HELECLOUD

Centralised Secrets Management with HashiCorp Vault on AWS White Paper

Zhulian Ginev

2021

CONTENTS

ABSTRACT

This whitepaper provides an overview of HashiCorp Vault and the best practices when using Vault to implement a centralised secrets management system on Amazon Web Services. It is intended for Solution Architects and DevOps Engineers who are responsible for building complex infrastructures.

INTRODUCTION

The purpose of this document is to present the recommended patterns for deploying centralised secrets management using HashiCorp Vault on AWS. These include several considerations that adhere to the AWS Well-Architected Framework and various other industry best practice guidelines for provisioning the infrastructure for a Vault system, as well as for properly operating it.

OVERVIEW

Centrally managing secrets is an essential part of a well-built production grade system. Understanding who is accessing what secrets is a challenging task, which rapidly becomes even more complex with the introduction of new applications, features, and services. HashiCorp Vault is an identity-based secrets and encryption management system that aims to address those issues by providing:

- Management of secrets and sensitive data protection
- A single source of secrets for both humans and machines
- Complete lifecycle management for secrets that:
  - Eliminates secret sprawl
  - Securely stores any secret
  - Provides governance for access to secrets

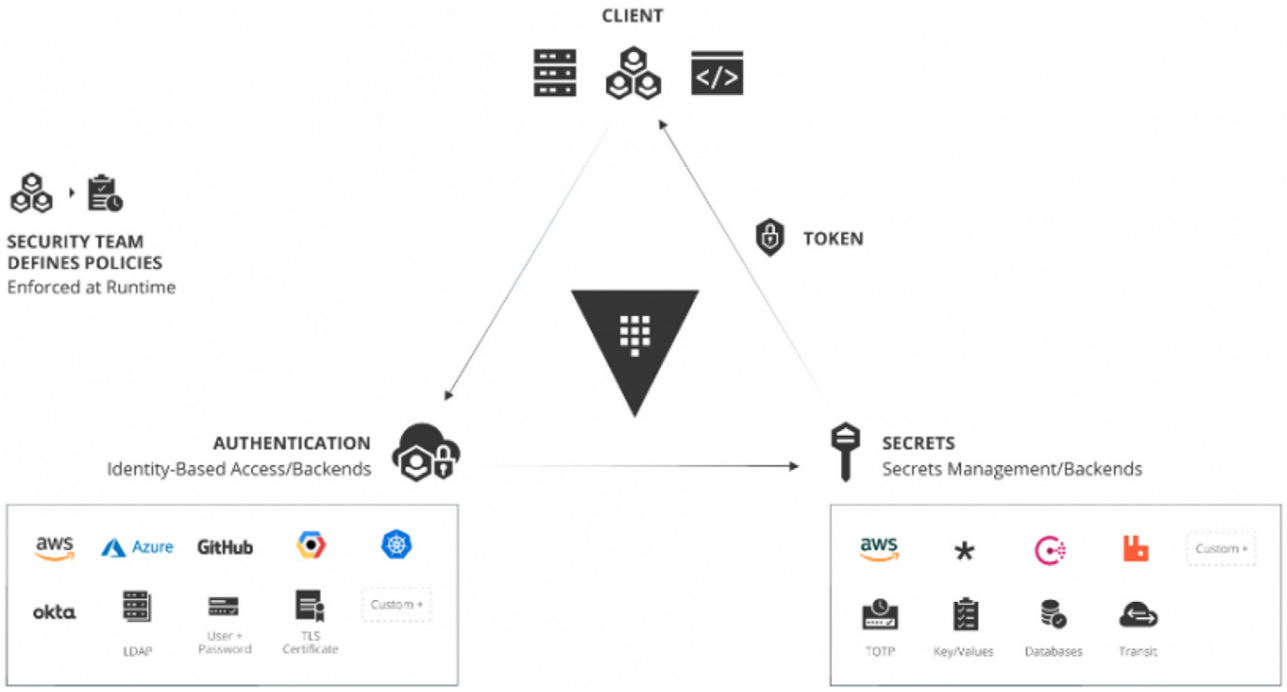
Any data that is deemed sensitive by your organisation has to be protected and though this data can be in different forms, have various purposes, and be found within multiple components of a modern system, the chances are that Vault will have a solution for it. Vault introduces the concept of Secrets Engines, which support all the common secret types, including but not limited to:

- Usernames and passwords
- API keys
- Encryption keys
- Certificates

HOW DOES VAULT WORK?

Vault can be used to store sensitive values and simultaneously dynamically generate access for specific services/applications on a lease. It can also be used to authenticate users (machines or humans) to make sure they are authorised to access a particular file.

Whenever a user or a machine needs to access a particular secret, they need to authenticate using a configured auth method. Depending on the auth method type, the authentication can happen via static or dynamic credentials. Once authenticated, policies attached to the user/machine are evaluated to determine whether they can access the secret or not. Finally, the secrets engine storing the secret issues a token, granting access to the requester. This flow is illustrated by the following diagram:

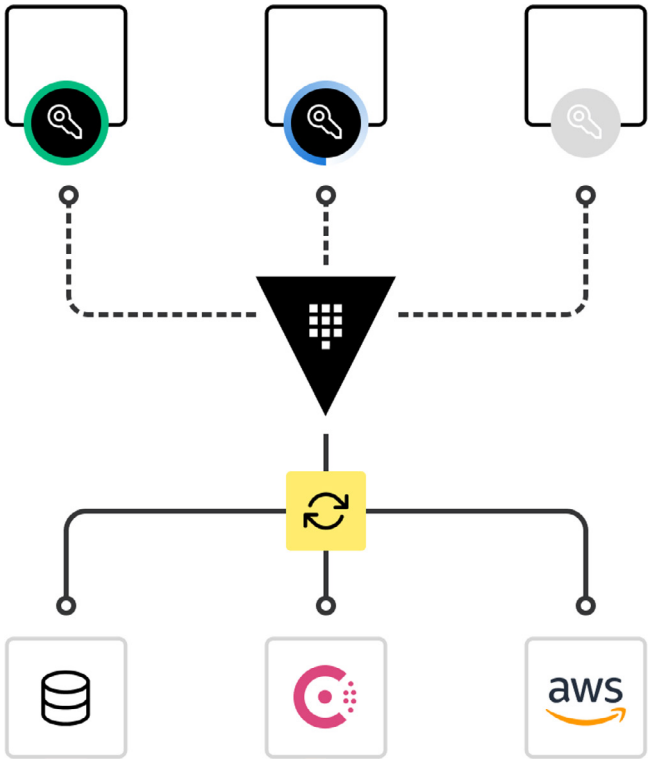


USE CASES FOR VAULT

To fully understand whether a Vault system is appropriate for your organisation, we will look at some of the key use cases that Vault addresses.

CENTRALISING STORAGE OF SECRETS

Modern systems require a centralised, dynamically scalable solution for storing secrets. Vault centrally manages and enforces access to secrets and systems based on trusted sources of application and user identity.



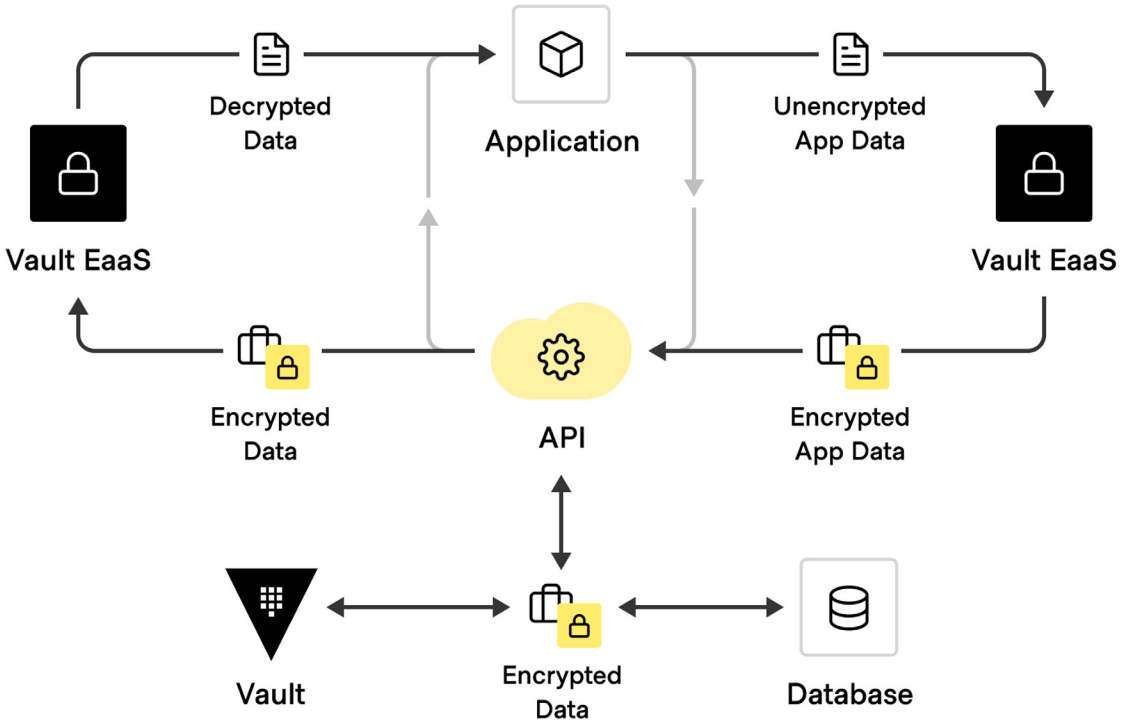
MIGRATING FROM STATIC TO DYNAMICALLY GENERATED SECRETS

The ability to create and enforce dynamic credentials allows us to adhere to least privilege principles without human interaction.

