**Overview**

In this tutorial, we'll explore Hashicorp's Vault – **a popular tool used to securely manage sensitive information in modern application architectures**.

The main topics we'll cover, include:

* What problem does Vault try to solve
* Vault's architecture and main concepts
* Setup of a simple test environment
* Interacting with Vault using its command line tool

**2. The Problem with Sensitive Information**

Before digging into Vault, let's try to understand the problem it tries to solve: sensitive information management.

**Most applications need access to sensitive data in order to work properly**. For instance, an e-commerce application may have a username/password configured somewhere in order to connect to its database. It may also need API keys to integrate with other service providers, such as payment gateways, logistics, and other business partners.

Database credentials and API Keys are some examples of sensitive information that we need to store and make available to our applications in a secure way.

A simple solution is to store those credentials in a configuration file and read them at startup time. The problem with this approach is obvious, though. **Whoever has access to this file share the same database privileges our application have** – usually giving her full access to all stored data.

We can try to make things a bit harder by encrypting those files. This approach, however, will not add much in terms of overall security. Mainly, because our application must have access to the master key. Encryption, when used in this way, will only achieve a “false” sense of security.

Modern applications and cloud environments tend to add some extra complexity: distributed services, multiple databases, messaging systems and so on, all have sensitive information spread a bit everywhere, thus increasing the risk of a security breach.

So, what can we do? Let's Vault it!

**3. What is Vault?**

Hashicorp Vault addresses the problem of managing sensitive information – a *secret* in Vault's parlance. **“Managing” in this context means that Vault controls all aspects of a sensitive piece of information**: its generation, storage, usage and, last but not least, its revocation.

Hashicorp offers two versions of Vault. The open-source version, used in this article, is free to use, even in commercial environments. A paid version is also available, which includes technical support at different SLAs and additional features, such as HSM (Hardware Security Module) support.

**3.1. Architecture & Key Features**

Vault's architecture is deceivingly simple. Its main components are:

* A persistence backend – storage for all secrets
* An API server which handles client requests and performs operations on secrets
* A number of *secret engines,*one for each type of supported secret type

By delegating all secret handling to Vault, we can mitigate some security issues:

* Our applications don't have to store them anymore – just ask Vault when needed and discard it
* We can use short-lived secrets, thus limiting the “window of opportunity” where an attacker can use a stolen secret

Vault encrypts all data with an encryption key before writing it to the store. This encryption key is encrypted by yet another key – the master key, used only at startup.

A key point in Vault's implementation is that it doesn't store the master key in the server. **This means that not even Vault can access its saved data after startup.**At this point, a Vault instance is said to be in a “sealed” state.

Later on, we'll go through the steps needed to generate the master key and unseal a Vault instance.

Once unsealed, Vault will be ready to accept API requests. Those requests, of course, need authentication, which brings us to how Vault authenticates clients and decides what they can or can't do.

**3.2. Authentication**

**To access secrets in Vault a client needs to authenticate itself using one of the supported methods**. The simplest method uses Tokens, which are just strings sent on every API request using a special HTTP header.

When initially installed, Vault automatically generates a “root token”. This token is the equivalent as root superuser in Linux systems, so its use should be limited to a minimum. As a best practice, we should use this root token just to create other tokens with fewer privileges and then revoke it. This isn't a problem, though, since we can later generate another root token using unseal keys.

Vault also support other authentication mechanisms such as LDAP, JWT, TLS Certificates, among others. All those mechanisms build on top of the basic token mechanism: once Vault validates our client, it will provide a token that we can then use to access other APIs.

Tokens have a few properties associated with them. The main properties are:

* A set of associated *Policies*(see next section)
* Time-to-live
* Whether it can be renewed
* Maximum usage count

Unless told otherwise, tokens created by Vault will form a parent-child relationship. A child token can have at most the same level of privileges it parent has.

The opposite isn't true: we can – and usually do – create a child token with restrictive policies Another key point about this relationship: **When we invalidate a token, all child tokens, and their descendants are also invalidated**.

