The complexity of managing secrets

Managing secrets throughout the lifecycle of an application is not a trivial task. If you're too restrictive with access, implementations can become too complex. But if you have a lax approach, you'll increase the risk of a security breach. Other concerns should also be addressed while managing the secrets:

- How to handle external and internal threats (like hackers and disgruntled employees)
- How to securely share secret management responsibility among a set of stakeholders
- How to allow seamless access to both humans and services
- How to audit who accessed a secret at a certain date and time

Hashicorp Vault is an open source secret management and distribution tool that proposes an answer to these and other questions. In this blog post, I'll introduce the technology and provide a walkthrough—through a Proof of Concept example—describing how to install, initialize, and access the secrets using token authentication. This walkthrough provides an example closer to enterprise reality than the one found in the official tutorial, while not going into details of more complex implementations.

Hashicorp Vault

Hashicorp Vault is developed and maintained by our friends from Hashicorp, famous for making delightful and popular tools like Vagrant, Consul, and Terraform. Hashicorp Vault follows the same guiding principles as the company's other solutions, resulting in a product that is simple to use due to its modularity, yet powerful and flexible in how it can be implemented. There are many ways of describing Hashicorp Vault's features, but Seth Vargo, one of its maintainers, summarizes it in this interview by citing four main pillars:

- Mitigation of internal and external threats: Allowing Vault to be sealed/unsealed in response to crises, and supporting a variety of security backends.
- Dynamic secrets: Vault is able to generate credentials automatically for different backends.
- Lease renewal/revoke access to secrets: Vault allows you to specify TTL rules for a secret, enabling fine control over permissions.
- Auditing: Once enabled, every secret request will be logged, with the output directed to a file or to syslog.

Having said that, is Vault truly secure? Besides being available as open source, the tool has undergone a third party security audit by iSEC, with details discussed here and here.

Backends

Vault offers modular plug-in for three main areas: encrypted secret storage, authentication controls, and audit logs:

- Secret storage: This is the solution that will "host" the secrets. Available backends include AWS S3, Consul, Generic (file storage), among others.
- Authentication controls: The authentication mechanism used. Available backends are AWS EC2, LDAP, Github, Tokens, Username/password, and others.

- Audit logs: Where logs are sent. Available File or syslog. The specific mix of backends will depend on your project's requirements and constraints, and more than one backend can be enabled. Additional information on each backend, including API request commands to access them, can be found in the official documentation.

Accessing secrets using token authentication and response wrapping

One possible scenario while using Vault is when you have a set of clients that are previously defined and need access to secrets. For example, in order for Mary the developer to access a secret using her Github account, a Vault manager would need to enable the GitHub backend, and based on the associated policy, Mary would only need to authenticate to Vault using her GitHub credentials. A similar scenario would take place for sample-app-service, which has an AWS IAM-based service account and requires access to certain secrets.

However, recently I worked on a client project that had a different requirement. Using a combination of Jenkins, Chef, and Microsoft's Hyper-V manager, a json file was used to describe VMs that should be dynamically created in the Hyper-V host. Given the dynamic nature of these VMs, and the client's preference for non-cloud solutions, it would be less than ideal to have to hardcode VM names in advance, in order to create Vault access credentials.

In this section, I'll describe a proof of concept (PoC) using Vault's token authentication backend to address the above scenario. Three participants will work in this PoC:

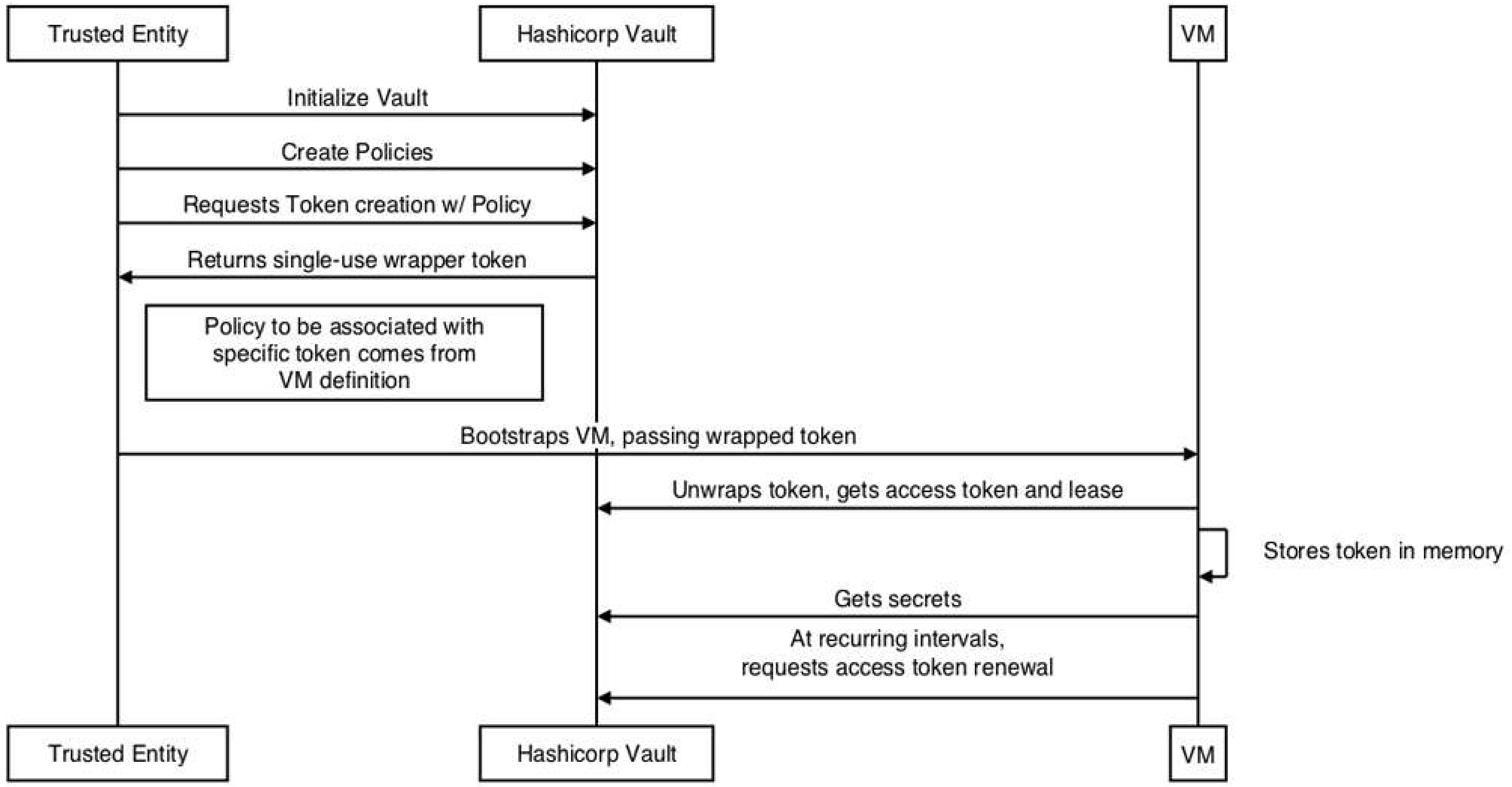
- A VM where Vault is installed

- A VM representing the client that wants access to a secret

- My workstation, which will be acting as a Trusted Entity. The Trusted Entity will be responsible for interacting with most of the Vault API, and passing the secrets to the end consumer. In the real world, this role can be played by Jenkins, Terraform, or another CI/Orchestration tool.

One critical problem that this PoC will address is how to pass the access token from the Trusted Entity to the client VM, while lowering the risk that it could be intercepted along the way. This will be accomplished by using Cubbyhole's response wrapping—a feature that allows a Vault response to be associated with a single use, time-limited token, stored within Vault. The Trusted Entity will only receive this token, and when the client VM uses it to retrieve the access token, it will expire—ensuring that the access token was only exposed once.

Here's a sequence diagram of what we plan to accomplish:


Trusted Entity
Hashicorp Vault
VM
Initialize Vault
Create Policies
Requests Token creation w/ Policy
Returns single-use wrapper token
Policy to be associated with specific token comes from VM definition
Bootstraps VM, passing wrapped token
Unwraps token, gets access token and lease
Stores token in memory
Gets secrets
At recurring intervals, requests access token renewal
Trusted Entity
Hashicorp Vault
VM

The code for this PoC can be found here.