Static Credential	Dynamic Credential
▪ Validate 24/7/365	▪ Short-lived
▪ Long-lived	▪ Follows principle of least privilege
▪ Manual password rotation	▪ Automatically revoked (based on lease)
▪ Frequently shared across teams	▪ Each system can have unique credentials
▪ Reused across systems	▪ Programmatically retrieved
▪ Susceptible to being added to VCS	▪ No human interaction needed
▪ Often highly privileged	
▪ Manually created	

DATA ENCRYPTION

According to best practices, it is recommended that all application data should be encrypted in transit and at rest. Vault provides encryption as a service with centralised key management to simplify encrypting data.



AUTOMATED X.509 CERTIFICATES MANAGEMENT

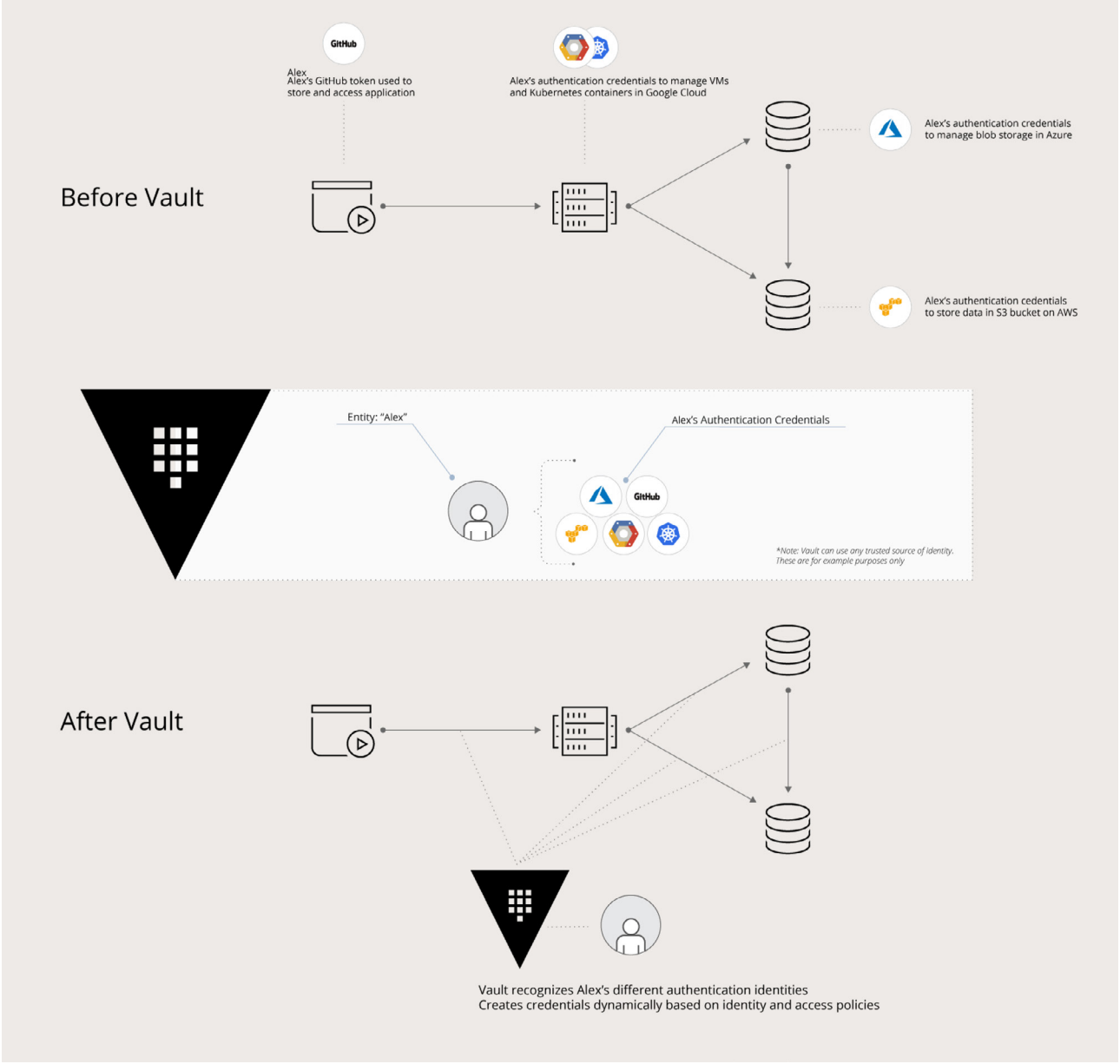
Generating, signing and retrieving certificates is usually a lengthy process that may have many manual steps associated with it. Your organisation might be using a different solution to automate it but that automation might still be overcomplicated and hard to manage. Vault solves this problem with its PKI secrets engine.

From a PKI engine perspective, Vault acts as a non-opinionated tool that focuses on automating PKI Operations and the process around fetching and renewing certificates in an automated fashion, Vault can act as a Root CA or as an intermediate (requires CSR signing from the root CA).

The process is described in detail **here**.

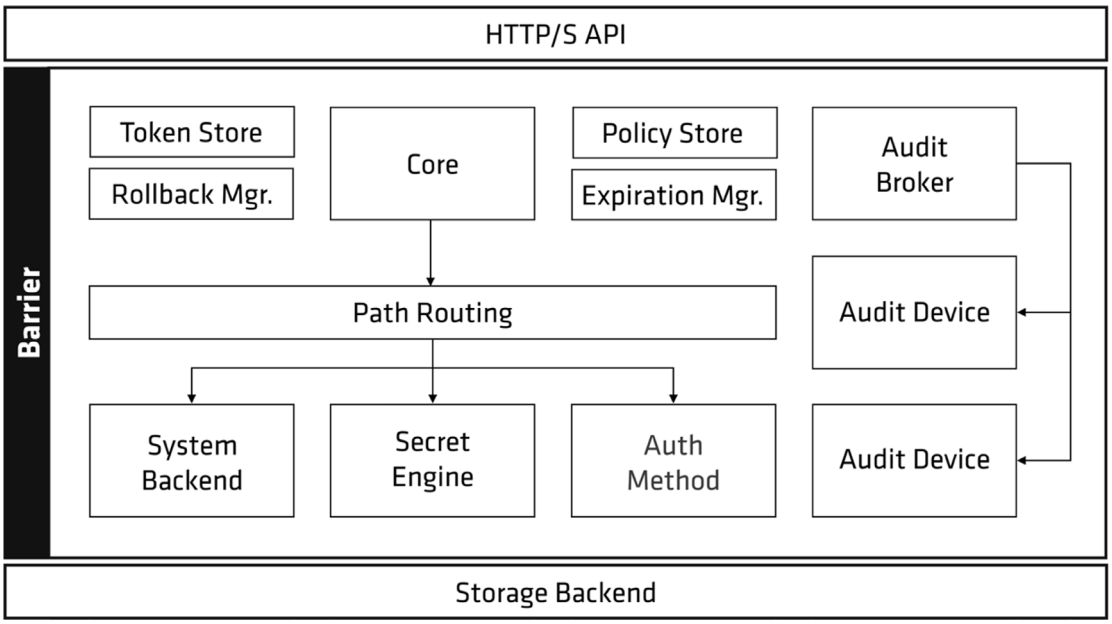
MIGRATING TO IDENTITY-BASED ACCESS

With Identity and Access Management (IAM) in Vault, users and organisations can map global access and policies to Vault Identity Entities and Groups. For example, let's say Alex is running an application in a Kubernetes container on Google Cloud that gets deployed from GitHub, it connects to a database in Azure, and replicates to AWS. Alex can integrate his different cloud identities into an "Alex" entity and centrally create policies granting and restricting access to secrets for the different components to connect securely to one another, all within one workflow and API. The diagram below represents how Vault plays a part in this.



ARCHITECTURE AND DESIGN

To help us visualise how Vault works under the hood, HashiCorp introduces the concept of "layers" with the following picture:



The main focus of this diagram is the separation of Vault components that are inside or outside the so-called "security barrier". The API and the storage backend are an essential part of Vault, without them a Vault system cannot function properly, but they are untrusted by default and need to be given the proper configuration, so they can be started by the Vault server on launch.