**3.3. Policies**

**Policies define exactly which secrets a client can access and which operations it can perform with them**. Let's see how a simple policy looks like:

|  |  |
| --- | --- |
| 1  2  3 | path "secret/accounting" {      capabilities = [ "read" ]  } |

Here we have used the HCL (Hashicorp's Configuration Language) syntax to define our policy. Vault also supports JSON for this purpose, but we'll stick to HCL in our examples since it is easier to read.

**Policies in Vault are “deny by default”**. A token attached to this sample policy will get access to secrets stored under *secret/accounting* and nothing else. At creation time a token can be attached to multiple policies. This is very useful because it allows us to create and test smaller policies and then apply them as required.

Another important aspect of policies is that they leverage lazy-evaluation. **This means that we can update a given policy and all tokens will be affected immediately.**

The policies described so far are also called Access Control List Policies, or ACL Policies. Vault also supports two additional policy types: EGP and RGP policies. Those are only available in the paid versions and extend the basic policy syntax with [Sentinel](https://www.vaultproject.io/docs/enterprise/sentinel/index.html) support.

When available, this allows us to take into account in our policies additional attributes such as time of the day, multiple authentication factors, client network origin, and so on. For instance, we can define a policy that allows access to a given secret only on business hours.

We can find more details on the policy syntax in [Vault's documentation](https://www.vaultproject.io/docs/concepts/policies.html).

**4. Secret Types**

Vault support a range of different secret types which address different use cases:

* *Key-Value:* simple static key-values pairs
* *Dynamically generated credentials*: generated by Vault upon request by a client
* *Cryptographic keys*: Used to perform cryptographic functions with client data

Each secret type is defined by the following attributes:

* A *mount* *point,*which defines its REST API prefix
* A set of operations exposed through the corresponding API
* A set of configuration parameters

A given secret instance is accessible via a *path*, much like a directory tree in a file system. **The first component of this path corresponds to the *mount point* where all secrets of this type are located**.

For instance, the string *secret/my-application*corresponds to the path under which we can find key-value pairs for *my-application*.

**4.1. Key-Value Secrets**

**Key-Value secrets are, as the name implies, simple pairs in the available under a given path**. For instance, we can store the pair *foo=bar* under the path */secret/my-application.*

Later on, we use the same path to retrieve the same pair or pairs – multiple pairs can be stored under the same path.

Vault support three kinds of Key-Value secrets:

* *Non-versioned Key-Pairs*, where updates replace existing values
* *Versioned Key-Pairs,* which keep up to a configurable number of old versions
* *Cubbyhole*, a special type of non-versioned key-pairs whose values are scoped to a given *access token* (more on those later).

Key-Value secrets are static by nature, so there is no concept of an associated expiration associated with them. The main use case for this kind of secret is to store credentials to access external systems, such as API keys.

In such scenarios credential updates are a semi-manual process, usually requiring someone to acquire new credentials and using Vault's command line or its UI to enter the new values.

**4.2. Dynamically Generated Secrets**

**Dynamic secrets are generated on the fly by Vault when requested by an application**. Vault support several types of dynamic secrets, including the following ones:

* Database credentials
* SSH Key Pairs
* X.509 Certificates
* AWS Credentials
* Google Cloud service accounts
* Active Directory accounts

All these follow the same usage pattern. First, we configure the secret engine with the details required to connect to the associated service. Then, we define one or more *roles,*which describe the actual secret creation.

Let's take the Database secret engine as an example. First, we must configure Vault with all user database connections details, including credentials from a preexisting user with admin privileges to create new users.

Then we create one or more roles (Vault roles, not Database roles) containing the actual SQL statements used to create a new user. Those usually include not only the user creation statement but also all the required *grant*statements required to access schema objects (tables, views and so on).

**When a client accesses the corresponding API, Vault will create a new temporary user in the database using the provided statements and return its credentials**. The client can then use those credentials to access the database during the period defined by the time-to-live attribute of the requested role.

Once a credential reaches its expiration time, Vault will automatically revoke any privilege associated with this user. A client can also request Vault to renew those credentials. The renewal process will happen only if supported by the specific database driver and allowed by the associated policy.

**4.3. Cryptographic Keys**

Secret engines of type handle cryptographic functions such as encryption, decryption, signing and so on. **All those operations use cryptographic keys generated and stored internally by Vault**. Unless explicitly told to do so, Vault will never expose a given cryptographic key.

