Adopting HashiCorp Vault

Deployment, Adoption, and Beyond

GUIDE

HashiCorp

Preface **3**

Day Zero **4**
Vault Internals and Key Cryptography Principles 4
Vault Solution Architecture 6
Single Site 6
Multiple sites 9

Day One **10**
Deployment Patterns 10
Monitoring 11
Telemetry 12
Audit 12
Backup and Restore 12
Failure Scenarios 12
Initialization Ceremony 12
Initialization 14
Basic Configuration 14
Root Token Revocation 18
Organizational Roles 18

Day Two **19**
Namespacing 19
Secure Introduction 19
Storing and Retrieving Secrets 20
API 20
Non-invasive patterns 20
Response wrapping 21
Encryption as a Service 21

Day N **23**
Storage Key Rotation 23
Master Key Rotation 23
Seal Key Rotation 24
DR Promotion 25
Policy Maintenance Patterns 26
Application onboarding 27

Preface

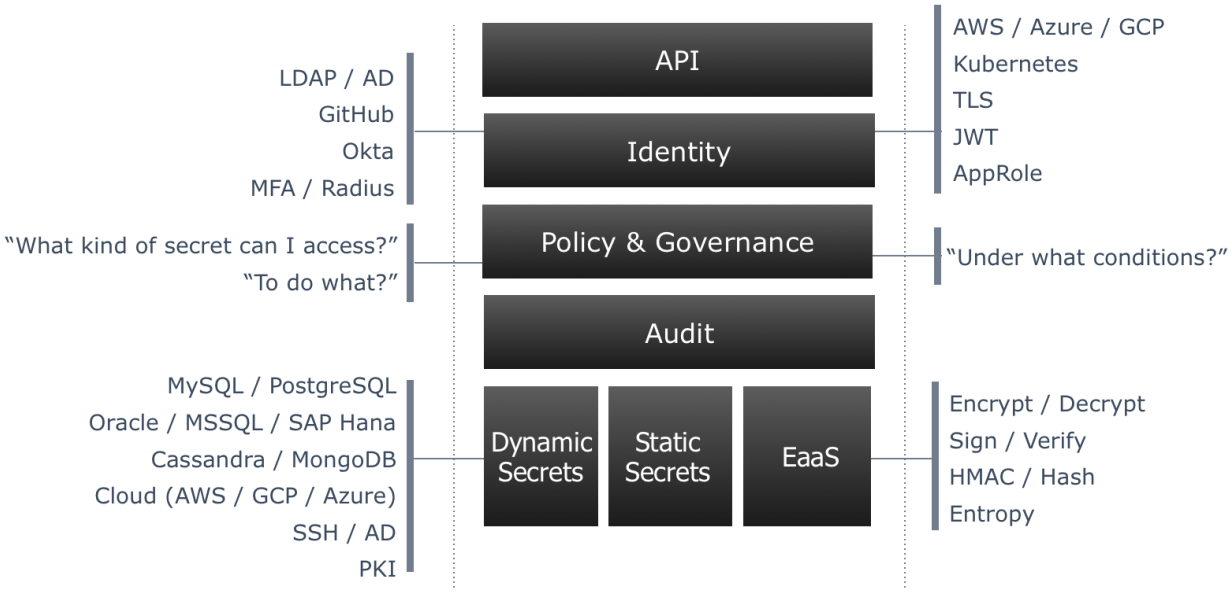
As with every HashiCorp product, when adopting Vault there is a "Crawl, Walk, Run" approach. As such, this document intends to provide some predictability in terms of what would be the required steps in each stage of HashiCorp Vault deployment and adoption, based both on software best practice and experience in deploying Vault at scale in large organizations. This document is not intended to be in depth documentation into Vault, but rather provide an overview of the journey, referencing documentation when appropriate, and focusing on operational aspects.

Parts of this document are intended to be foundations for an organizational runbook, describing certain procedures commonly run in Vault.

