Seminar Report

Encryption Tools: Hashicorp Vault

Sonal Lakhotia

MatrNr: 14913835

Supervisor: Hendrik Nolte

Georg-August-Universität Göttingen
Institute of Computer Science

March 26, 2023

Abstract

Hashicorp Vault is an identity-based secrets and encryption management, cloud agnostic system. A secret is anything that needs strict control access. It includes Application Programming Interface (API) encryption keys, credentials, passwords, and certificates. Vault provides encryption services that incorporate authentication and authorization methods. Modern systems require access to multiple secrets, such as database credentials, API keys for external services, and service-oriented architecture communication credentials. Vault ensures access to secrets and other sensitive data is managed and stored with security. It allows restricted access that is auditable. Vault validates and authorizes clients such as users, machines, and apps before giving them access to secrets and storing sensitive data. Vault configuration is possible using Vault User Interface (UI), Command Line Interface (CLI), or Hypertext Transfer Protocol (HTTP) API. The report demonstrates Vault configuration for production with Transport Layer Security (TLS) configuration. It also illustrates specific use cases for Vault, such as limited Time-to-live (TTL) service tokens, one-time use tokens for access to secrets, and storing encrypted secrets in Random-Access Memory (RAM).

Contents

List of Figures

List of Listings

List of Abbreviations

UI	User Interface
TLS	Transport Layer Security
CLI	Command Line Interface
TTL	Time-to-live
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
RAM	Random-Access Memory
LDAP	Lightweight Directory Access Protocol
GPG	GNU Privacy Guard
HA	High Availability
PGP	Pretty Good Privacy
tmpfs	Temporary File System
HCP	Hashicorp Cloud Platform
AWS	Amazon Web Services

1 Introduction

Secrets such as credentials, TLS, host, client certificates, and API tokens hold specific access rights. They are responsible for restricting unauthorized access to applications or machines. In some cases, these credentials sprawl across the organization in plain text, application source code, config files, and other locations in enterprises. Although most application developers ensure that these details are encrypted and stored in a database, data might be hacked by malicious means. Bots used in websites or applications might scrape user credentials or Amazon Web Services (AWS) tokens and use it to their advantage. It is difficult and daunting to determine who has access to what and prevent malicious attacks by internal and external attackers. Vault resolves these challenges. It takes all the sensitive information and centralizes them so that they are defined in one location and prevents unwanted exposure to secrets and sensitive data. It ensures that authenticated users, applications, and systems access the resources . Vault provides an audit trail that tracks and maintains the client's actions . Vault has many pluggable elements, such as secret engines and authentication methods. They allow integration with external systems. These components to manage and protect secrets in dynamic infrastructures. Figure 1 demonstrates vault capabilities. It caters to identity-based access, policy-based authorization, and sensitive data retrieval based on client requests.

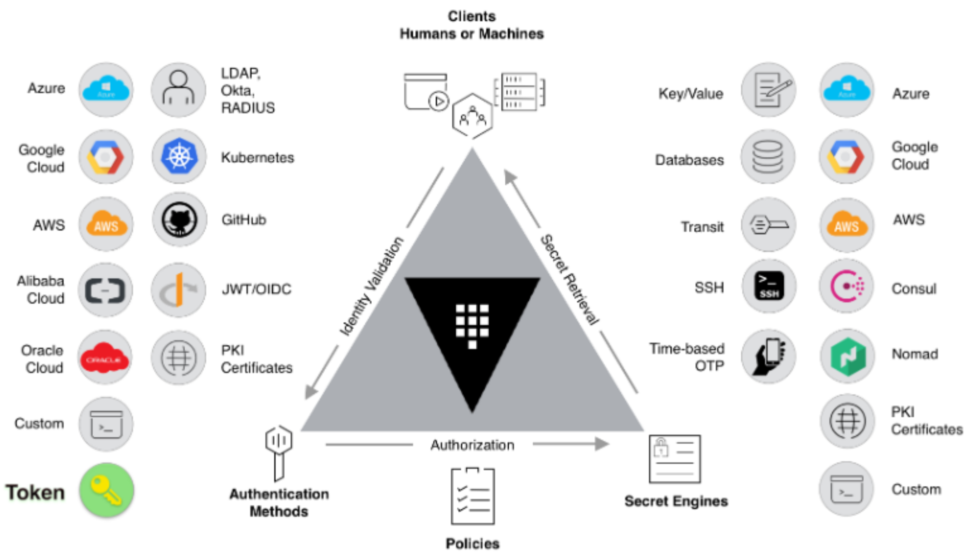

Figure 1: Vault Secret Management System [Has20]