The associated API allows clients to send Vault plain-text data and receive an encrypted version of it. The opposite is also possible: We can send encrypted data and get back the original text.

Currently, there is only one engine of this type: the *Transit* engine. This engine supports popular keys types, such as RSA and ECDSA, and also supports *Convergent Encryption.* When using this mode, a given plaintext value always result in the same cyphertext result, a property that is very useful in some applications.

For instance, we can use this mode to encrypt credit card numbers in a transaction log table. With convergent encryption, every time we insert a new transaction, the encrypted credit card value would be the same, thus allowing the use of regular SQL queries for reporting, searching and so on.

**5. Vault Setup**

<https://www.vaultproject.io/docs/configuration/index.html>

In this section, we will create a local test environment so we test the Vault's capabilities.

Vault's deployment is simple: just [download the package](https://www.vaultproject.io/downloads.html) that corresponds to our operating system and extracts its executable (*vault*or *vault.exe*on Windows) to some directory on our PATH. **This executable contains the server and is also the standard client**. [There is also an official Docker image available](https://hub.docker.com/_/vault/), but we will not cover it here.

Vault support a *development* mode, which is fine for some quick testing and getting used to its command line tool, but it is way too simplistic for real use cases: **all data is lost on restart and API access uses plain HTTP**.

Instead, we'll use file-based persistent storage and setup HTTPS so we can explore some of the real-life configuration details that can be a source of problems.

**5.1. Starting Vault Server**

Vault uses a configuration file using HCL or JSON format. The following file defines all the configuration needed to start our server using a file storage and a self-signed certificate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | storage "file" {    path = "./vault-data"  }  listener "tcp" {    address = "127.0.0.1:8200"    tls\_cert\_file = "./src/test/vault-config/localhost.cert"    tls\_key\_file = "./src/test/vault-config/localhost.key"  } |

Now, let's run Vault. Open a command shell, go to the directory containing our configuration file and run this command:

|  |  |
| --- | --- |
| 1 | $ vault server -config ./vault-test.hcl |

Vault will start and show a few initialization messages. They'll include its version, some configuration details and the address where the API is available. That's it – our Vault server is up and running.

**5.2. Vault Initialization**

Our Vault server now is running, but since this is its first run, we need to initialize it.

Let's open a new shell and execute the following commands to achieve this:

|  |  |
| --- | --- |
| 1  2  3 | $ export VAULT\_ADDR=https://localhost:8200  $ export VAULT\_CACERT=./src/test/vault-config/localhost.cert  $ vault operator init |

Here we have defined a few environment variables, so we don't have to pass them to Vault every time as parameters:

* *VAULT\_ADDR*: base URI where our API server will serve requests
* *VAULT\_CACERT*: Path to our server's certificate public key

In our case, we use the *VAULT\_CACERT* so we can use HTTPS to access Vault's API. We need this because we're using self-signed certificates. This would not be necessary for productions environments, where we usually have access to CA-signed certificates.

After issuing the above command, we should see a message like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Unseal Key 1: <key share 1 value>  Unseal Key 2: <key share 2 value>  Unseal Key 3: <key share 3 value>  Unseal Key 4: <key share 4 value>  Unseal Key 5: <key share 5 value>    Initial Root Token: <root token value>    ... more messages omitted |

The five first lines are the master key shares that we will later use to unseal Vault's storage. **Please note that Vault only displays the master key shares will during initialization – and never more.** **Take note and store them safely or we'll lose access to our secrets upon server restart!**

Also, please take note of the *root token*, as we will need it later. Unlike unseal keys, **root tokens can easily be generated at a later time**, so it is safe to destroy it once all configuration tasks are complete. Since we will be issuing commands later that require an authentication token, let's save the root token for now in an environment variable:

|  |  |
| --- | --- |
| 1 | $ export VAULT\_TOKEN=<root token value> (Unix/Linux) |

Let's see our server status now that we have initialized it, with the following command:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $ vault status  Key                Value  ---                -----  Seal Type          shamir  Sealed             true  Total Shares       5  Threshold          3  Unseal Progress    0/3  Unseal Nonce       n/a  Version            0.10.4  HA Enabled         false |

We can see that Vault is still sealed. We can also follow the unseal progress: “0/3” means that Vault needs three shares, but got none so far. Let's move ahead and provide it with our shares.