Day Zero

Vault Internals and Key Cryptography Principles

HashiCorp Vault is a secrets management solution that brokers access for both humans and machines, through programmatic access, to systems. Secrets can be stored, dynamically generated, and in the case of encryption, keys can be consumed as a service without the need to expose the underlying key materials. The workflow in terms of secret consumption can be summarised using the following diagram:

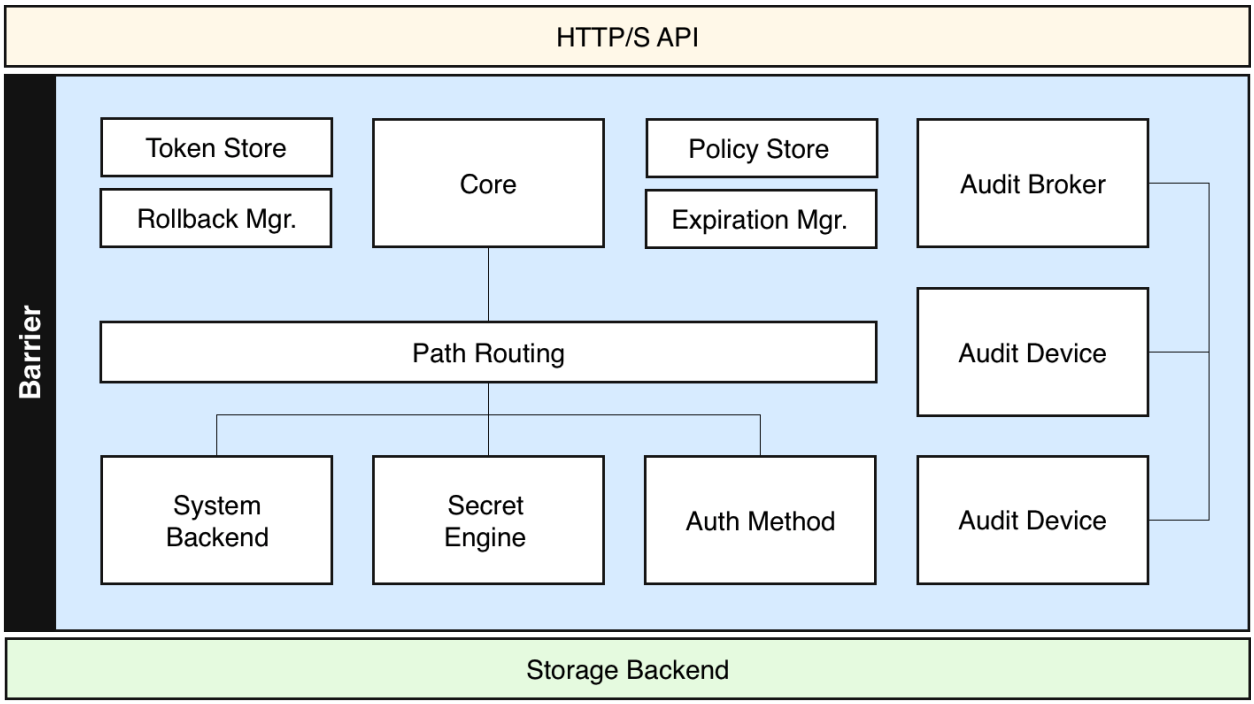


- Vault's primary interface is through a HTTP Restful API. Both the CLI and the Web GUI interface with Vault through the same API. A developer would use this API for programmatic access. There is no other way to expose functionality in Vault other than through this API.
- In order to consume secrets, clients (either users or applications) need to establish their identity. While Vault supports the common authentication platforms for users, such as LDAP or Active Directory, it also adds different methods of programatically establishing an application identity based on platform trust, leveraging AWS IAM, Google IAM, Azure Application Groups, TLS Certificates, and Kubernetes namespaces among others. Upon authentication, and based on certain identity attributes like group membership or project name, Vault will grant a short lived token aligned with a specific set of policies.
- Policies in Vault are decoupled from identities and define what set of secrets a particular identity can access. They are defined as code in HashiCorp Configuration Language (HCL). Rich policy can be defined using Sentinel rules[1], that are designed to answer "under what condition" an entity would get access to the secret, rather than the traditional "who gets access" to what secret defined in regular ACLs.