Installation

To run this PoC, you'll need to have:

1. Vagrant, a Virtual Machine manager—instructions for different platforms here.
2. Git, a version-control system. Instructions here.

Starting VMs

Once both are installed, open the cmd prompt and execute:

```
git clone https://github.com/stenio123/hashicorp-vault-token-auth-poc.git
cd hashicorp-vault-token-auth-poc
vagrant up
```

console_1.sh hosted with ♡ by **GitHub** **view raw**

This will start the Hashicorp Vault VM and the Client VM that needs access to the secrets.

Initializing Vault

Before you can execute instructions against Vault, like creating secrets and access tokens, you need to initialize it. Run the following in the command prompt to connect to Vault using Vagrant:

```
vagrant ssh vault
```

console_2.sh hosted with ♡ by **GitHub** **view raw**

Once inside Vault, execute the following to initialize Vault with a given configuration file:

```
# This runs Vault in the background. Output is sent to nohup.out
nohup ./vault server -config=sync/HashicorpVault/config.hcl &
# Press <enter> to see the Exit 1 status code

# Using the address defined in the above configuration file
export VAULT_ADDR=http://0.0.0.0:8200

# Initializes unseal tokens and root access token
./vault init

# If you want to avoid having to type "./" before executing Vault, you can add this folder to your PATH by executing:
export PATH=$PATH:`pwd`
```

console_3.sh hosted with ♡ by **GitHub** **view raw**

This will output five unseal tokens, which need to be used to “unseal” or “seal” the Vault. Secrets can only be accessed when Vault is “unsealed,” and the configuration used for initialization defaults to minimum of three tokens to allow unseal.

In a real world scenario, these tokens would be distributed among key stakeholders, guaranteeing distributed responsibility for secrets management. They should be stored safely in the same way as personal passwords.

There will also be an additional token in the output –this is the root access token, with full permission to Vault. It should only be used for initial configuration, while recurring operations should be done by policy-constrained tokens.

With the tokens available, execute this command three times, entering one of the provided unseal tokens:

```
vault unseal
```

console_4.sh hosted with ♡ by **GitHub** **view raw**

Before performing any operations on an unsealed Vault, you need to first login. This can be accomplished by executing:

```
vault auth [root token]

# Should return output:
Successfully authenticated! You are now logged in.
token: [root token]
token_duration: 0
token_policies: [root]
```

console_5.sh hosted with ♡ by **GitHub** **view raw**

As a best practice, we will enable auditing log:

```
vault audit-enable file path=./vault_audit.log
```

console_6.sh hosted with ♡ by **GitHub** **view raw**

You can also check the default secrets folders by executing:

```
vault mounts
```

console_7.sh hosted with ♡ by **GitHub** view raw

This will return /cubbyhole, /secrets and /sys, with descriptions of their intended use. For this PoC, we're going to use all three of them.

Adding policies

The git repository includes three access policies, with the idea of mapping an enterprise's environment secrets to different folders. First we create the policies by executing:

```
vault policy-write production sync/HashicorpVault/policies/production.hcl
vault policy-write qa sync/HashicorpVault/policies/qa.hcl
vault policy-write development sync/HashicorpVault/policies/development.hcl
```

console_8.sh hosted with ♡ by **GitHub** view raw

Writing secrets

Now that we have the policies in place, let's add some sample secrets to each of the mapped environments:

```
vault write secret/production/password value=MyProdPassword
vault write secret/production/qa value=MyQAPassword
vault write secret/production/development value=MyDevelopmentPassword
```

console_9.sh hosted with ♡ by **GitHub** view raw

Create an access token associated with a policy

Here's where we'll create the token that the Node VM will use to communicate with Hashicorp Vault. Since this will be created by the Trusted Entity (i.e. Jenkins), we'll use the wrapper pattern to prevent anyone other than the Node VM itself and Vault from knowing the access token. By passing the -wrap-ttl parameter during token creation, Vault returns a Cubbyhole wrapped token, with a specified, single use time-to-live. The idea is: once it's used, it returns the access id token and lease, and can't be used again.

Therefore:

```
vault token-create -policy=production -wrap-ttl=20m

# Will output response (different token value):
Key                             Value
---                             -----
wrapping_token:                 d88d7612-14af-e58d-035f-f9446991bca4
wrapping_token_ttl:             20m0s
wrapping_token_creation_time:   2016-09-02 11:33:08.021734288 -0400 EDT
wrapped_accessor:               1cd465e7-67bd-7d75-385f-18afb47c489a
```

console_10.sh hosted with ♡ by **GitHub** **view raw**

This creates an access token and lease with "production" policy applied to it, and stores this information on the Cubbyhole.