**5.3. Vault Unseal**

We now unseal Vault so we can start using its secret services. We need to provide any three of the five key shares in order to complete the unseal process:

|  |  |
| --- | --- |
| 1  2  3 | $ vault operator unseal <key share 1 value>  $ vault operator unseal <key share 2 value>  $ vault operator unseal <key share 3 value> |

After issuing each command vault will print the unseal progress, including how many shares it needs. Upon sending the last key share, we'll see a message like this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Key             Value  ---             -----  Seal Type       shamir  Sealed          false  ... other properties omitted |

The “Sealed” property is “false” in this case, which means that Vault is ready to accept commands.

**6. Testing Vault**

In this section, we will test our Vault setup using two of its supported secret types: Key/Value and Database. We will also show how to create new tokens with specific policies attached to them.

**6.1. Using Key/Value Secrets**

First, let's store secret Key-Value pairs and read them back. Assuming the command shell used to initialize Vault is still open, we use the following command to store those pairs under the *secret/fakebank* path:

|  |  |
| --- | --- |
| 1 | $ vault kv put secret/fakebank api\_key=abc1234 api\_secret=1a2b3c4d |

We can now recover those pairs at any time with the following command:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ vault kv get secret/fakebank  ======= Data =======  Key           Value  ---           -----  api\_key       abc1234  api\_secret    1a2b3c4d |

This simple test shows us that Vault is working as it should. We can now test some additional functionalities.

**6.2. Creating New Tokens**

So far we have used the root token in order to authenticate our requests. Since a root token is *way* too powerful, it is considered a best practice to use tokens with fewer privileges and shorter time-to-live.

Let's create a new token that we can use just like the root token, but expires after just a minute:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ vault token create -ttl 1m  Key                  Value  ---                  -----  token                <token value>  token\_accessor       <token accessor value>  token\_duration       1m  token\_renewable      true  token\_policies       ["root"]  identity\_policies    []  policies             ["root"] |

Let's test this token, using it to read the key/value pairs that we've created before:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ export VAULT\_TOKEN=<token value>  $ vault kv get secret/fakebank  ======= Data =======  Key           Value  ---           -----  api\_key       abc1234  api\_secret    1a2b3c4d |

If we wait a minute and try to reissue this command, we get an error message:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ vault kv get secret/fakebank  Error making API request.    URL: GET https://localhost:8200/v1/sys/internal/ui/mounts/secret/fakebank  Code: 403. Errors:    \* permission denied |

The message indicates that our token is no longer valid, which is what we've expected.

**6.3. Testing Policies**

The sample token we've created in the previous section was shorted lived, but still very powerful. Let's now use policies to create more restricted tokens.

For instance, let's define a policy that allows only read access to the *secret/fakebank* path we used before:

|  |  |
| --- | --- |
|  | $ cat > sample-policy.hcl <<EOF  path "secret/fakebank" {      capabilities = ["read"]  }  EOF  $ export VAULT\_TOKEN=<root token>  $ vault policy write fakebank-ro ./sample-policy.hcl  Success! Uploaded policy: fakebank-ro |

Now we create a token with this policy with the following command:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $ export VAULT\_TOKEN=<root token>  $ vault token create -policy=fakebank-ro  Key                  Value  ---                  -----  token                <token value>  token\_accessor       <token accessor value>  token\_duration       768h  token\_renewable      true  token\_policies       ["default" "fakebank-ro"]  identity\_policies    []  policies             ["default" "fakebank-ro"] |

As we've done before, let's read our secret values using this token:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ export VAULT\_TOKEN=<token value>  $ vault kv get secret/fakebank  ======= Data =======  Key           Value  ---           -----  api\_key       abc1234  api\_secret    1a2b3c4d |

So far, so good. We can read data, as expected. Let's see what happens when we try to update this secret:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ vault kv put secret/fakebank api\_key=foo api\_secret=bar  Error writing data to secret/fakebank: Error making API request.    URL: PUT https://127.0.0.1:8200/v1/secret/fakebank  Code: 403. Errors:    \* permission denied |

Since our policy does not explicitly allows writes, Vault returns a 403 – Access Denied status code.