[1] https://www.vaultproject.io/docs/enterprise/sentinel/index.html

- Vault sends audit information to a SIEM system or logging backend via Syslog, File or Socket. Vault will not respond if it cannot provide audit information appropriately.
- Ultimately Vault can either store or generate secrets dynamically. By virtue of "mounting" an engine:
  - Static secrets can be stored and versioned using the KV/2 engine.
  - Secrets of different types can be dynamically generated using different engines, for Databases, SSH / AD access, PKI (X.509 Certificates) among others
  - Key Material can be consumed via specific endpoints that provide encryption/decryption/signing/verification/HMAC capabilities, without ever leaving Vault.

Vault internal architecture can be summarized using the following diagram:

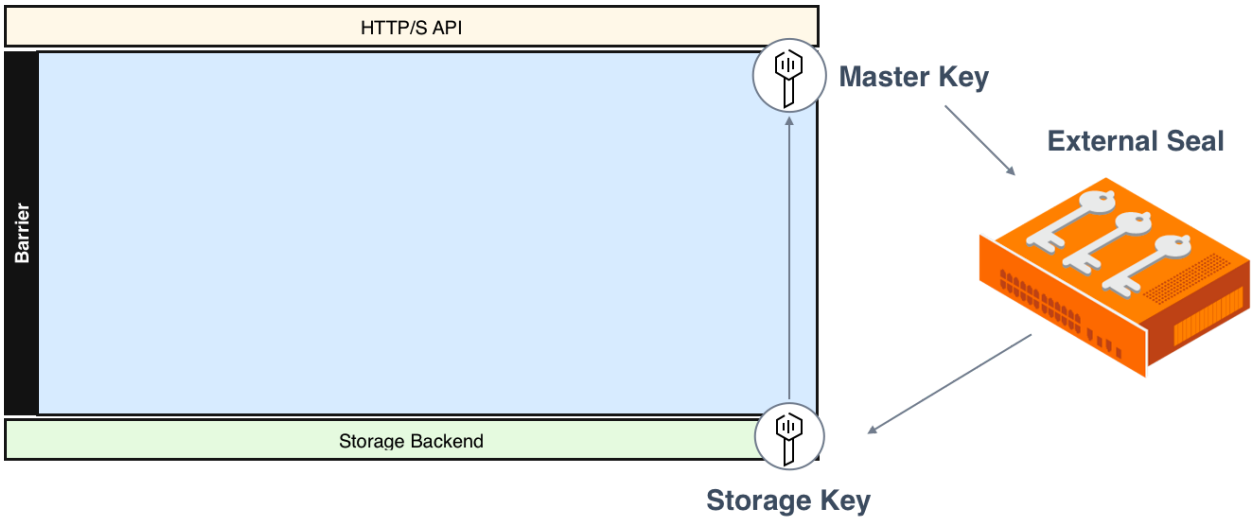


From a data organization perspective, Vault has a pseudo-hierarchical API path, in which top level engines can be mounted to store or generate certain secrets, providing either an arbitrary path (i.e. /secret/sales/password), or a predefined path for dynamic secrets (e.g. /pki/issue/internal).

No secret leaves the cryptographic barrier (which only exists in volatile memory on the Vault host) in unencrypted form. This barrier is protected by a set of keys. Information served through the HTTP API is encrypted using TLS. Information stored in the storage backend is encrypted using a Storage Key, which is encrypted by a Master Key, which is optionally encrypted by an external Seal for operational ease, and automates the unsealing procedure[2], as shown in the diagram below. The same seal can be also used to

[2] https://www.vaultproject.io/docs/internals/architecture.html

encrypt secrets. For example, for secrets that need to be encrypted in a manner compliant with FIPS, Vault's "Seal Wrap[3]" in conjunction with an HSM can be used to achieve this functionality.