The Vault server launches in a "sealed state", which prohibits any API requests to the storage backend. This effectively means that Vault cannot work until it is unsealed and we will discuss the different unsealing methods later in this document. After the Vault is unsealed, requests can be processed from the HTTP API to the Core. The core is used to manage the flow of requests through the system, enforce ACLs, and ensure audit logging is performed.

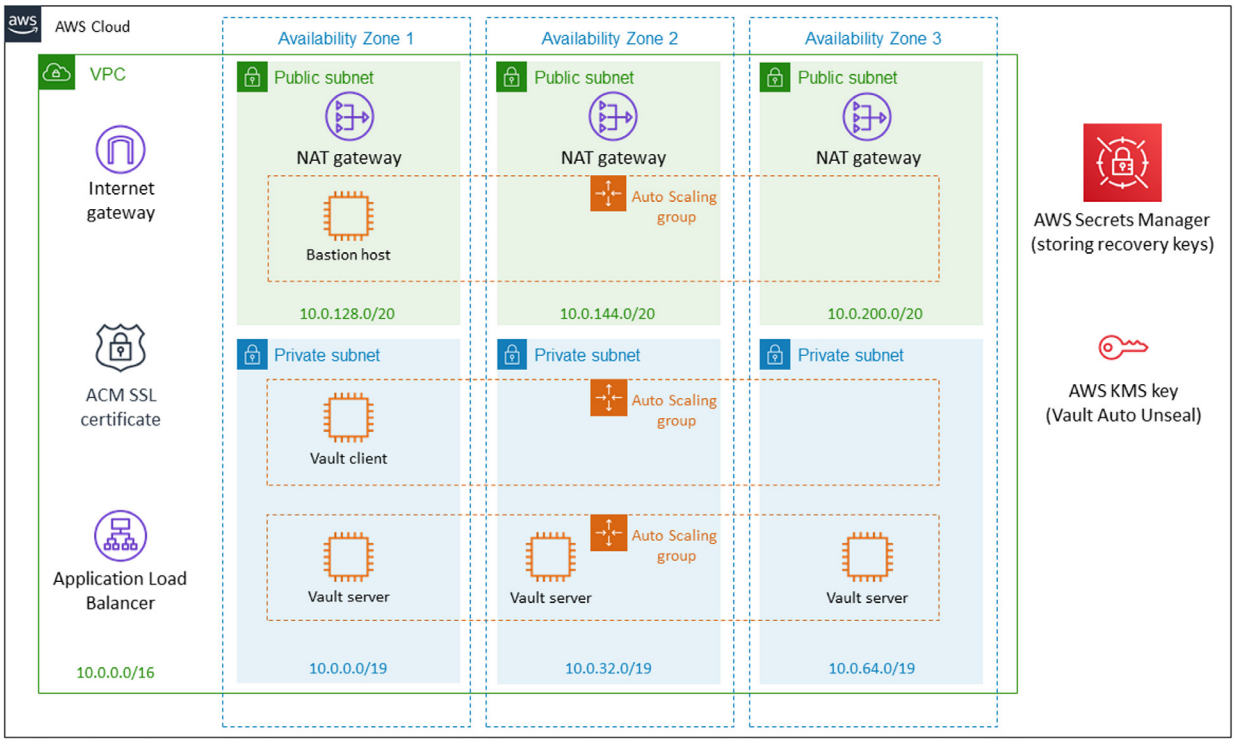
The configuration of audit devices, auth methods, and secret engines must be stored in Vault since they are security sensitive. This means that they cannot be configured outside of the barrier and are protected by Vault ACLs and tracked by audit logs.

Policies in Vault are just a named ACL rule. There is a "root" policy created by default, which permits access to all resources. You can define any number of policies with fine-grained control over paths. HashiCorp define Vault's operating mode as "allowed-access" mode, which is similar to AWS policies that work in an "implicit deny" mode. You can override the implicit deny with an explicit allow statement on specific actions.

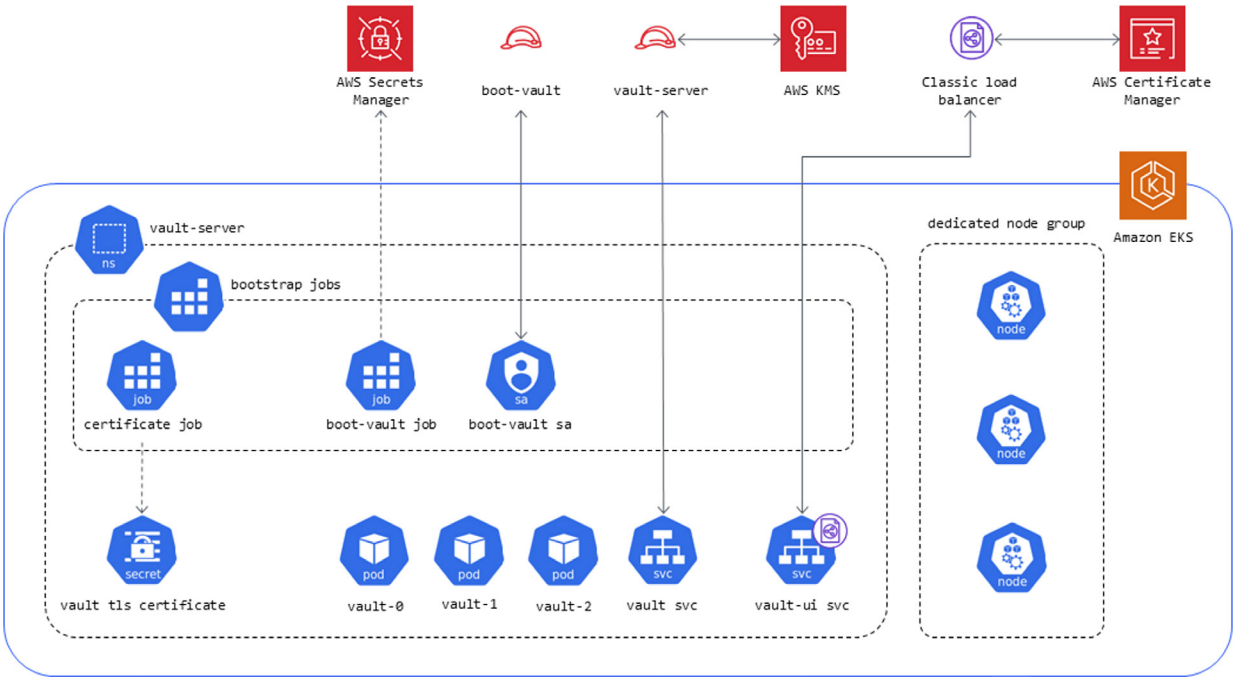
More detailed information on this topic can be found in HashiCorp's Vault Reference Architecture.

There are plenty of different strategies for deploying Vault on AWS, some examples are:

HashiCorp Vault on AWS EC2



HashiCorp Vault on AWS EKS



There are a couple of notable takeaways from these architectures, as well as variations that we can make:

- Several AWS native services are easily integrated with Vault, including:
  - AWS Certificate Manager – used to generate the Vault LB certificate for TLS
  - AWS Secrets Manager – used to store recovery keys for the Vault server
  - AWS KMS key – used to create a KMS key, which can then be integrated with the Vault server to allow auto unsealing using the same key
- The Vault servers can be deployed on VMs (EC2) or in a micro-services model as containers in an ECS cluster (official HashiCorp Vault docker image), or in a Kubernetes cluster (EKS) where we can use the official HashiCorp helm charts for Vault.

Below, we will look at some of the ways to automate the Vault deployment and integrate the appropriate bootstrapping process for the various types of architectures after providing some more context about the initial configuration considerations that you need to make.

CONFIGURATION AND INITIALISATION

The very first steps of configuring a Vault server are to modify the Vault configuration file and start the Vault initialisation process, which prepares the storage backend to receive data. This is only performed once on a single Vault node, even in a cluster setup. At this stage, the initial root token is generated and returned to the user who is performing the initialisation. You also have the option to define thresholds, the number of key shares, encryption and recovery keys.

The Vault servers are configured using a file written in either HCL or JSON format. The configuration file consists of several stanzas and parameters that define a variety of configuration options. The configuration file is specified when starting Vault using the **config** flag:

```
$ vault server –config <location>
```

The general practice is to store the configuration file in the /etc directory (e.g. /etc/vault.d/vault.json), but you can place it anywhere on the Vault server filesystem.

The configuration options that can be defined within the config file include:

- Storage Backend
- Server Listeners
- Cluster Listener
- TLS Certificates
- Seal/Unseal Config
- Cluster Metadata
- Log Level
- UI Config
- Telemetry (publishes metrics to upstream systems)

A basic configuration example may look like this:

```
storage "consul" {
  address = "127.0.0.1:8500"
  path    = "vault"
}

listener "tcp" {
  address     = "127.0.0.1:8200"
  tls_disable = 1
}

telemetry {
  statsite_address = "127.0.0.1:8125"
  disable_hostname = true
}
```

A full list of configuration parameters can be found within the Vault Configuration documentation.