As demonstrated in Figure 1, Vault validates and authorizes clients before providing them access to secrets or sensitive data. Vault predominantly works with tokens. A token is associated with the client's policy. A policy rule is path-based and constraints client's accessibility. Using Vault, clients can be assigned tokens manually, or the client can log in and obtain them. The core Vault workflow consists of four stages as demonstrated in Figure 2.

1. **Authenticate**: Vault authentication involves a process in which a client delivers information to Vault, and it validates and determines the client's identity. When an

auth method authenticates the client, a token is generated, and associated with a policy

2. **Validation**: Vault validates the client against third-party trusted sources, such as GitHub, Lightweight Directory Access Protocol (LDAP), AppRole, or other sources.

3. **Authorize**: The Vault security policy determines if a client matches its accessibility requirements. This policy is a set of rules that defines which API endpoints a client can access with its Vault token. Policies provide a declarative way to grant or prohibit access to specific paths and operations in Vault.

4. **Access**: Vault issues a token based on policies that determine the client's identity. It grants access to keys, secrets, and encryption capabilities. The client uses their Vault token for accessing resources.

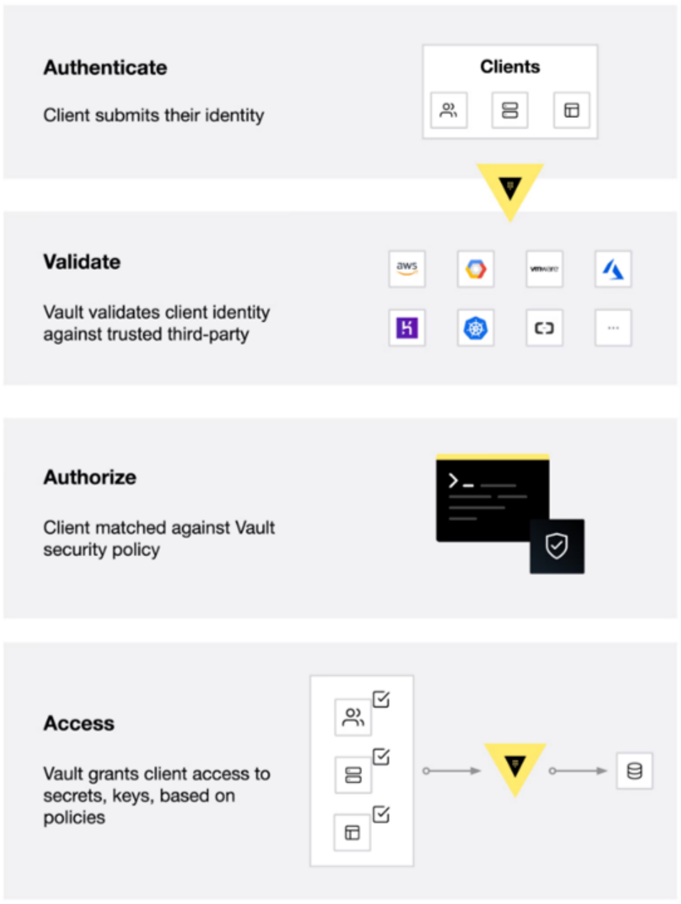
Authenticate
Client submits their identity
Clients
Validate
Vault validates client identity against trusted third-party
aws
Authorize
Client matched against Vault security policy
Access
Vault grants client access to secrets, keys, based on policies

Figure 2: Vault Core Workflow [Has20]

1.1 Vault Key Features

This section enlists Vault's key features that enable strict and restricted access to sensitive data and credentials.

1. **Secure Secret Storage**: Vault stores arbitrary key/value secrets. It encrypts these secrets before writing them to the persistent storage. Accessing raw storage alone

would not grant access to sensitive data. Vault can store these secrets in the disk, Consul, and more.