Vault Solution Architecture

HashiCorp Vault is designed using distributed systems concepts and paradigms. As such, there are many possibilities in terms of deployment, but only a handful are thoroughly tested and supported by HashiCorp. There are different strategies when it comes to single site or multisite resiliency and scaling requirements. For the purpose of this document, and any architectural decisions, you should consider a site as a geographical area that does not exceed eight milliseconds in round trip time.

Single Site

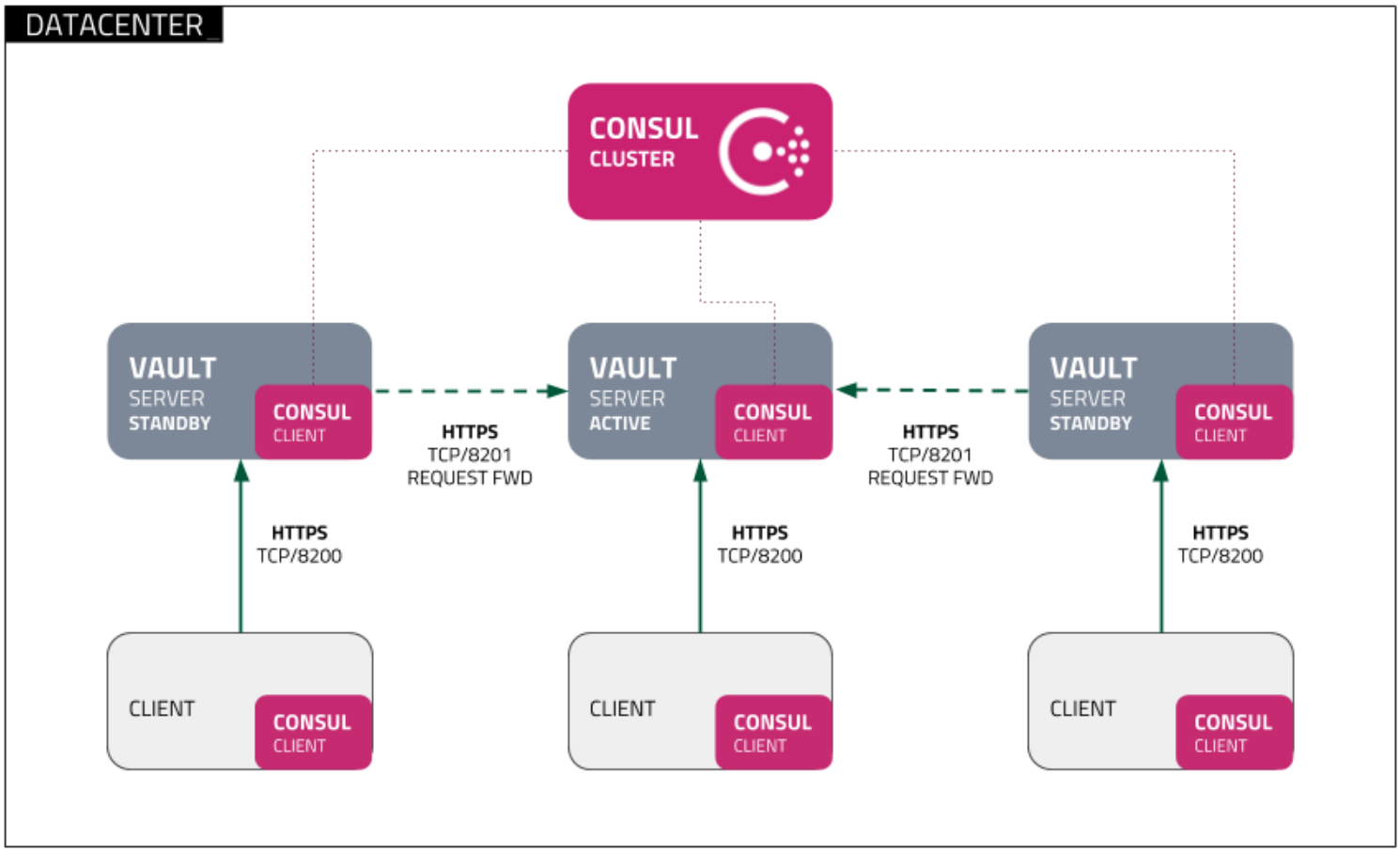
When it comes to a single site, a production deployment of Vault is a clustered unit, that would generally consist of three nodes. In its open source variant, one node is automatically designated as "Active", while the remaining two are in "Standby" mode (not actively providing service). These "Standby" nodes would take the "Active" role automatically, upon orderly shutdown of the "Active node" (in which case failover is immediate), or failure (failover under an unclean shutdown will take 45 seconds), when using HashiCorp Consul, the recommended storage backend. When Vault Enterprise Premium is being used, the standby nodes can be set up as "Performance Standbys", which would enable them to scale the Cluster horizontally, by virtue of answering certain queries directly, while transparently forwarding some others to the Active node in the cluster. As mentioned before, there is a hard limitation in terms of latency between Active and Standby nodes.