Vault can be initialised via CLI, API or UI. If you want to initialise Vault through the CLI with default options, all you have to do is execute:

```
$ vault operator init
```

Non-default options can be passed as flags to this command. A list of flags and some initialisation examples with non-default options can be found within the operator init documentation.

STORAGE BACKENDS

Storage Backends are used for storing Vault data. If you are using Vault open-source, you can choose and configure a backend that is best suited for your business needs. Vault supports an exhaustive list of storage backends and the configuration from Vault perspective is as simple as defining a storage stanza in the Vault configuration file and defining the needed parameters:

```
storage [NAME] {
  [PARAMETERS...]
}
```

Consul example:

```
storage “consul” {
  address = “127.0.0.1:8500”
  path = “vault/”
  token = “1a2b3c4d-1234-abcd-1234-1a2b3c4d5e6a”
}
```

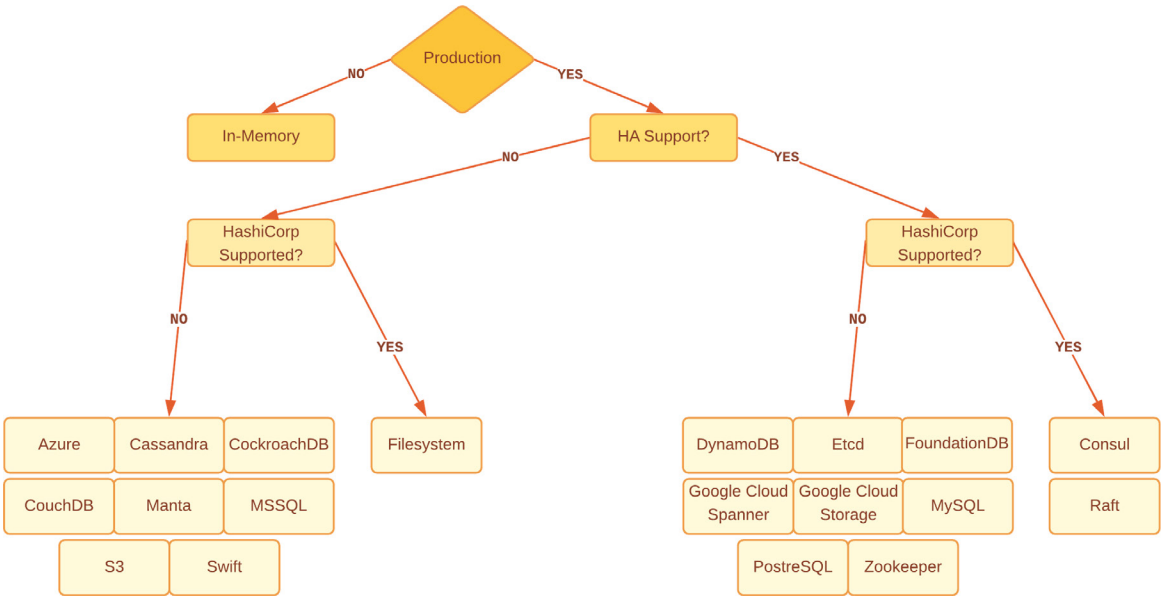
Raft example:

```
storage “raft” {
  path = “/opt/vault/data”
  node_id = “node-a-eu-west-1.example.com”
  retry_join {
     auto_join = “provider=aws region eu-west-1 tag_key=vault tag_value=eu-west-1”
  }
}
```

A full list of supported storage backends with example configurations and parameters can be found within the storage stanza documentation.

Note: Enterprise Vault Clusters should use either Consul or Integrated Storage (Raft) as storage backends. Realistically, you could use any of the other options as well, but it would not be a HashiCorp supported solution.

The following decision tree can help with the decision-making process when picking which storage backend to use:



AUDIT DEVICES

Keeping detailed logs of all authenticated requests and responses to Vault is an extremely important aspect of security, which is achieved by enabling audit devices. As a best practice, you should always have more than one audit device enabled. The audit logs are formatted in a JSON format and use **HMAC-SHA256** hashing to ensure sensitive data is not captured as plain text.

However, the values of those secrets can be exposed by accessing **/sts/audit-hash API** and comparing the log files, which can be done by any user with the privileges to do so. Therefore, the log files should be well-protected.

There are three types of audit devices that can be enabled:

- **File** – appends audit logs to a file, does not handle log rotation on its own and easily integrates with 3rd party data collector tools (Splunk, ELK Stack, New Relic, Datadog, Fluentd, etc.).
- **Syslog** – writes audit logs to a syslog, sends only to local agent.
- **Socket** – writes audit logs to a socket (supports TCP, UDP and Unix sockets), is unreliable due to the underlying protocol that it uses and should only be used where strong guarantees are required.

Enabling an audit device is once again as easy as executing a single command:

```
$ vault audit enable <type> file_path=<path>
```

Note: Vault needs to have the necessary permissions to write to the specified log file before completing a request. It prioritises safety over availability and if it cannot write to a persistent log, it will stop responding to **any client requests**, which is a system down situation.

CLUSTERING AND HIGH AVAILABILITY

Vault provides built-in high availability in the form of clustering. Multiple Vault nodes form a cluster when sharing a storage backend without additional configuration. This shared storage backend must support HA. HashiCorp recommends forming Vault clusters by deploying a minimum of 3 Vault servers, each in a different availability zone in order to achieve a potent HA strategy.

There are several storage backends that support HA mode, including Consul, ZooKeeper and etcd. However, HashiCorp recommend using Consul as a storage backend as it is the solution we are currently using in production environments.

Vault supports three types of node configurations in a cluster setup, including:

- **Active Node** – supports reads and writes in a basic Vault cluster setup.
- **Standby Node** – forwards all requests to the active node.
- **Performance Standby** – processes reads without forwarding to active node and forwards writes to the active node. This is a performance replication setup that requires an Enterprise licence. We will discuss it in more detail in later sections.

Vault is typically front-ended by either a load balancer or Consul. The cluster listener configuration that is specified within the Vault configuration file contains the configuration for a cluster LB, which in this case is used to ensure HA rather than load balance traffic between the nodes.

The load balancer determines which node is the active one by executing a simple health check on:

```
https://<ipaddr>:8200/v1/sys/health
```

Which responds with:

- **HTTP Response 200** = initialised, unsealed, active node
- **HTTP Response 429** = unsealed, standby node

The same endpoint can also be used to query the state of a Vault cluster:

```
$ curl -sS http://localhost:8200/v1/sys/health | jq
 {
  "initialized": true,
  "sealed": false,
  "standby": false,
  "performance_standby": false,
  "replication_performance_mode": "disabled",
  "replication_dr_mode": "disabled",
  "server_time_utc": 1545132958,
  "version": "1.0.0-beta1",
  "cluster_name": "vault-cluster-fc75786e",
  "cluster_id": "bb14f30a-6585-d225-ca12-0b2011de4b23"
}
```

Beyond the load balancer, a Vault cluster can be integrated with Consul and you can easily access the Vault nodes using the Consul DNS records:

- Active node: **active.vault.service.consul**
- Standby node: **standby.vault.service.consul**

For further reading on the topic, I would highly recommend looking at:

- Bryan Krausen's Webinar on Designing High Availability for HashiCorp Vault in AWS
- Halodoc's article on Vault High Availability and Scalability
- HashiCorp's documentation:
  - HA Mode
  - Vault HA Cluster with Integrated Storage
  - Vault High Availability with Consul

PORTS AND PROTOCOLS

As discussed previously, all communication for Vault uses TLS and the default communication ports are:

- TCP/8200 – UI and API
- TCP/8201 – server to server communication

When deploying Vault on AWS, you would need to consider the following requirements to ensure Vault communication is established at a network level:

Source	Target	Port	Protocol	Direction	Description
Vault Clients	Vault LB	8200	TCP	Ingress	Vault API
Vault LB	Vault Nodes	8200	TCP	Ingress	Vault API
Vault Nodes	Vault Nodes	8201	TCP	Bidirectional	Request Forwarding
Vault Nodes	Vault Nodes	8301	TCP/UDP	Bidirectional	LAN Gossip Communication
Vault Nodes	Consul Nodes	8301	TCP/UDP	Bidirectional	LAN Gossip Communication
Vault Nodes	Consul Nodes	8500	TCP	Ingress	Consul API
Consul Nodes	Consul Nodes	8300	TCP/UDP	Bidirectional	Server RPC
Consul Nodes	Consul Nodes	8301	TCP/UDP	Bidirectional	LAN Gossip Communication
Vault Primary Cluster	Secondary Cluster	8201	TCP	Bidirectional	Vault Replication

UNSEALING VAULT

Initially, Vault starts in a sealed state, meaning it knows where to access the data, and how, but cannot decrypt it. No operations besides status checks and unsealing are possible as long as Vault is in a sealed state.

When you unseal Vault, a node reconstructs the Vault master key in order to decrypt the encryption key and read the data. The encryption key is then stored in memory.