2. **Dynamic Secrets**: Vault generates on-demand secrets for some systems, such as AWS or SQL databases. If the lease is up, automatic revocation of these dynamic secrets is possible.

3. **Data Encryption**: Vault encrypts and decrypts data without storing it. Using Vault, security teams define encryption parameters, and developers store encrypted data in a location, such as a SQL database, without designing their encryption methods.

4. **Leasing and Renewal**: All secrets in Vault have an associated lease. When the lease ends, Vault automatically revokes that secret. Clients can renew leases via built-in APIs for review.

5. **Revocation**: Vault supports secret revocation. It can revoke a tree of secrets. Secrets read by a specific user or all credentials of a particular type form a tree of secrets. In case of an intrusion, revocation assists in key rolling and locking down systems.

2 Vault Configuration

Vault provides built-in and pre-configured dev servers for exploring and experimenting with it. Vault servers are explicitly configured for production installations. This section demonstrates Vault installation, Vault server configuration for production mode, and Vault seal/unseal processes.

2.1 Installing Vault

Hashicorp officially maintains and signs packages for several OS distributions. Listing 1 demonstrates Vault installation in Debian/Ubuntu. It also installs the HashiCorp GNU Privacy Guard (GPG) key and verifies the key's fingerprint.

```
$ sudo apt update && sudo apt install gpg
```

Listing 1: Installing Vault

2.2 Initiating Vault Server

Listing 2 demonstrates successful installation of Vault. Vault works as a client-server application. The Vault server starts in a *sealed* state. That is, it knows how to access the physical data storage but does not know to decrypt it. Data stored in Vault is encrypted using the encryption key in the keyring, and the encryption key is encrypted using the root key. Vault should be *unsealed* to access it. *Unsealing* Vault provides a root key necessary to read the decryption key and decrypt data. After Vault is unsealed, all operations are carried out via Vault CLI that interacts with the server over a TLS connection.

```
$ vault
Usage: vault <command> [args]

Common commands:
    read        Read data and retrieves secrets
    write       Write data, configuration, and secrets
    delete      Delete secrets and configuration
    list        List data or secrets
    login       Authenticate locally
    agent       Start a Vault agent
    server      Start a Vault server
    status      Print seal and HA status
```

Listing 2: Verifying vault installation

2.2.1 Vault Development Server

The Vault development server mode does not require any setup. Listing 3 shows the initialization of the dev server, initial root token creation, and the unseal key for re-authentication purposes. The development server stores all the data in memory and listens on localhost without TLS. TLS provides security, confidentiality, authenticity, and integrity through certificates. Client-Server applications use TLS to prevent Man-in-the-attack, eavesdropping, message forgery and tampering. It unseals automatically and optionally sets the initial root token. Vault attackers might be most interested in obtaining the initial root token to disrupt the system, and it is easily accessible in dev server mode. It is most suited for experimentation and exploration. Listing 3 explicitly states that the development mode is not for use in production work. Listing 4 shows that the Vault is unsealed and runs entirely in-memory.

```
$ vault server -dev

Unseal Key: GkhonqUuELi5sqJFsp9McJUhw77SVqPS+AYjV7F0zgI=
Root Token: hvs.pVpvQLcEi1JpKUUtH3qJKiXd

Development mode should NOT be used in production installations!
```

Listing 3: Starting vault dev server

2.2.2 Vault Configuration for Production

Vault development servers are not suitable for production installations. The dev server automatically unseals the vault and does not use TLS. It is not a security operation in the real environment. Vault is configured in production mode using config files. Vault should always use TLS to provide secure communication between the client and server. Each host should have a certificate file and a key file. Listing 5 demonstrates a configuration file (config.hcl) for starting the vault server in production mode.

```
$ export VAULT_ADDR='http://127.0.0.1:8200'
$ export VAULT_TOKEN='hvs.7eT6GRceYELd9qOraUAMENDw'
$ vault status

Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.12.2
Build Date      2022-11-23T12:53:46Z
Storage Type    inmem
Cluster Name    vault-cluster-756c87e0
Cluster ID      483e63b7-8ec0-7f66-69de-3407f0d1cf64
HA Enabled      false
```

Listing 4: Verifying vault status

The primary configurations in the config file are explained further.