[3] https://www.vaultproject.io/docs/enterprise/sealwrap/index.html

Vault can use many different Storage Backends. However, HashiCorp only offers support for Vault clusters using Consul as a truly scalable production grade solution. Typically the Consul backend is deployed as a 5 node cluster to support a 3 node Vault cluster.

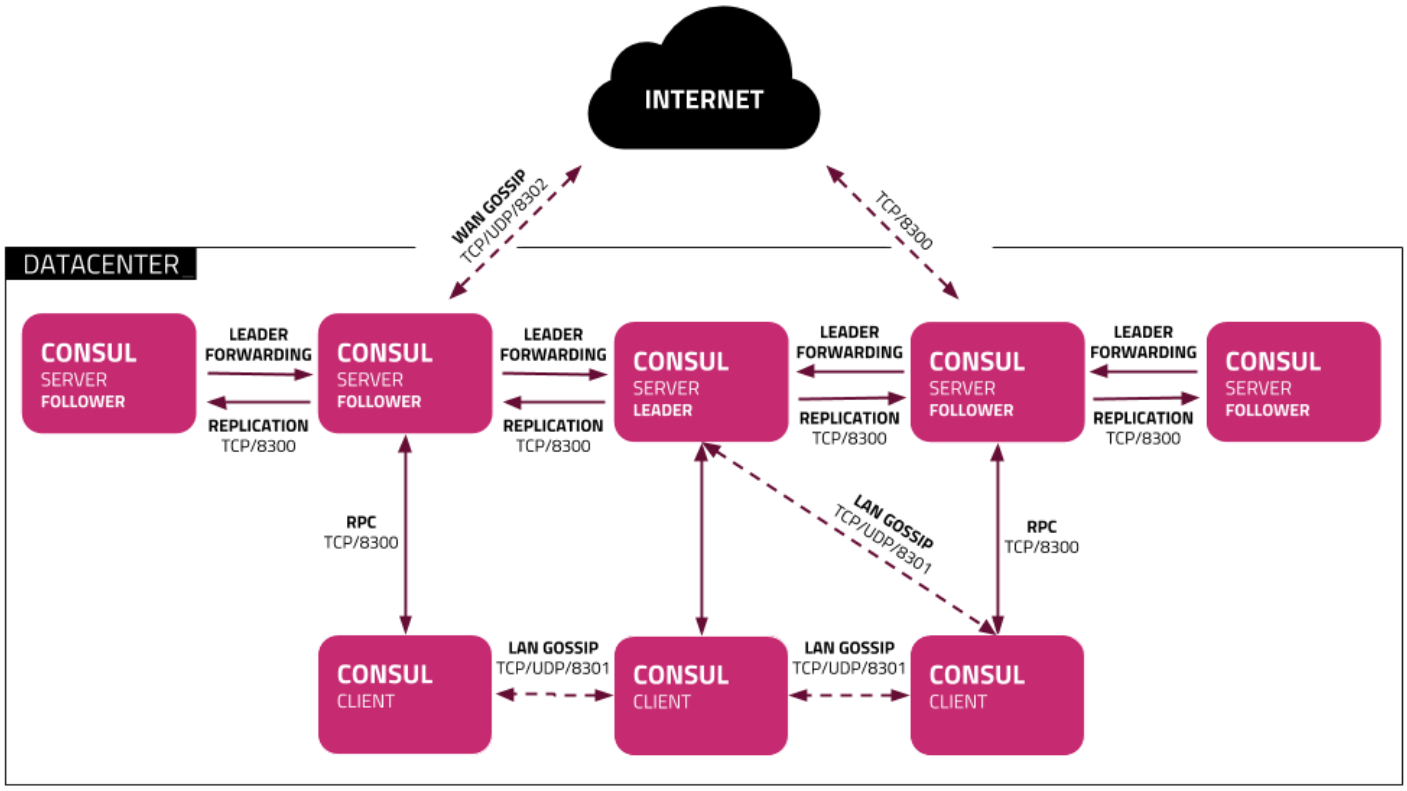
HashiCorp Consul must maintain a quorum at all times to provide services (N+1 of the nodes/votes), and quorum loss is not an acceptable operational scenario; as such, there are a number of strategies and *Autopilot*[4] features available in the product to ensure service continuity and scaling, such as Non-Voting members, Redundancy Zones and Update Migrations.

