # vault read auth/approle/role/myapprole/role-id
Key          Value
---          -
role_id      b15460ff-fea0-43fc-1002-a045fb60dfc4

/ # vault write -f auth/approle/role/myapprole/secret-id
Key          Value
---          -
secret_id    d2f13e1f-f32a-f60a-86d8-0b5cdaeb821b
secret_id_accessor 2acff656-d049-c4b3-a752-6125e69210ba

```

Add the appropriate properties:

```

quarkus.vault.authentication.app-role.role-id=b15460ff-fea0-43fc-1002-a045fb60dfc4
quarkus.vault.authentication.app-role.secret-id=d2f13e1f-f32a-f60a-86d8-0b5cdaeb821b

```

After compiling and running the application you should be able to curl it on the **private-key** endpoint to see the secret information **123456** as usual.

Approle using Response Wrapping

Similarly to direct client token authentication, it is possible to wrap the **secret-id**:

```

/ # vault write -wrap-ttl=60s -f
auth/approle/role/myapprole/secret-id
Key          Value
---          -
wrapping_token:      s.aSq7tcRqfeboZqLMPfa5gkXJ
wrapping_accessor:    u5EPZ0nqyIJN8mT44od67WMS
wrapping_token_ttl:    1m
wrapping_token_creation_time: 2020-04-14 16:59:25.482352967
+0000 UTC
wrapping_token_creation_path:
auth/approle/role/myapprole/secret-id

```

Replace the **secret-id** property with **secret-id-wrapping-token** in the configuration:

```
quarkus.vault.authentication.app-role.secret-id-wrapping-  
token=s.aSq7tcRqfeboZqLMPfa5gkXJ
```

Finally, recompile the application without tests (otherwise you are going to consume the wrapping token), launch it and curl the `private-key` endpoint. As usual, you should see `123456`.

Kubernetes Authentication

Vault provides an integration with Kubernetes to allow containers to authenticate with Vault using their Kubernetes JWT token located in `/var/run/secrets/kubernetes.io/serviceaccount`.

The setup is more involved than with the other auth methods because we need to allow Vault to talk to the master API to be able to validate the JWT token that the application will authenticate with.

auth-delegator

The first step involves creating a `vault-auth-sa` service account with `auth-delegator` role that Vault will use to communicate with the master API.

First create file `vault-auth-k8s.yml`:

```
---  
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: vault-auth-sa  
---  
kind: ClusterRoleBinding  
apiVersion: rbac.authorization.k8s.io/v1  
metadata:  
  name: vault-auth-delegator  
subjects:  
- kind: User  
  name: vault-auth-sa  
  namespace: default  
roleRef:  
  kind: ClusterRole  
  name: system:auth-delegator  
apiGroup: rbac.authorization.k8s.io
```

And apply it: `kubectl apply -f vault-auth-k8s.yml`.

Once the objects are created, we need to capture the JWT token of this service account, and grab the public certificate of the cluster:

```
secret_name=$(kubectl get sa vault-auth-sa -o json | jq -r
'.secrets[0].name')
token=$(kubectl get secret $secret_name -o json | jq -r
'.data.token' | base64 --decode)
echo token=$token
kubectl get secret $secret_name -o json | jq -r '.data."ca.crt"' |
base64 -D > /tmp/ca.crt
```

Vault

The next step requires to exec interactively with the root token into the Vault container to configure the Kubernetes auth method:

```
docker exec -it dev-vault sh
export VAULT_TOKEN=s.5VUS8pte13RqekCB2fmMT3u2
```

Once inside the pod, set the token variable to the value that was just printed in the console before, and recreate **ca.crt** with the same content as **/tmp/ca.crt** outside the container. Finally use **kubectl config view** to assess the url of your Kubernetes cluster:

```
token=...      => set the value printed in the console just before
vi ca.crt      => copy/paste /tmp/ca.crt from outside the
container
kubernetes_host => url from the kubectl config view (e.g.
https://kubernetes.docker.internal:6443)
```

Now we have all the information we need to configure Vault:

```
vault auth enable kubernetes

# configure master API access from Vault
vault write auth/kubernetes/config \
    token_reviewer_jwt=$token \
    kubernetes_host=$kubernetes_host \
    kubernetes_ca_cert=@ca.crt

# create vault-quickstart-role role
vault write auth/kubernetes/role/vault-quickstart-role \
    bound_service_account_names=vault-quickstart-sa \
    bound_service_account_namespaces=default \
    policies=vault-quickstart-policy \
    ttl=1h
```

Deploy the application

Add the following properties to the application (and remove any other authentication Vault property, plus replace `<host>` by the ip or name of the local host that is running the Vault and PostgreSQL containers, such that they will be accessible from the pod):

```
quarkus.vault.url=http://<host>:8200
quarkus.datasource.url = jdbc:postgresql://<host>:5432/mydatabase

quarkus.vault.authentication.kubernetes.role=vault-quickstart-role

quarkus.log.category."io.quarkus.vault".level=DEBUG
```

Now build the application:

```
./mvnw package -DskipTests
docker build -f src/main/docker/Dockerfile.jvm -t quarkus/vault-quickstart-jvm .
```

Create a `vault-quickstart-k8s.yml` file with:


```

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vault-quickstart-sa
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: vaultapp
  name: vaultapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vaultapp
  template:
    metadata:
      labels:
        app: vaultapp
    spec:
      serviceAccountName: vault-quickstart-sa
      containers:
        - image: quarkus/vault-quickstart-jvm
          imagePullPolicy: Never
          name: vaultapp
          ports:
            - containerPort: 8080
              name: vaultport
              protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  name: vaultapp
  labels:
    app: vaultapp
spec:
  type: NodePort
  ports:
    - name: vault
      port: 8080
      nodePort: 30400
  selector:
    app: vaultapp

```

And apply it: `kubectl apply -f vault-quickstart-k8s.yml`.

This will deploy the application, and make it available on port ~~30400~~ of the Kubernetes host.

You can check that the application has started from the logs:

```
kubectl get pods
pod=$(kubectl get pod -l app=vaultapp -o json | jq -r
'.items[0].metadata.name')
kubectl logs $pod
```

You should see:

```
exec java -Dquarkus.http.host=0.0.0.0
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
-XX:+UseParallelGC -XX:GCTimeRatio=4
-XX:AdaptiveSizePolicyWeight=90 -XX:MinHeapFreeRatio=20
-XX:MaxHeapFreeRatio=40 -XX:+ExitOnOutOfMemoryError -cp . -jar
/deployments/app.jar

--/ _ _ \ / / / / _ | / _ \ / / / / / / _ /
-/ / / / / / / _ | / , _ / , < / / / / \ \
--\ _ _ \ _ _ / / | _ / | _ / | _ \ _ _ / _ /
2020-04-15 18:30:00,983 DEBUG
[io.qua.vau.run.con.VaultConfigSource] (main) loaded vault runtime
config VaultRuntimeConfig{url=Optional[http://<host>:8200],
kubernetesAuthenticationRole=vault-quickstart-role,
kubernetesJwtTokenPath='/var/run/secrets/kubernetes.io/serviceaccou
nt/token', userpassUsername='', userpassPassword='***',
appRoleRoleId='', appRoleSecretId='***',
appRoleSecretIdWrappingToken='***', clientToken=***,
clientTokenWrappingToken=***, renewGracePeriod=PT1H,
cachePeriod=PT10M, logConfidentialityLevel=MEDIUM,
kvSecretEngineVersion=2, kvSecretEngineMountPath='secret',
tlsSkipVerify=false, tlsCaCert=Optional.empty, connectTimeout=PT5S,
readTimeout=PT1S}
2020-04-15 18:30:00,985 DEBUG
[io.qua.vau.run.con.VaultConfigSource] (main) loaded vault
buildtime config
io.quarkus.vault.runtime.config.VaultBuildTimeConfig@4163f1cd
2020-04-15 18:30:01,310 DEBUG
[io.qua.vau.run.cli.OkHttpClientFactory] (main) create
SSLSocketFactory with tls
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
2020-04-15 18:30:01,559 DEBUG [io.qua.vau.run.VaultAuthManager]
(main) authenticate with jwt at:
/var/run/secrets/kubernetes.io/serviceaccount/token => ***
```

```
2020-04-15 18:30:01,779 DEBUG [io.qua.vau.run.VaultAuthManager]
(main) created new login token: {clientToken: ***, renewable: true,
leaseDuration: 3600s, valid_until: Wed Apr 15 19:30:01 GMT 2020}
2020-04-15 18:30:01,802 DEBUG
[io.qua.vau.run.con.VaultConfigSource] (main) loaded 1 properties
from vault
2020-04-15 18:30:02,722 DEBUG [io.qua.vau.run.VaultAuthManager]
(Agroal_7305849841) extended login token: {clientToken: ***,
renewable: true, leaseDuration: 3600s, valid_until: Wed Apr 15
19:30:02 GMT 2020}
2020-04-15 18:30:03,274 INFO [io.quarkus] (main) vault-quickstart
1.0-SNAPSHOT (powered by Quarkus 999-SNAPSHOT) started in 4.255s.
Listening on: http://0.0.0.0:8080
2020-04-15 18:30:03,276 INFO [io.quarkus] (main) Profile prod
activated.
2020-04-15 18:30:03,276 INFO [io.quarkus] (main) Installed
features: [agroal, cdi, hibernate-orm, jdbc-postgresql, narayana-
jta, resteasy, vault]
```

Notice in particular the following log line:

```
authenticate with jwt at:
/var/run/secrets/kubernetes.io/serviceaccount/token => ***
```

Finally curl the **private-key** endpoint to make sure you can retrieve your secret:

```
curl http://localhost:30400/hello/private-key
```

You should see **123456**.