It is worth mentioning that there is an API endpoint that can be used to query the seal status of a particular Vault node:

```
$ curl -sS http://localhost:8200/v1/sys/seal-status | jq
 {
  "type": "shamir",
  "initialized": true,
  "sealed": false,
  "t": 1,
  "n": 1,
  "progress": 0,
  "nonce": "",
  "version": "1.0.0-beta1",
  "migration": false,
  "cluster_name": "vault-cluster-fc75786e",
  "cluster_id": "bb14f30a-6585-d225-ca12-0b2011de4b23",
  "recovery_seal": false
}
```

This raises the question: if Vault is unsealed, when and why would we seal it? Vault may need to be sealed if:

- Key shards are inadvertently exposed
- The network is compromised or a network intrusion is detected
- Spyware/malware is detected on the Vault nodes

There are three methods to unseal Vault:

- Key Shards (Manual) Unseal [Default method]
- Auto-Unseal, which includes:
  - AWS KMS Auto-Unseal
  - Azure Key Vault Auto-Unseal
  - GCP KMS Auto-Unseal
  - AliCloud KMS Auto-Unseal
  - HSM Auto-Unseal
- Transit Auto-Unseal

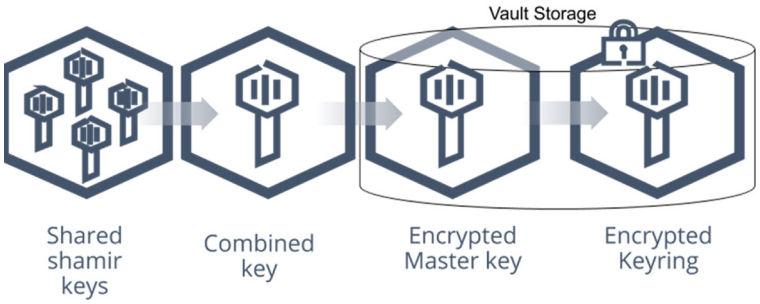
We will discuss Key Shards Unseal and AWS KMS Auto-Unseal for the purposes of this document. However, should you need more information about any of the other unseal methods, I would recommend taking a look at the official Vault documentation:

- Seal/Unseal
- Recommended Pattern for Vault Unseal
- Recommended Pattern for Stateless Vault for Transit Auto Unseal
- Halodoc's article on Vault Auto Unseal via AWS KMS

KEY SHARDS UNSEAL

The default Vault config uses a Shamir seal. Instead of distributing the unseal key as a single key to an operator, Vault uses an algorithm known as Shamir's Secret Sharing to split the key into shards. A predefined threshold of shards is required to reconstruct the unseal key, which is then used to decrypt the master key. You can override this threshold by passing the key-threshold flag, as part of the Vault initialisation ceremony.

Therefore, the unseal process is: the shards are added one at a time (in any order) until enough shards are present to reconstruct the key and decrypt the master key.



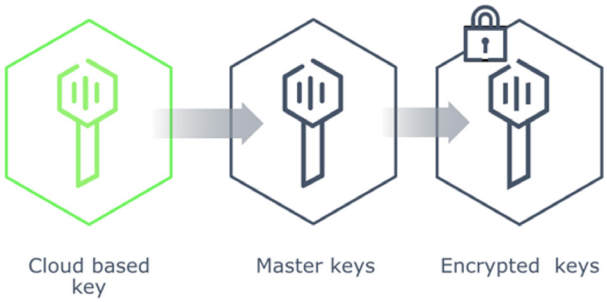
Note: The Encryption Key is never displayed, it is created when unsealing the Vault using shares of the Master Key. If Vault is sealed, the Encryption key is destroyed and a new Encryption key will be generated upon the next 'unseal' command.

Some of the takeaways for unsealing Vault with key shards are:

- It is the default unsealing option – no extra configuration is needed.
- No single person should have access to all key shards.
- Each key shard should be stored by a different employee.
- When initialising Vault, you can require individual key shards to be encrypted with different PGP keys.
- When unsealing Vault, you will need to provide a number of key shards, which is equal to the minimum threshold for unsealing.
- Key shards should not be stored online and should be well-protected.

AWS KMS AUTO-UNSEAL

When you are deploying a Vault system on AWS, it is a good idea to take advantage of the Auto Unseal mechanism, which utilises AWS KMS keys to make unsealing a much more convenient and reliable process.



The AWS KMS Auto-Unseal only requires a simple configuration stanza to be added to the Vault configuration file that identifies the particular KMS key to be used for decryption. Auto Unseal will automatically unseal Vault upon Vault service restart without additional intervention. The Auto Unseal configuration stanza looks like:

```
cat /etc/vault.d/vault.hcl


storage "file" {
  path = "/opt/vault"
}
listener "tcp" {
  address     = "0.0.0.0:8200"
  tls_disable = "true"
}
seal "awskms" {
  region = "us-east-1"
  kms_key_id = "d7c1ffd9-8cce-45e7-be4a-bb38dd205966"
}
ui=true
```

This snippet above is a direct reference from HashiCorp's webinar for Auto-Unseal, which contains a detailed demo for configuring AWS KMS Auto-Unseal, as well as for migrating from the default unseal mechanism to Auto Unseal.

DEPLOYMENT PATTERNS AND VAULT ONBOARDING

The preferred deployment pattern for the recommended architecture is to perform immutable builds of HashiCorp Vault and Consul. Since Vault is a HashiCorp product, it integrates easily with other products like Packer and Terraform, which can be used to build immutable images and provision the Vault infrastructure as code respectively. There are a number of examples that can be used to integrate Vault with existing or new CI/CD pipelines deployed in AWS.

Vault Deployment:

- AWS Vault Terraform Module
- Gruntwork's Vault/Consul Cluster Terraform Module
- Vault on AWS ECS Terraform Module
- Vault on AWS EKS using Terraform Guide
- Vault on AWS EKS Quick Start (CloudFormation)
- Vault on AWS EC2 Quick Start (CloudFormation)
- Vault Helm Chart for an AWS EKS Deployment

Vault Image Builds:

- Vault and Consul Packer templates
- Vault Role for Ansible Galaxy
- Consul Role for Ansible Galaxy
- Vault module for Puppet
- Vault cookbook for Chef

BACKUP STRATEGIES

As previously discussed, Vault stores its data on the storage backend that you configure. Therefore, the strategy for backing up the Vault data varies depending on your configuration.

Since HashiCorp recommends using either Consul or Integrated Storage (Raft) as storage backends, they support the backup procedures outlined in their Vault Data Backup Standard Procedure document.

BASIC POLICY CONFIGURATION

Depending on the various administrative roles that your organisation wants to incorporate with Vault you may want to create policies that define access to specific administrative actions into Vault (like adding mounts, policies, configuring further authentication, audit, etc...), cryptographic actions (like starting key rotation operations), and ultimately, consumption patterns, which are generally defined later based on requirements. I would recommend looking at HashiCorp's Brian Green gist on methods of writing Vault ACL policies to understand how to create them via API or CLI and how to define them in Terraform.

Additionally, some more great resources to start with are:

- HashiCorp's Policies document
- Recommended Pattern for Vault ACL Policy Path Templates
- HashiCorp's guide to writing policies
- Nicolas Corrarello's "A Vault Policy Masterclass" presentation

KEYS ROTATION

In the world of Vault, there are three keys that all play a major part in Vault's data encryption/decryption process: the **storage**, the **master** and the **seal** keys. A proper rotation strategy is needed to ensure that Vault and its data are secure.

STORAGE KEY

The Storage Key encrypts every secret that is stored in Vault, and only lives unencrypted in memory. This key can be rotated online by simply sending a call to the right API endpoint, or from the CLI:

```
$ vault operator rotate
Key Term        3
Install Time    20 Dec 21 11:47 UTC
```

This requires the right privileges as set in the policy. Once the key has started rotating, every new secret gets encrypted with the new key. This is a straightforward process that most organisations carry out every six months unless there is a compromise.