- **ui**: Enables the Web user interface to be accessible from any machine on the subnet if no firewalls are in place at "https://127.0.0.1:8200/ui".
- **storage**: Integrated Storage (raft) is a production-ready backend and is better than in-memory storage used by dev servers. This is the physical backend that Vault uses for storage.
- **listener**: Vault uses TLS in production to provide secure communication between clients and the Vault server. It requires a certificate file and key file on each Vault host. The certificate and key were generated as -

  "openssl req -out tls.crt -new -keyout tls.key -newkey rsa:4096 -nodes -sha256 -x509 -subj "/O=HashiCorp/CN=Vault" -addext "subjectAltName = IP:127.0.0.1,DNS:c100-199.cloud.gwdg.de" -days 3650"
- **cluster_addr**: It indicates the address and port to be used for communication between the Vault nodes in a cluster.
- **api_addr**: It specifies the address to be advertised to route client requests.

2.3 Initializing the Vault

Vault initialization happens once when the server is started against a new backend that has not been used with Vault before. It is a process of configuring Vault. When running in High Availability (HA) mode, this happens once per cluster, not per server. This section demonstrates the initial root token creation, encryption keys generation, and unseal keys creation during initialization. The HA mode is automatically enabled when a specific data

```
ui = true
disable_mlock = true

storage "raft" {
  path    = "/opt/vault/data"
  node_id = "node1"
}

listener "tcp" {
  address = "127.0.0.1:8200"
  tls_cert_file = "/opt/vault/tls/certs/tls.crt"
  tls_key_file = "/opt/vault/tls/certs/tls.key"
}

cluster_addr = "https://127.0.0.1:8201"
api_addr = "https://127.0.0.1:8200"
```

Listing 5: Vault Configuration File

store that supports the mode is used. Starting the Vault server as per the configuration file shows that Vault is running with TLS enabled.

- Starting Vault server: vault server -config /home/cloud/config.hcl
- Vault server output should have TLS: enabled:

  Listener 1: tcp (addr: "127.0.0.1:8200",cluster address: "127.0.0.1:8201", max_request_duration: "1m30s", max_request_size: "33554432", tls: "enabled")
- Storage: raft (HA available)
- A new terminal session is launched and environment variables are set as:
  export VAULT_ADDR='https://127.0.0.1:8200'
  export VAULT_CACERT='/opt/vault/tls/certs/tls.crt'
- To initialize Vault: vault operator init
  This is an unauthenticated request that works for new Vaults without any data.
- Vault initialization provides information of incredibly high importance as shown in Listing 6. The unseal keys and the initial root token. This is the only time that all unseal keys are close together. In a real environment, these unseal keys can be provided to five different users and encrypted with their Pretty Good Privacy (PGP) keys so that vault security is integrated and no breach is possible.
- The vault server is initialized in a sealed state. Vault can access the physical storage but cannot read anything from it as it does not know how to decode it. Unsealing Vault is the process of teaching it to decrypt the data. Vault uses an algorithm known as Shamir's Secret Sharing to split the root key into shards. Only with the threshold number of keys it is reconstructed and the data is finally accessed.

```
$ vault operator init

Unseal Key 1: xnsfjd5NVli+kOVLASbvSFIMwJNAxRvtifnJZhUhyll2
Unseal Key 2: eCdlrwBjKHLZ3k1pPW9q3grI3ToVfpOGOU5/wlcoVJcT
Unseal Key 3: +hPyp6oTg1xY7we03h74ztM4j74x9lMM+aLobgPezWrc
Unseal Key 4: btqKkOM/hPznu2NR1cDUXa+x+CX3hr1Ol4ucnk50sioW
Unseal Key 5: 4f0TMnHBGt+c5w/slBANV84K4CDCniM5aURf0xm0Ka0r

Initial Root Token: hvs.g1TtWg13Q8CpWF37jDTbINMN

$ vault status

Key                 Value
---                 -----
Seal Type           shamir
Initialized         true
Sealed              true
Total Shares        5
Threshold           3
Unseal Progress     0/3
```

Listing 6: Vault status and unseal key generation

- To unseal the Vault: vault operator unseal After entering one of the unseal keys the vault status can be seen as shown in Listing 7

- Three different unseal keys are required to unseal Vault. If the unseal keys are correct, output, as shown in Listing 8, is obtained. The vault is unsealed and the initial root token could be used for authentication.

