Codifying Vault Policies and Configuration

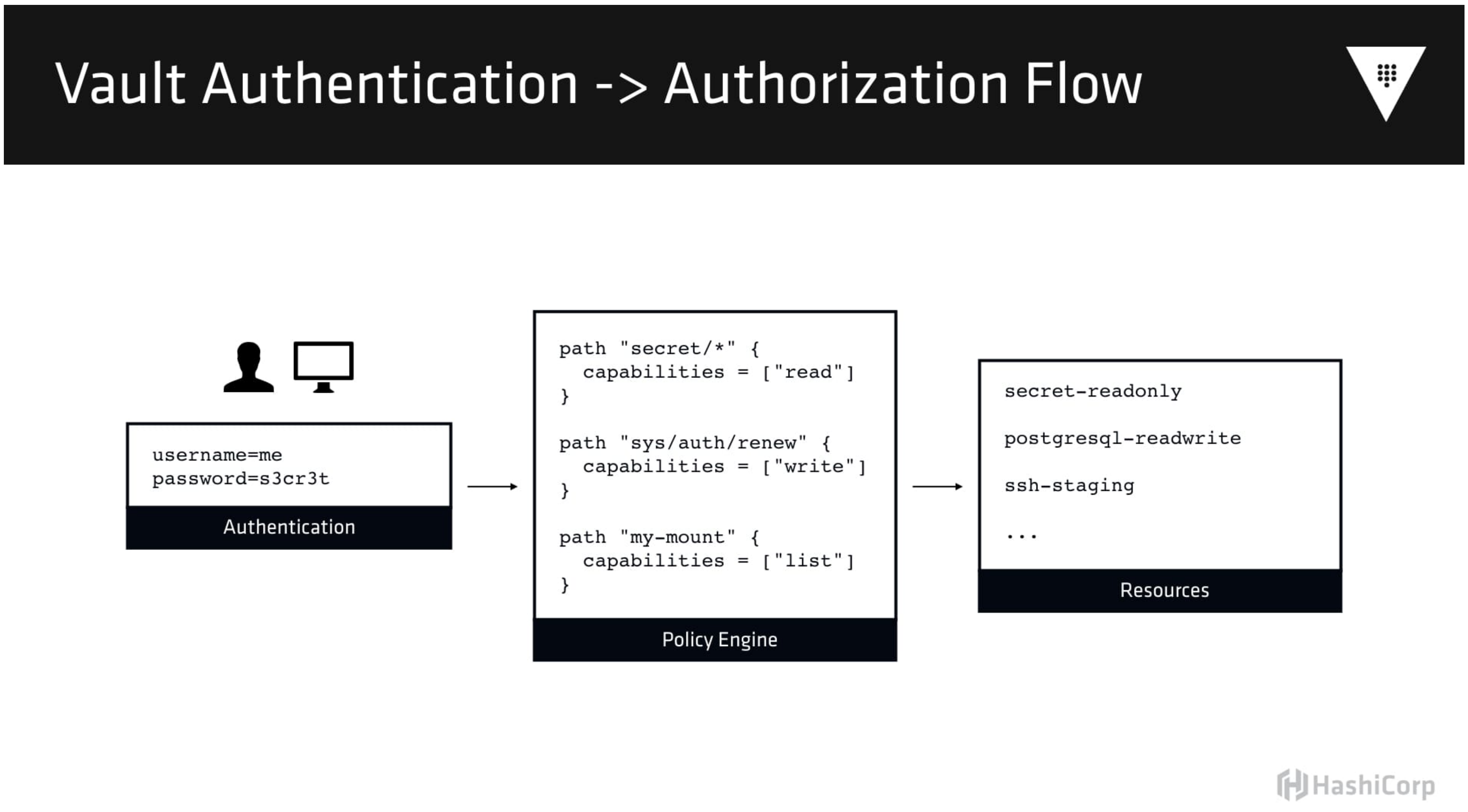
NOV 14 2016 SETH VARGO

One of the pillars behind the Tao of HashiCorp is "Automation through Codification". Recently I had the pleasure of participating in some very thoughtful discussions on whether Vault embodies that principle, specifically as it relates to Vault's configuration and policies.

This post discusses techniques for capturing your Vault policies and configurations in source control, providing repeatable workflows, continuous integration of policy testing, and much more.

