Quarkus - Working with HashiCorp Vault’s Transit Secret Engine

Vault’s Transit Secret Engine offers an "encryption as a service" functionality. It allows to store encryption keys into Vault, and provides services to encrypt/decrypt and sign/verify arbitrary pieces of data. This brings several advantages, such as:

- limited key exposure: keys never leave the Vault. Instead the data is sent to Vault to get encrypted/decrypted/signed/verified.
- reliable and consistent algorithms: algorithms are implemented by Vault. It supports a wide variety, and there is only one implementation used for a given algorithm type, regardless of the client’s technology.
- it relieves developers from having to embed cryptographic libraries into their applications.

In the context of Quarkus, the main (non administration) services are being covered:

- encrypt: encrypt some data and return the cipher text
- decrypt: decrypt the cipher text and return the clear data
- rewrap: reencrypt a cipher text using the most recent key version
- sign: sign a piece of data
- verify signature: verify that a signature is correct for a piece of data

See Vault Transit Secret Engine’s official documentation

> 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 "Starting Vault" section of the Vault guide
- roughly 15 minutes
- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- Docker installed

Setup

We assume there is a Vault running from the Vault guide, and the root token is known. The first step consists in activating the Transit Secret Engine, and creating the different keys:

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

vault secrets enable transit
# ==> Success! Enabled the transit secrets engine at: transit/

vault write -f transit/keys/my-encryption-key
# ==> Success! Data written to: transit/keys/my-encryption-key

vault write transit/keys/my-sign-key type=ecdsa-p256
# ==> Success! Data written to: transit/keys/my-sign-key
```

Note that you did not have to provide the key value. You only provide its name, and Vault will generate it for you and keep it secured.

Once the keys have been created, we now need to create a policy that provides access for it:

```
cat <<EOF | vault policy write vault-transit-quickstart-policy -
path "transit/encrypt/my-encryption-key" {
  capabilities = [ "update" ]
}
path "transit/decrypt/my-encryption-key" {
  capabilities = [ "update" ]
}
path "transit/sign/my-sign-key" {
  capabilities = [ "update"]
}
path "transit/verify/my-sign-key" {
  capabilities = [ "update" ]
}
EOF
# ==> Success! Uploaded policy: vault-transit-quickstart-policy
```

And finally, let’s add the `vault-transit-quickstart-policy` to user `bob` that was created in the Vault guide:

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

To check that the configuration is correct, try logging in:

```
vault login -method=userpass username=bob password=sinclair
```

You should see:

```
Success! You are now authenticated. The token information displayed
below
is already stored in the token helper. You do NOT need to run
"vault login"
again. Future Vault requests will automatically use this token.

Key                    Value
---                    -----
token                  s.s93BVzJPzBiIGuYJHBTkG8Uw
token_accessor         0KNipTAgxWbxsrjixASNiwdY
token_duration         768h
token_renewable        true
token_policies         ["default" "vault-quickstart-policy"]
identity_policies      []
policies               ["default" "vault-quickstart-policy"]
token_meta_username    bob
```

Now set `VAULT_TOKEN` to the `token` above (instead of the root token), and try encrypting some data:

```
export VAULT_TOKEN=s.s93BVzJPzBiIGuYJHBTkG8Uw
# note: "my secret data" in base64 is "bXkgc2VjcmV0IGRhdGEK"
vault write transit/encrypt/my-encryption-key
plaintext=bXkgc2VjcmV0IGRhdGEK
```

You should see:

```
Key              Value
---              -----
ciphertext
vault:v1:vIQxsLANFbcfKofJL55zjoIXV6MqAzvjKUUQLGg5pWTz0W2Qab/B4nEJaQ
==
```

Encrypt and Decrypt

First, let’s create a simple Quarkus application with vault and jsonb capabilities:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.Alpha1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=vault-transit-quickstart \
    -DclassName="org.acme.quickstart.GreetingResource" \
    -Dpath="/hello" \
    -Dextensions="vault,resteasy-jsonb"
cd vault-transit-quickstart
```

Now, configure access to Vault from the `application.properties`:

```
# vault url
quarkus.vault.url=http://localhost:8200

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

Note that you did not need to specify the existence of a particular encryption key in the configuration. You only do so in special cases such as specifying the key type for upsert, or changing the signature algorithm, … Check the complete configuration for more information.

We can then add a new endpoint that will allow us to encrypt and decrypt data:

```
@Path("/transit")
@Produces(TEXT_PLAIN)
@Consumes(TEXT_PLAIN)
public class TransitResource {

    @Inject
    public VaultTransitSecretEngine transitSecretEngine;

    @POST
    @Path("/encrypt")
    public String encrypt(String clearData) {
        return transitSecretEngine.encrypt("my-encryption-key",
clearData);
    }

    @POST
    @Path("/decrypt")
    public String decrypt(String cipherText) {
        return transitSecretEngine.decrypt("my-encryption-key",
cipherText).asString();
    }
}
```

After compiling and starting the Quarkus application, let's encrypt some data:

```
curl -X POST --data 'some secret data' --header "Content-Type:
text/plain"  http://localhost:8080/transit/encrypt
# ==>
vault:v1:fN4P7WNjIegpb3lD/pSuhXvyONhGrI21gcKNcedk+5jpjguOw6JkqXYXlk
Y=
```

And decrypt back this cipher text:

```
curl -X POST --data
'vault:v1:fN4P7WNjIegpb3lD/pSuhXvyONhGrI21gcKNcedk+5jpjguOw6JkqXYXl
kY=' --header "Content-Type: text/plain"
http://localhost:8080/transit/decrypt
# ==> some secret data
```

Sign and Verify

Let's add 2 new methods to our `TransitResource`:

```
@POST
@Path("/sign")
public String sign(String input) {
    return transitSecretEngine.sign("my-sign-key", input);
}

public static class VerifyRequest {
    public String signature;
    public String input;
}

@POST
@Path("/verify")
@Consumes(APPLICATION_JSON)
public Response verify(VerifyRequest request) {
    transitSecretEngine.verifySignature("my-sign-key", request
.signature, request.input);
    return Response.accepted().build();
}
```

And start signing some data:

```
curl -X POST --data 'some secret data' --header "Content-Type:
text/plain"  http://localhost:8080/transit/sign
# ==>
vault:v1:MEUCIQDl+nE4y4E878bkugGG6FG1/RsttaQnoWfZHppeuk4TnQIgTGWTtM
hVPCzN8VH/EEr2qp5h34lI1bnEP6L1F+QQoPI=
```

And finally, let’s make sure the signature is matching our input data:

```
curl -v -X POST --data '{"input":"some secret
data","signature":"vault:v1:MEUCIQDl+nE4y4E878bkugGG6FG1/RsttaQnoWf
ZHppeuk4TnQIgTGWTtMhVPCzN8VH/EEr2qp5h34lI1bnEP6L1F+QQoPI="}'
--header "Content-Type: application/json"
http://localhost:8080/transit/verify
# ==> ... < HTTP/1.1 202 Accepted
```

Conclusion

The Transit Secret Engine is a powerful tool in the enterprise. We have seen the most obvious functions of the interface, but the rest of the methods or flavors should not be overlooked:

- For instance batch oriented methods are strongly recommended for mass operations (encrypt, decrypt, …)

- Transit contexts allow key derivation where one key is used to derive other keys for specific named contexts (e.g. person names, person addresses, …)
- Rewrapping allows to reencrypt data with the most recent key version when the Vault administrator decides to rotate keys

Feel free to look at the `VaultTransitSecretEngine` interface plus the dedicated Transit Secret Engine configuration properties in the Vault guide for all the details.