MASTER KEY

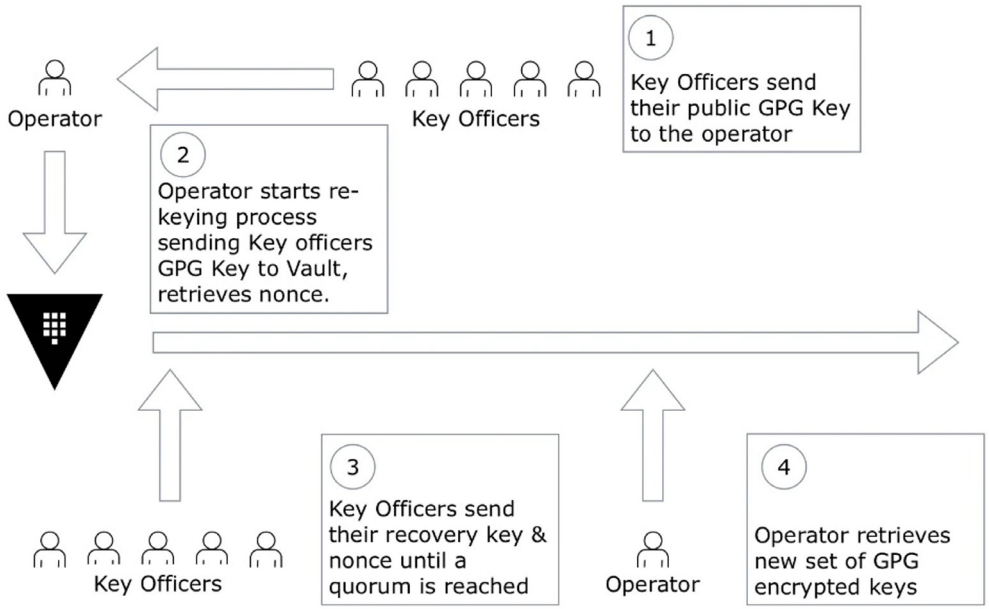
The Master Key wraps the Storage Key, and only lives unencrypted in memory. When using automatic unsealing, the Master Key will be encrypted by the Seal Key, and recovery keys will be provided for certain operations. This process is also online, and causes no disruption, but requires the key holders to input their current shard or recovery key to validate the process, and it is time-bound. This procedure is carried out as follows:

1. Key Holders send their GPG Public keys to the operator.
2. An operator would start the re-keying process, providing the key holders GPG public keys, and retrieve a nonce from Vault.
3. The nonce is handed over to Key Officers to validate the operation. A quorum of Key Officers proceeds to submit their keys.
4. The operator (once a quorum of Key Officers is reached), can retrieve the new keys encrypted using GPG, and hand them over to Key Officers.
5. Key Officers can proceed to validate their new shares.

This procedure should be carried out whenever a Key Holder is no longer available for an extended period. Traditionally in organisations, there is a level of collaboration with a Human Resources department. Alternatively, these procedures already exist for organisations using HSMs and can be leveraged for Vault.

Outside of the scenario above, this procedure is generally carried out by organisations yearly, unless there is a compromise.

An illustration of the procedure can be found below.



Note: This scenario is only valid if you are using manual unseal. For example, with AWS KMS Auto-Unseal, the KMS key will wrap the Master Key, meaning that you only need to rotate the KMS key to accomplish this task.

SEAL KEY

This procedure varies based on the unseal method used. If you are using AWS KMS Auto-Unseal, you could incorporate a strategy for rotating AWS KMS keys for the AWS KMS key associated with the seal stanza in the Vault configuration file.

POLICY MAINTENANCE

When policy is managed centrally, it is quite common to implement a pipeline to maintain policy additions to Vault, to enable a merge approval system, where potential consumers can request access, and both operators and central security teams can allow the merge. Terraform can be used to read policy files and ensure compliance between code and policy.

Policy templates are also used to reduce the number of policies maintained, based on interpolating values from attributes, or key value pairs from entities.

As an example, from an entity that contains an "application" key, with an "APMA" value:

```
$ vault read identity/entity/id/670577b3-0acb-2f5c-6e14-0222d3478ce3
Key                 Value
---                 -----
aliases             [map[canonical_id:670577b3-0acb-2f5c-6e14-0222d3478ce3
creation_time:2018-11-13T13:15:04.400343Z mount_path:auth/userpass/
mount_type:userpass name:nico id:14129432-cf46-d6a6-3bdf-dcad71a51f4a
last_update_time:2018-11-13T13:15:04.400343Z merged_from_canonical_ids:<nil>
metadata:<nil> mount_accessor:auth_userpass_31a3c976]]
creation_time       2018-11-13T13:14:57.741079Z
direct_group_ids    []
disabled            false
group_ids           []
id                  670577b3-0acb-2f5c-6e14-0222d3478ce3
inherited_group_ids []
last_update_time    2018-11-13T13:20:53.648665Z
merged_entity_ids   <nil>
metadata            map[application:APMA]
name                nico
policies            [default test]
```

A policy could exist to give access to a static secret mount that matches the value of a defined key:

```
path "{{identity.entity.metadata.application}}/*" {
  capabilities = ["read","create","update","delete"]
}
```

Governance-oriented policies can be introduced using HashiCorp Sentinel as either Role or Endpoint Governing Policies.

APPLICATION ONBOARDING

Adding new secrets into Vault and enabling new applications to consume them is the most common operation performed in Vault.

Each organisation has its own guidelines in regards to secret generation, consumption, and handover points, but generally speaking, the following aspects are agreed upon in advance:

- The nature of the consumer involvement in the process. Will the consumer be responsible for authenticating with Vault, securing a token, and then obtaining a secret, or is the expectation that there is a certain degree of automation present that secures the token beforehand? The handover point needs to be agreed upon in advance.
- Is it the case that there is a team handling the runtime, and as such, the developer has no involvement, and tools external to the application are going to be used to consume the secret?
- Is the consumer part of a namespace and as such are we supposed to create a role for the new application?
- If using static secrets, is the consumer expected to manually store the secret in Vault, and will it be automatically refreshed by a process?

Most commonly, as described previously, organisations start with auditing static passwords that have been stored in multiple sources and store them into Vault to bring them under management. From there, applications can consume it using the patterns described above, and procedures can be introduced to manage their lifecycle, or effectively shift the lifecycle management to a Secret Engine.

The steps to onboarding an application, otherwise, are quite generic:

- Set up authentication with Vault, or simply create a role mapping to the policy (unless one of the techniques mentioned above is used)
- Define where secrets will be stored
- Set up the application to consume it

AUTH METHODS OVERVIEW

Auth Methods are Vault components that perform authentication and manage identities. They are responsible for assigning an identity and policies to a user.

Once authenticated, Vault will issue a client token used to make all subsequent read and write requests. The goal of all auth methods is to obtain a token, which is associated with one or multiple policies and has a TTL.

There are various **Auth Methods** that can be enabled depending on your use case. An important consideration when deploying Vault on AWS is utilising the **AWS Auth Method**. There is also a great FAUN publication on **Using AWS IAM Auth Method With HashiCorp Vault** that I highly recommend reading.

Auth methods can be enabled/disabled using the Vault API or the CLI:

```
vault auth enable userpass
```

In addition to the general Auth Methods documentation provided by HashiCorp, we recommend following our **short tutorials**, which contain example solutions for particular use cases.

POLICIES OVERVIEW

Vault policies provide operators with a way to permit or deny access to certain actions within Vault based on paths. This gives us the ability to provide granular control over who gets access to what secrets. This is why it is important to follow the principle of least privilege when writing policies.

Vault policies are also:

- Following a declarative syntax and can be written in HCL or JSON.
- Attached to a token. The token can have multiple policies attached.
- Cumulative with additive capabilities
- Deny by Default (implicit deny). All access must be granted explicitly.
- Explicit deny takes precedence over any other permission.

There are two policies created by default when Vault is launched:

- **root policy** – superuser policy that cannot be modified or deleted as it is attached to all root tokens
- **default policy** – common permissions policy, which can be modified but cannot be deleted as it is attached to all non-root tokens (attachments can be removed)

Policies can be listed or read easily using the following commands:

```
# List policies
$ vault policy list
default
root

# Read a policy
$ vault policy read <policy name>
```

Policies also have several capabilities that are associated with action statements:

- Create
- Read
- Update
- Delete
- List
- Sudo
- Deny

Paths can also be customised in several ways, including with a few wildcards that are supported. HashiCorp have provided a wealth of information on customising policies. Links to the policy writing documentation can be found in the previous section.

TOKENS OVERVIEW

Tokens is the core method for authentication into Vault and most operations require an existing token. The token method is a built-in Vault auth method, which is responsible for authenticating, creating, revoking, storing tokens and more.

The token auth method **cannot be disabled** as not only can tokens be used directly, but they can be used through other auth methods as well (e.g., using the AWS IAM auth method still requires a Vault token to perform any action on a particular secret, so it generates or assumes one and returns it in the API request header).

Tokens always have one or more policies attached, controlling the actions which they can perform.

As of Vault 1.0, there are two types of tokens:

- **Service tokens** (default token type), which are:
  - Persisted to storage
  - Can be renewed and revoked
  - Can have child tokens
- **Batch tokens**, which are encrypted binary large objects (blobs) and are:
  - Lightweight and scalable
  - Not persisted to storage (in-memory)
  - Ideal for high-volume operations, such as encryption
  - Can be used for DR Replication cluster promotion

You can find a detailed token type comparison here.

Tokens also carry important metadata that can be retrieved through the CLI or API, this metadata includes:

- Accessor
- Attached Policies
- TTL / Max TTL
- Number of uses left
- Orphaned tokens
- Renewal status

You can use the token lookup command to display the token or accessor metadata.

SECRETS ENGINES OVERVIEW

Vault has a variety of secrets engines that store, generate, or encrypt data. Basic engines will simply store and read data, while more complex engines will connect to external services and have the ability to generate dynamic credentials on demand.

As usual, HashiCorp provides rich documentation on secrets engines and I would recommend looking at:

- Secrets Engines main documentation
- Getting started with Secrets Engines tutorial

However, the documentation is missing a best practices document for organising secrets. While this is a topic that is entirely dependent on a user's or organisation's specific use case, here are some ideas to consider.