Background

Vault is a tool for managing secrets. Specifically, Vault is a secret management and acquisition engine with robust policy and configuration options. Users or machines authenticate to Vault using an *authentication* backend like username/password, GitHub, LDAP, or AppRole, and Vault maps that authentication information to an *authorization* in the form of policies. Here is that same flow expressed in image form.



```
username=me
password=s3cr3t
```

```
path "secret/*" {
  capabilities = ["read"]
}

path "sys/auth/renew" {
  capabilities = ["write"]
}

path "my-mount" {
  capabilities = ["list"]
}
```

```
secret-readonly

postgresql-readwrite

ssh-staging

...
```

These pieces – the authentication, the policies, and the resources themselves – require configuration. Vault must be configured to talk to GitHub or LDAP. Vault must be configured to map authentications to a set of policies. Vault must be configured to communicate with those third-party resources such as databases or remote systems. Let us examine the steps required to connect Vault to a PostgreSQL server, give a user the ability to authenticate to Vault using a username and password, and then assign that authenticated user a policy that permits generating PostgreSQL credentials. (Note: this is not meant to be a comprehensive guide, for detailed information, please see the Vault documentation).

```
# Connect Vault to PostgreSQL
vault mount postgresql
vault write postgresql/config/connection connection_url="..."
vault write postgresql/config/lease lease=2h lease_max=36h
vault write postgresql/roles/readonly sql="CREATE USER ...;"

# Create a policy which permits generating those credentials
vault policy-write postgresql-readonly <<EOH
path "postgresql/creds/readonly" {
  capabilities = ["read"]
}
EOH

# Create an authentication (userpass) which maps that policy to a
# user on successful authentication
vault auth-enable userpass
vault write auth/userpass/users/me \
  password=s3cr3t \
  policies=postgresql-readonly
```

If you are just getting started with Vault, this is the easiest way to get started. However, you may be questioning how sustainable this process is for large Vault clusters or for maintaining the Vault server configuration over time. If you permit operators to manually configure the Vault server in this manner, there is little oversight, and it is prone to typographical errors.

API-Driven

Everything in Vault is an API; even the CLI is just a very thin wrapper around Vault's robust HTTP API. For example, here are the same set of commands from above, executed using `curl` instead of the Vault CLI:

```
CURL_OPT="-H X-Vault-Token:$VAULT_TOKEN -X POST"

# Connect Vault to PostgreSQL
curl ${CURL_OPT} $VAULT_ADDR/v1/sys/mounts/postgresql \
  -d '{"type":"postgresql"}'
curl ${CURL_OPT} $VAULT_ADDR/v1/postgresql/config/connection \
  -d '{"connection_url":"..."}'
curl ${CURL_OPT} $VAULT_ADDR/v1/postgresql/config/lease \
  -d '{"lease":"2h", "lease_max":"36h"}'
curl ${CURL_OPT} $VAULT_ADDR/v1/postgresql/roles/readonly \
  -d '{"sql":"CREATE USER ...;"}'

# Create a policy which permits generating those credentials
curl ${CURL_OPT} $VAULT_ADDR/v1/sys/policy/postgresql-readonly \
  -d '{"rules":"path \"postgresql/creds/readonly\" {\n  capabilities = [\"read\"]\n}"}'

# Create an authentication (userpass) which maps that policy to a
# user on successful authentication
curl ${CURL_OPT} $VAULT_ADDR/v1/sys/auth/userpass \
  -d '{"type":"userpass"}'
curl ${CURL_OPT} $VAULT_ADDR/v1/auth/userpass/users/me \
    -d '{"password":"s3cr3t", "policies":"postgresql-readonly"}'
```

Because Vault is API-driven, it is possible to automate these operations, without even installing the Vault CLI. Even better, because Vault's hierarchy closely resembles that of a filesystem, it is possible to codify these relationships in a single, simple repository.

Always be cautious when issuing Vault commands via the CLI, especially on shared machines, as secrets and sensitive data may appear in the shell history.