Optional parameters

If additional control is desired, a few other parameters can be used with token-create:

- explicit_max_ttl: Sets the time after which this token can never be renewed.

- num_uses: Number of times this token can be used. Default is unlimited.
- renewable: If token is renewable or not. Default is true.
- ttl: Time token is valid. Default is 720hs.

Send single-use token to VM

You can take different approaches to sending the single-use token to the Client VM. In our PoC, we'll do a simple remote copy by copying the "wrapping token" value from the above output, opening another terminal window and issuing:

```
# Ensures that the ssh key has correct permissions
sudo chmod 400 ClientVM/ssh/unsafe_id_rsa
echo [single access token] | ssh -i ClientVM/ssh/unsafe_id_rsa -oStrictHostKeyChecking=no vagrant@192.168.0.51 "tee temporary-token.txt
```

Client VM gets access token

Now that the Node VM has the single access token. Open a third console window and execute:

```
vagrant ssh client

export VAULT_TOKEN=`cat temporary-token.txt`

RESPONSE=$(curl -H "X-Vault-Token: $VAULT_TOKEN" -X GET http://192.168.0.50:8200/v1/cubbyhole/response)
```

If returns permission is denied, token is either expired or has been compromised. Trusted Entity should be notified to contact Vault to revoke that token and create a new one.

If successful, it will return the output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   569  100   569    0     0   223k      0 --:--:-- --:--:-- --:--:--  555k
```

output_1.sh hosted with ♡ by GitHub view raw

In the background, this has assigned the following json to the 'RESPONSE' variable:

```
{
   "request_id":"940e5211-76f1-b33f-7130-af48e49c5a7c",
   "lease_id":"",
   "renewable":false,
   "lease_duration":0,
   "data":{
      "response":"{\"request_id\":\"f8b9ea99-2001-54db-3260-70b0c836aa7b\",\"lease_id\":\"\",\"renewable\":false,\"lease_duration\":0,\
   },
   "wrap_info":null,
   "warnings":null,
   "auth":null
}
```

response_1.json hosted with ♡ by GitHub view raw

To parse this json response, you can use regex or the jq tool command line json processor.

Using jq:

```
echo $RESPONSE | jq -r '.data.response | fromjson.auth.client_token' | ACCESS_TOKEN=-
# This sequence of commands uses shell pipes, where the output from the left is the input of the right:
# echo $RESPONSE
#   Outputs the wrapped token response
# jq -r '.data.response | fromjson.auth.client_token'
#   Uses jq to parse the json until response, and then parse again until client_token
# ACCESS_TOKEN=-
#   Inserts the output of the previous command in the environment variable ACCESS_TOKEN
```

console_15.sh hosted with ♡ by **GitHub** **view raw**

This will store the access token in the ACCESS_TOKEN variable in memory.

Client VM communicating with Vault

Now anytime Client wants to communicate with Vault, it should use the value in the $ACCESS_TOKEN environment variable, for example:

```
curl -H "X-Vault-Token: $ACCESS_TOKEN" -X GET http://192.168.0.50:8200/v1/secret/production/password
```

console_13.sh hosted with ♡ by **GitHub** **view raw**

This will return the json.

```
{
   "request_id":"8196ade4-9589-8bbd-685c-ac29e44c349c",
   "lease_id":"",
   "renewable":false,
   "lease_duration":2592000,
   "data":{
      "value":"MyProdPassword"
   },
   "wrap_info":null,
   "warnings":null,
   "auth":null
}
```

response_2.json hosted with ♡ by **GitHub** **view raw**

This token has ttl and will expire in 720hs. In order to keep alive, at regular intervals the Client VM must issue.

```
curl -H "X-Vault-Token: $ACCESS_TOKEN" -X POST http://192.168.0.50:8200/v1/auth/token/renew-self
```

console_14.sh hosted with ♡ by **GitHub** **view raw**

Lease renewal is only possible if token still valid. If expired or revoked, notify Trusted Entity and go back to the instructions described in the “Create an access token associated with a policy” section.