Domain Centric Organisation

Engine Path	Secret Path	Key	Value
amazon	publication/node1	access_key	secret
amazon	publication/node1	secret_key	secret
amazon	publication/node2	access_key	secret
amazon	publication/node2	secret_key	secret
cloud	dev	username	password
cloud	prod	username	password

Keep in mind that while this organisation should work for static credentials like IAM user programmatic keys, it might not be appropriate if you are, for example, using the AWS secrets engine to generate dynamic credentials, which will be short-lived and revoked when the lease on their Vault tokens expires.

Node Centric Organisation

Engine Path	Secret Path	Key	Value
node1	cloud/dev	username	password
node1	cloud/prod	username	password
node2	cloud/dev	username	password
node2	cloud/prod	username	password
shared	cloud/cicd_deploy_dev	username	password
shared	cloud/cicd_deploy_prod	username	password

This makes creating policies to limit node access much easier. Nodes can be limited to their specific path only. A shared path for non-node specific credentials can be shared with all nodes.

Again, consider using more than the key value secrets engine whenever possible due to the additional features, like dynamic creation, that are wrapped around them.

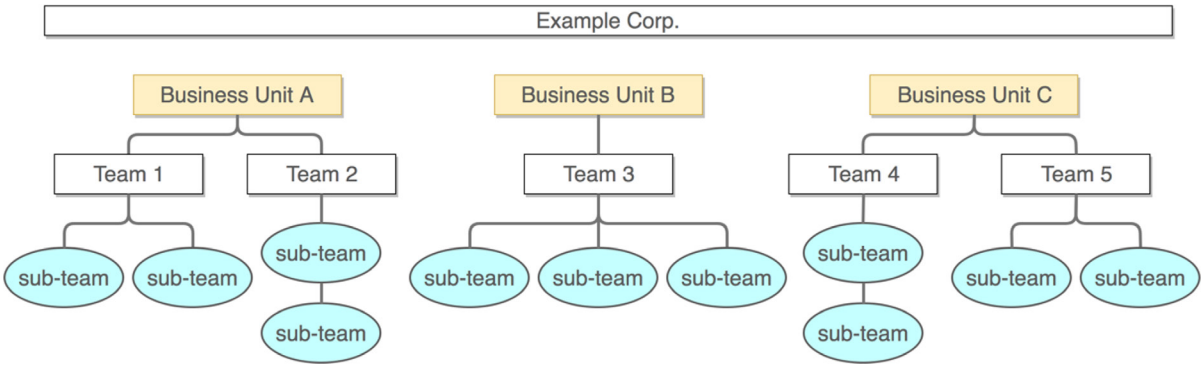
VAULT ENTERPRISE AND ADVANCED FEATURES AND CONSIDERATIONS

Purchasing a Vault Enterprise licence means that you and your organisation are committed to using Vault as a "Centralised Secrets Management" solution. In this section, we will look at some of the key features of Vault Enterprise. We will also discuss some more advanced topics like security hardening, as well as using Consul and Sentinel for Vault.

NAMESPACES

Namespaces are a Vault Enterprise exclusive feature that allows organisations to provide "Vault as a Service", essentially giving them a centrally managed Vault hub. More specifically, namespaces allow us to create:

- Isolated environments on a single Vault environment
- Multi-tenant, but centrally managed solution
- Delegation of Vault responsibilities



Each namespace has its own:

- Policies
- Auth Methods
- Secrets Engines
- Tokens
- Identities

The default Vault namespace is **root**. Other namespaces are created in a hierarchical fashion, making paths relative to each namespace. This in turn makes re-using commands and policies across multiple namespaces a relatively straightforward job.

Tokens are only valid in a single namespace, but you can create an entity that has access to other namespaces.

Once again, you could refer to the HashiCorp documentation for more information on namespaces:

- Vault Enterprise Namespaces
- List Namespaces via the API or CLI
- Secure Multi-Tenancy with Namespaces tutorial

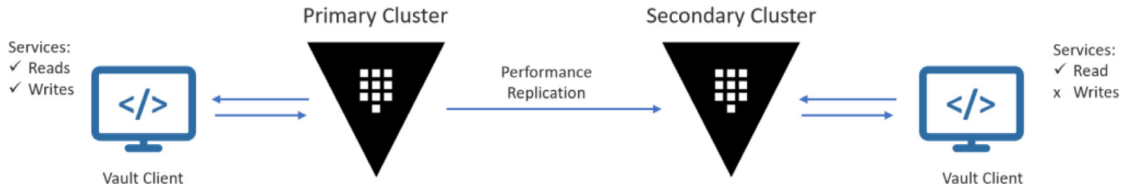
SECURITY HARDENING

Vault is a security product, and it itself must be secured. A mixture of both Vault and OS security hardening is essential for the overall security of a Vault system. HashiCorp have outlined some recommendations in their Production Hardening document. However, for a much more comprehensive list of suggestions, I highly recommend looking at Bryan Krausen's HashiCorp Vault: The Advanced Course.

REPLICATION

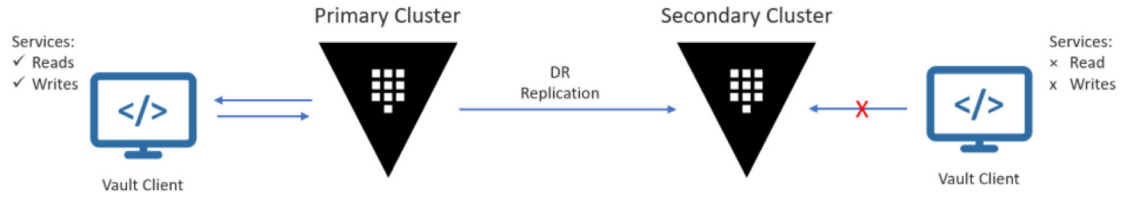
HashiCorp offer both Disaster Recovery Replication (Vault Enterprise), as well as Performance Replication (Vault Enterprise Premium) capabilities. You can architect a Vault system using either one or both features depending on your organisation's needs.

Performance Replication



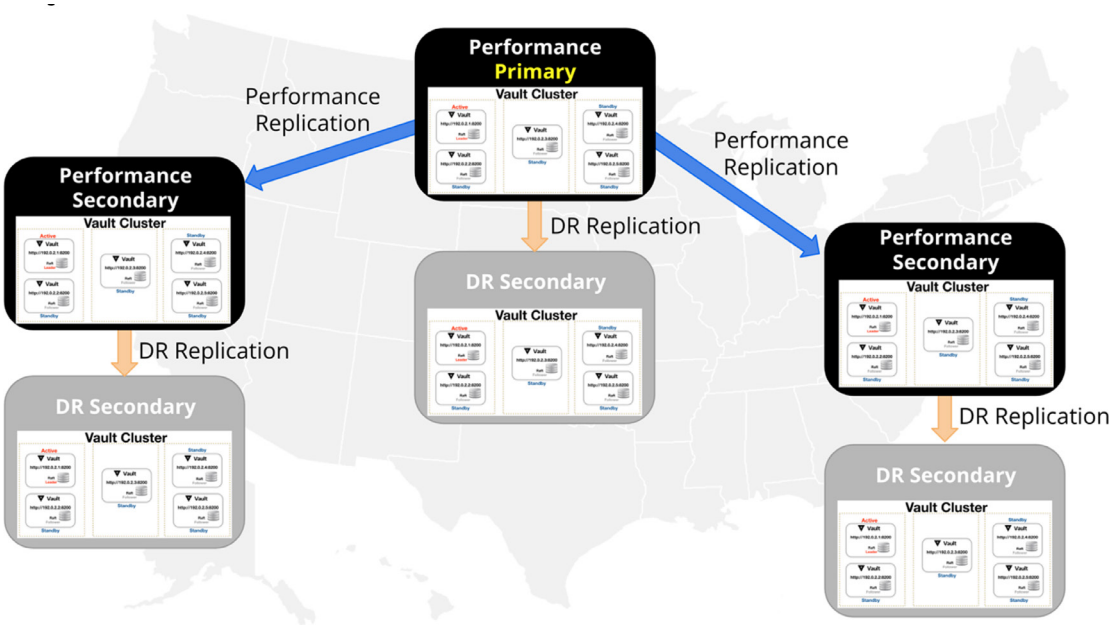
- Replicates the underlying configuration, policies, and other data
- Ability to service reads from client requests
- Clients will authenticate the performance replicated cluster separately
- Does not replicate tokens or leases