**6.4. Using Dynamic Database Credentials**

As our final example in this article, let's use Vault's Database secret engine in order to create dynamic credentials. We assume here that we have a MySQL server available locally and that we can access it with “root” privileges. We will also use a very simple schema consisting of a single table – *account* .

The SQL script used to create this schema and the privileged user is available here.

Now, let's configure Vault to use this database. The database secret engine is not enabled by default, so we must fix this before we can proceed:

|  |  |
| --- | --- |
| 1  2 | $ vault secrets enable database  Success! Enabled the database secrets engine at: database/ |

We now create a database configuration resource :

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ vault write database/config/mysql-fakebank \    plugin\_name=mysql-legacy-database-plugin \    connection\_url="{{username}}:{{password}}@tcp(127.0.0.1:3306)/fakebank" \    allowed\_roles="\*" \    username="fakebank-admin" \    password="Sup&rSecre7!" |

The path prefix *database/config* is where all database configurations must be stored.  We choose the name *mysql-fakebank* so we can easily figure out to which database this configuration refers to. As for the configuration keys:

* *plugin\_name:* Defines which database plugin will be used. The available plugin names are described in [**Vault's docs**](https://www.vaultproject.io/docs/secrets/databases/index.html)
* *connection\_url*: This is a template used by the plugin when connecting to the database. Notice the {{username}} and {{password}} template placeholders. When connecting to the database, Vault will replace those placeholders by actual values
* *allowed\_roles*: Define which Vault roles (discussed next) can use this configuration. In our case we use “\*”, so its available to all roles
* *username & password:* This is the account that Vault will use to perform database operations, such as creating a new user and revoking its privileges

**Vault Database Role Setup**

The final configuration task is to create a Vault database role resource that contains the SQL commands required to create a user. We can create as many roles as needed, according to our security requirements.

Here, we create a role that grants read-only access to all tables of the *fakebank* schema:

|  |  |
| --- | --- |
| 1  2  3 | $ vault write database/roles/fakebank-accounts-ro \      db\_name=mysql-fakebank \      creation\_statements="CREATE USER '{{name}}'@'%' IDENTIFIED BY '{{password}}';GRANT SELECT ON fakebank.\* TO '{{name}}'@'%';" |

The database engine defines the path prefix*database/roles*as the location to store roles. *fakebank-accounts-ro* is the role name that we'll later use when creating dynamic credentials. We also supply the following keys:

* *db\_name*: Name of an existing database configuration. Corresponds to the last part of the path we used when creating the configuration resource
* *creation\_statements:* A list of SQL statement templates that Vault will use to create a new user

**Creating Dynamic Credentials**

Once we have a database role and its corresponding configuration ready, we generate new dynamic credentials with the following command:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ vault read database/creds/fakebank-accounts-ro  Key                Value  ---                -----  lease\_id           database/creds/fakebank-accounts-ro/0c0a8bef-761a-2ef2-2fed-4ee4a4a076e4  lease\_duration     1h  lease\_renewable    true  password           <password>  username           <username> |

The *database/creds* prefix is used to generate credentials for the available roles. Since we have used the *fakebank-accounts-ro*role, the returned username/password will be restricted to *select* operations.

We can verify this by connecting to the database using the supplied credentials and then performing some SQL commands:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ mysql -h 127.0.0.1 -u <username> -p fakebank  Enter password:  MySQL [fakebank]> select \* from account;  ... omitted for brevity  2 rows in set (0.00 sec)  MySQL [fakebank]> delete from account;  ERROR 1142 (42000): DELETE command denied to user 'v-fake-9xoSKPkj1'@'localhost' for table 'account' |

**We can see that the first *select* completed successfully, but we could not perform the *delete* statement**. Finally, if we wait for one hour and try to connect using those same credentials, we will not be able to connect anymore to the database. Vault has automatically revoked all privileges from this user

**7. Conclusion**

In this article have explored the basics of Hashicorp's Vault, including some background on the problem it tries to address, its architecture and basic use.

Along the way, we have created a simple but functional test environment that we´ll use in follow-up articles.

The next article will cover a very specific use case for Vault: **Using it in the context of Spring Boot application**. Stay tuned!