Layout and Design

The layout and design of this repository is simple - everything after the `v1/` in the API becomes a folder, and each payload becomes a JSON file. For example, a repository for the commands above might look something like this:

```
vault-policy/
  \_ data
    \_ auth/
      \_ userpass/
        \_ users/
          \_ me.json
    \_ postgresql/
      \_ config/
        \_ connection.json
        \_ lease.json
      \_ roles/
        \_ readonly.json
    \_ sys/
      \_ auth/
        \_ userpass.json
      \_ policy/
        \_ postgresql-readonly.json
      \_ mounts/
        \_ postgresql.json
```

Where each of those JSON files contains the contents of the payload, as JSON. Effectively we have mapped the above `curl` commands to a filesystem structure, by path.

This layout gives us powerful flexibility, since it permits a declarative syntax for specifying Vault configuration. Additionally, we have zero requirements on the Vault CLI, meaning we can apply these policies to a Vault server using a tool like `curl` or any language that has a built-in mechanism for making HTTP requests.

You **should not** store actual secrets or sensitive data in this layout. That would defeat the purpose of using Vault to manage secrets. These secrets should still be configured manually.

Provisioning

Next we need to apply the configuration described by this file structure to the Vault server, via the Vault API. I have intentionally called this "provisioning" and not "mirroring" or "configuring", because of an important caveat – if you delete something from the repository, it will **not** be deleted from Vault. This layout and subsequent script allows us to provision a Vault server from scratch, and the operations themselves are idempotent (safe to repeat).

```
#!/usr/bin/env bash
set -e

shopt -s nullglob

function provision() {
  set +e
  pushd "$1" > /dev/null
  for f in $(ls "$1"/*.json); do
    p="$1/${f%.json}"
    echo "Provisioning $p"
    curl \
      --silent \
      --location \
      --fail \
      --header "X-Vault-Token: ${VAULT_TOKEN}" \
      --data @"${f}" \
      "${VAULT_ADDR}/v1/${p}"
  done
  popd > /dev/null
  set -e
}

echo "Verifying Vault is unsealed"
vault status > /dev/null

pushd data >/dev/null
provision sys/auth
provision sys/mounts
provision sys/policy
provision postgresql/config
provision postgresql/roles
provision auth/userpass/users
popd > /dev/null
```

You should **never** store actual secrets or sensitive data in this repository. That would defeat the purpose of using Vault to manage secrets. These secrets should still be configured manually.

This script is not designed to be a complete solution to your Vault configuration needs, but rather serves as an excellent starting point for you to build your own. Due to some dependencies in the order of operations, you may need to configure mounts before you configure authentication, for example. This script not only gives you the flexibility to do so, but also provides the groundwork for writing something like this in your language of choice, such as Ruby, Python, or Go.

Some potential enhancements left to the reader include:

- Using `jq` to parse output and exit on certain errors
- Adding post-convergence testing, either using the Vault CLI or `curl`, which verifies the correct configuration and policies are in place

- Connecting this process to a centralized and controlled executor for this script
- Parse environment variables into the JSON so that sensitive values like the PostgreSQL `connection_url` is not saved in plain-text

It is important to note that this is **not** a mechanism for backing up Vault. While this can be used to bring a Vault cluster to a desired configuration, it does not contain any static or dynamically-generated secrets. You should still backup Vault's storage backend accordingly.

Bonus: Continuous Integration

If you have an existing continuous integration service, you can test configuration changes before pushing them into production by spinning up a Vault server in development mode and provisioning it. This will quickly catch any invalid syntaxes, bad paths, or accidental typographical errors. After the Vault server is successfully provisioned, you could use any number of tools to verify its state.

This example uses Travis CI, but it is largely applicable to any continuous integration service like Circle CI or Jenkins.

```
sudo: false

env:
  - VAULT_VERSION=0.6.2 VAULT_TOKEN=root

before_install:
  - scripts/install.sh

before_script:
  - vault server -dev -dev-root-token-id="$VAULT_TOKEN"

script:
  - scripts/provision.sh
```

One important thing to point out is the use of `-dev-root-token-id` to set the initial root token. This avoids the need to write a custom function to extract the token.

Conclusion

Vault is a powerful secret management solution complete with a robust HTTP API. This API enables us to easily codify configurations and policies without the need to build complex tooling and logic. We hope you enjoyed this short tutorial, and we are always open to new ideas. If you have a unique technique for configuring a Vault server, let us know by reaching out on Twitter or via email. Thank you for reading.

The sample repository and script for this post are available on GitHub.