Disaster Recovery Replication

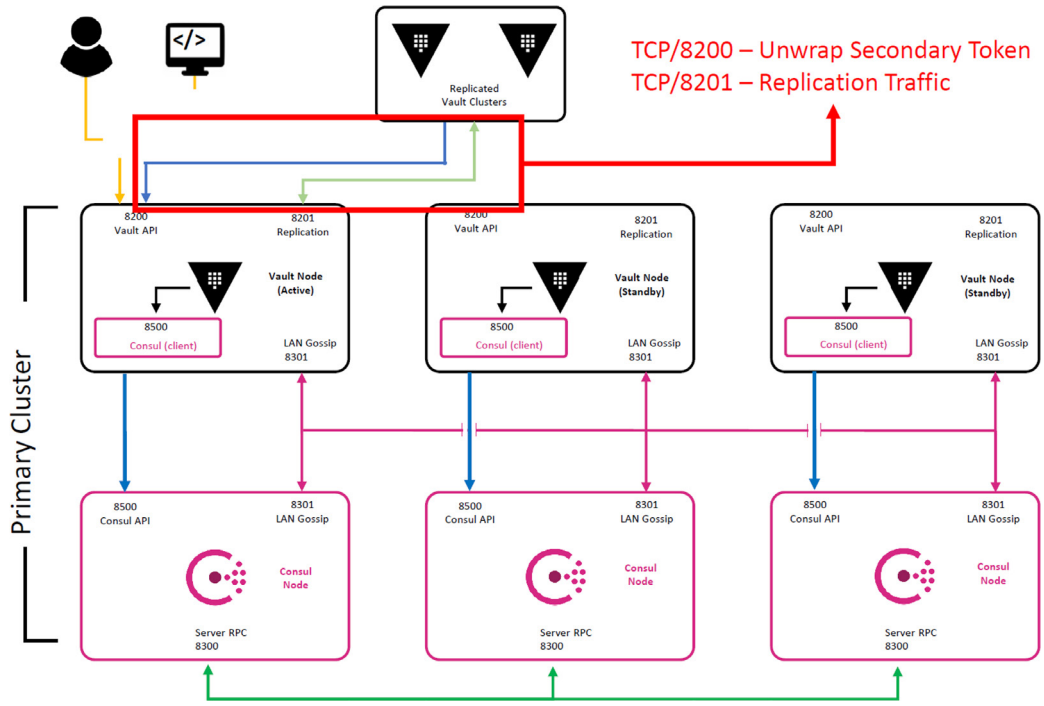


- Replicates the underlying configuration, policies, and all other data
- Cannot service reads from client requests
- Clients should authenticate with the primary cluster only (or a perf cluster)
- Will replicate tokens and leases created on the primary cluster

Using both features form an architecture that looks like:



Communication Model



It is important to note that the so-called **secondary token** is required to permit a secondary cluster to replicate from the primary cluster. This is a one-time use token, as soon as cluster replication is configured successfully, the secondary token becomes redundant.

The secondary token includes sensitive information such as:

- The redirect address of the primary cluster
- The client certificate and CA certificate

Some suggestions for further reading on Vault replication include:

- Vault Enterprise Replication
- Setting up Performance Replication
- Disaster Recovery Replication Setup
- Check Replication Performance Status via API

HASHICORP CONSUL ACLS FOR VAULT

Consul ACLs are a system for controlling access to Consul data and the general Vault communication channels.

- Consul ACLs rely on policies and associated tokens for clients
- Policies are composed of rules permitting or denying access to Consul data and functions
- ACLs must be bootstrapped before they can be used

Why would you need Consul ACLs?

- Vault stores data on Consul's integrated Key Value (KV) store
- The data stored in the KV needs to be protected to ensure the continuous functionality of Vault
- Although the data is encrypted by Vault, the KV data should not be accessed under any circumstances
- The only way to protect this KV data is to use Consul ACLs

Some suggestions for further reading on Consul ACLs include:

- Secure Consul with Access Control Lists (ACLs)
- Consul ACL System
- Consul ACL Rules
- Consul Agent Configuration
- Bryan Krausen's Consul ACLs

HASHICORP SENTINEL FOR VAULT

Sentinel is HashiCorp's Policy as Code product, which provides additional access control capability to Vault (requires Vault Enterprise licence).

Sentinel adds two additional Vault policy types:

- **Role Governing Policies (RGPs)** – policies tied to specific tokens, entities, or groups
- **Endpoint Governing Policies (EGPs)** – policies associated with particular paths

All Sentinel policies also have three available enforcement levels:

- **Advisory** – the policy is allowed to fail
- **Soft Mandatory** – the policy must pass unless an override is specified
- **Hard Mandatory** – the policy must pass no matter what

Some typical use cases for using Sentinel policies in a Vault system include:

- Validation of input data
- Limiting access from a specific CIDR or IP address
- Disallow certain configurations in Vault
- Ensure access only during business hours and/or workdays
- Deny all previously created tokens

Some suggestions for further reading on Sentinel for Vault include:

- HashiCorp Sentinel Documentation
- Sentinel Policies for Vault tutorial
- Sentinel Properties for Vault
- Sentinel Enforcement Levels
- Bryan Krausen's example Sentinel policies for Vault (covers most of the use cases described above)

ALTERNATIVE SOLUTIONS TO VAULT

When you are considering building a HashiCorp Vault system on AWS, it is important to be aware that AWS offer two managed services in the form of AWS **Secrets Manager** and AWS **Systems Manager Parameter Store**. These are meant to tackle some of the issues that Vault tackles. While Vault offers a more sophisticated solution with a plethora of additional features, you need to be certain that it is the most suitable option for your business use case before choosing it.

We are going to look at some of the differences between the aforementioned solutions, which should make choosing the appropriate one relatively straightforward.

DEPLOYMENT AND SETUP

HashiCorp Vault

You are in charge of setting up and scaling the service and managing the infrastructure security and operations. This allows for a much higher degree of customisation at the cost of effort in deploying and maintaining the system.

AWS managed services are fully managed by AWS and are as straightforward to use as it gets. They also have built-in resiliency capabilities.

SCALABILITY AND FLEXIBILITY

HashiCorp Vault has an immense number of plug-ins and operations aiming to support nearly all tech – both on-premise and AWS.

AWS managed services scale effortlessly since AWS takes care of it, but have fewer features than Vault and support fewer use cases. Natively integrates with a lot of AWS services.

PRICING

AWS Secrets Manager

Check pricing here: https://aws.amazon.com/secrets-manager/pricing/

AWS Parameter Store

Check pricing here: https://aws.amazon.com/systems-manager/pricing/

HashiCorp Vault

HashiCorp Vault is a free, open source product with an add-on enterprise offering. The enterprise platform includes disaster recovery, namespaces, and monitoring, as well as various features for scale and governance. You can see the full breakdown of features on the HashiCorp Vault pricing page. Deploying Vault servers in the recommended HA setup could, however, be significantly more costly compared to the AWS native services.

FEATURES

Parameter	HashiCorp Vault	AWS Services (Secrets and Systems Manager)
Supported type of secrets	SSH keys, TLS keys, Docker credentials, service accounts tokens – OpenShift\Kubernetes\ Nomad, login pass pairs, services and API certificates, env variables	SSH keys, Docker credentials, service accounts tokens – Kubernetes, login pass pairs, services and API certificates, env variables
Supported types of applications	Cloud – AWS, GCP, Azure, Alibaba Cloud, databases – Oracle, MS SQL, MySQL, CI \ CD – GitLab CI, TeamCity, Container Orchestration – Kubernetes, OpenShift, Swarm, IaaC – Ansible, Terraform, Helm	Cloud – AWS, databases – Amazon RDS, Amazon Redshift, Amazon DocumentDB, CI \ CD – GitLab CI, Jenkins, Container Orchestration – Kubernetes, Swarm, IaaC – Terraform, CloudFormation
Control Groups (Approval Workflows)	Yes (Vault Enterprise Plus)	No
Authentication	Tokens, TLS certificate based, Cloud platform based (for example via AWS IAM), OIDC, SAML, Kerberos and SAML	IAM (which supports OIDC and SAML by extension)
Dynamic Secrets and Secrets rotation	Yes	Yes (Secrets Manager only)
ACL	Yes	Yes
Secrets wrapping for one-time access	Yes	No
Management interfaces	API, CLI, Web	API, CLI, Web
High availability	Yes	Yes
Logging access to secrets	Yes, it cannot work if there is no audit device enabled	Yes, via Cloudtrail
Tokenisation for sensitive data	Yes (Vault Enterprise)	No
Editions	Open Source, Enterprise + Modules, Cloud Managed	N/A
Delivery type	On-Premise, IaaS, SaaS (via HashiCorp Cloud Platform)	SaaS
Availability on cloud marketplaces	AWS, Azure, Google Cloud	AWS

CONCLUSION

Having a centralised secrets management system should be an essential part of any modern business that has applications and services deployed in AWS and beyond.

While AWS offer convenient managed solutions for building a working secrets management system that are easier to setup, HashiCorp Vault is a feature-rich product that has a considerable advantage in terms of customisation and control and should be considered by any organisation that values security.

REFERENCES

https://learn.hashicorp.com/

https://www.vaultproject.io/docs

https://www.terraform.io/docs

https://docs.hashicorp.com/consul

https://docs.hashicorp.com/sentinel

https://aws.amazon.com/quickstart/

https://medium.com/hashicorp-engineering/certificates-issuing-and-renewal-with-vault-and-consul-template-18e766228dac

https://github.com/btkrausen/hashicorp

WWW.HELECLOUD.COM