- Listing 9 shows a root user is logged in. As a root user, the Vault can be sealed again in case of an emergency or suspicion by a single operator.

3 Vault Use-Cases

Vault is an identity-based, sensitive data management system that finds use in many scenarios, such as secret storage, key management, and data encryption. It primarily works with tokens. Each token is attached to a path-based client policy. The policy rules restrict the actions and accessibility of the clients to specific paths. Tokens can be created manually and assigned to clients, or the clients can log in and obtain them. In this section specific scenarios, such as using tokens, type of tokens, token TTL, usage limit, and secret storage in RAM are discussed.

```
$ vault operator unseal

Unseal Key (will be hidden): xnsfjd5NVli+kOVLASbvSFIMwJNAxRvtifnJZhUhyll2
Key                  Value
---                  -----
Seal Type            shamir
Initialized          true
Sealed               true
Total Shares         5
Threshold            3
Unseal Progress      1/3
```

Listing 7: Unseal progress to unseal the Vault

```
$ vault status

Key               Value
---               -----
Seal Type         shamir
Initialized       true
Sealed            false
Total Shares      5
Threshold         3
```

Listing 8: Unsealed Vault

3.1 Single-Use Tokens with Limited TTL with Cubbyhole

Accessibility rights are set as per the Vault policy attached to tokens for secret management. Tokens enable users to access sensitive information. Trusted entities such as Chef, Jenkins, etc read secrets from Vault after obtaining a token. There might be a case that these trusted entities or their host machine are rebooted and it must re-authenticate with Vault securely with an initial token. In such a situation, cubbyhole response wrapping is very useful.

- In Vault, a cubbyhole is like a client's locker. All secrets are namespaced under the client's token. If the token is revoked or if it expires, all the secrets in the cubbyhole are revoked as well.
- It is not possible to reach into another person's cubbyhole even as a root user.
- Cubbyhole minimizes the risk of unauthorized access as opposed to key/value secret engine as secrets in key/value secret engine are accessible to any token as long as the policy allows it.
- Vault creates a temporary single-use token (wrapping token) and inserts the response into the token's cubbyhole with a short TTL when response wrapping is requested.

```
$ vault login

Token (will be hidden): hvs.g1TtWg13Q8CpWF37jDTbINMN
Key                     Value
---                     -----
token                   hvs.g1TtWg13Q8CpWF37jDTbINMN
token_accessor          JJtBxGV7l2bGvuRAkSi3wsDV
token_duration
token_renewable         false
token_policies          ["root"]
identity_policies       []
policies                ["root"]
```

Listing 9: Logging in as the root user

- Only the client expecting the response has the wrapping token to unwrap this secret.
- Cubbyhole response wrapping is beneficial as it provides a **cover**. The value being transmitted is not an actual secret but a reference to the secret.
- Any malicious activity can be easily traced as only a single client can unwrap the token and view the response.
- The lifetime of secret exposure is limited.

Single-Token and Limited TTL Usage Scenario: In this section, a scenario is assumed where an app needs to retrieve secrets from Vault at the cubbyhole/private path. However, the app does not have a valid client token to read secrets in cubbyhole/private. Vault admin wraps the secret values using the cubbyhole response wrapping and sends the wrapping token to the app. The app unwraps the secrets before the wrapping token expires.

As seen in Listing 5, Vault is configured for production mode. Using the initial root token is not recommended. Tokens with an appropriate set of policies based on their role in the organization should be used. Listing 10 shows the essential permissions included in the policy of an admin in the organization.

A token is created with limited TTL and limited usage as per the policy as shown in Listing 11. The use limit of the token can be limited to 1 if an admin decides so. In a new terminal session, the environment variable VAULT_TOKEN is set and A secret is created cubbyhole/cred path as shown in Listing 12. The secrets at cubbyhole/cred are read and the output is wrapped using the -wrap-ttl flag to specify that the response should be wrapped. **Life of the wrapping token is set to be 120 seconds (2 minutes)**. As seen in Listing 12 the secrets are not displayed at the cubbyhole/cred path but the response includes a wrapping token. An environment variable WRAPPING_TOKEN is set with the obtained token.

The Client/Entity app receives a wrapping token from the admin. To read secrets at the cubbyhole/cred path, the unwrap operation should be run using the wrapping token. The Client/Entity app does not require a client token to unwrap tokens using a valid wrapping token. The secrets stored at cubbyhole/cred are displayed as seen in Listing 13.

```
# Manage tokens
path "auth/token/*" {
  capabilities = [ "create", "read", "update", "delete", "sudo" ]
}

# Write ACL policies
path "sys/policies/acl/*" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}

# Allow a token to manage its own cubbyhole
path "cubbyhole/*" {
    capabilities = ["create", "read", "update", "delete", "list"]
}

# Allow a token to wrap arbitrary values in a response-wrapping token
path "sys/wrapping/wrap" {
    capabilities = ["update"]
}

# Allow a token to unwrap a response-wrapping token
path "sys/wrapping/unwrap" {
    capabilities = ["update"]
}
```

Listing 10: Essential permissions in Vault policy for the scenario

The wrapping token is limited to a **single-use** hence when the secrets are unwrapped again, it shows an error.

3.2 Secret Storage in RAM

File systems whose content resides in virtual memory can be created using the *Temporary File System (tmpfs)* facility. Data access is extremely fast as the files in this file system reside in RAM. All sensitive data from virtual memory would be lost hence it assures maximum security. Although, it is not encouraged to use in-memory storage for production as data does not persist beyond restarts. Listing 14 shows storage configurations for secret storage in RAM.

4 Future Work

Exploration and implementation of other vault use cases, such as identity-based access, data encryption, and generating dynamic secrets in the future, would pave way for further learning and enhanced secret management. Employing storage backends such as Consul and Hashicorp Cloud Platform (HCP) Vault would provide a consistent user experience.

```
$ vault token create -ttl=1h -use-limit=3 -field token  -policy=policy

hvs.CAESIApNUyEXTi3vgGZhB_QanCcvv................
```

Listing 11: Vault token creation as per the policy

```
$ export VAULT_TOKEN='hvs.CAESIApNUyEXTi3vgGZhB_QanCcvv....'
$ vault kv put cubbyhole/cred username="Chef123" password="Chef123"

Success! Data written to: cubbyhole/cred

$ vault kv get -wrap-ttl=120 cubbyhole/cred

Key                                 Value
---                                 -----
wrapping_token:                     hvs.CAESIBpP_nexpAigJ9xIHU9NAi7aFX......
wrapping_accessor:                  lWw9oesMjZJUJiSoASTnECmm
wrapping_token_ttl:                 2m
wrapping_token_creation_path:       cubbyhole/cred
```

Listing 12: Secret creation and Wrapped response creation

Monitoring and Troubleshooting to inspect the Vault environment and understanding audit and operational logs in depth would help with a better understanding of an error.

5 Conclusion

Vault's capabilities ensure the secure management of sensitive data and impose role-based access to resources. Vault simplifies accessibility using tokens and policies attached to these based on the user role. Exploring, learning about Vault seal/unseal, and configuring Vault with end-to-end TLS was the most challenging aspect of the practical experience with Vault, provided that most of the implementation in Vault references were carried out in dev server mode. The code listings in the report exhibit my practical experience with Hashicorp Vault. In this report, Vault authorization using a role-based policy is demonstrated. The initial root token is not secure when used for production mode therefore authentication is carried out using tokens generated as per the policy. Cubbyhole response wrapping with a single-use token with limited TTL is described. The practical executions also illustrate Vault storage to be in-memory or RAM. It is seen that all data is lost when the running system is rebooted. It might be fruitful for carrying out tasks of great importance without the risk of losing them to attackers. But it is not recommended to use memory storage for production tasks. Vault workflow and numerous use cases accentuated my learning about this incredible secret management tool.

```
$ VAULT_TOKEN=$WRAPPING_TOKEN vault unwrap

Key          Value
---          -----
password     Chef123
username     Chef123
```

Listing 13: Unwrapped secrets

```
storage "inmem" {}

storage "file" {
  path    = "/mnt/mytmpfs"
  node_id = "node2"
}
```

Listing 14: Storage in virtual memory configuration

References

[Has20] PA Hashicorp. *Hashicorp vault.* 2020.