The most common single site Vault deployment would look as follows:



[4] https://www.consul.io/docs/guides/autopilot.htm

The Consul cluster icon in the above image consists of the components illustrated in the following diagram (in this situation, Consul clients would be the Vault servers):



This architecture is designed to accommodate the following principles:

- A maximum of two nodes of each service may fail without causing disruption, although if using "Performance Standbys" there might be degradation, until further servers are reprovisioned.
- Systems are spread across an odd number of network areas or availability zones, to accommodate for the failure of up to a full zone.
- There is a single data source (storage backend), and a single cryptographic seal in each area.

Multiple sites

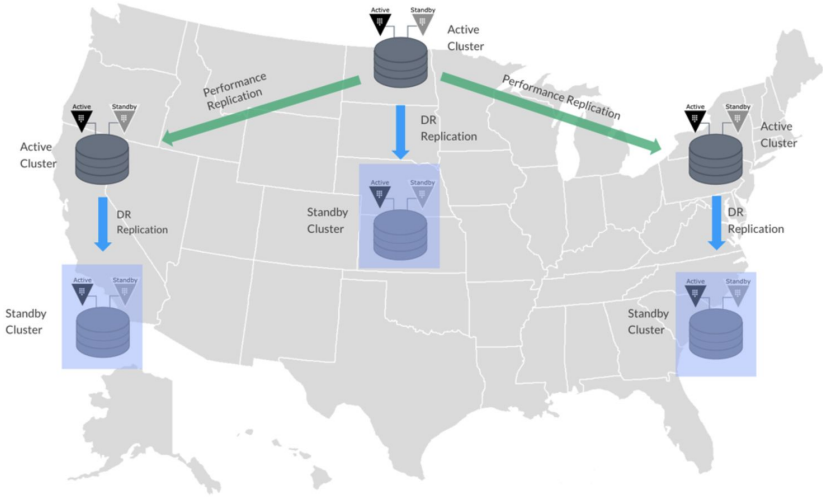
HashiCorp Vault provides two replication modes[5], in which a Primary cluster would selectively ship logs to Secondary Clusters.

The minimum requirement from a resiliency perspective, is to provision a Disaster Recovery (DR) Replica, which is a warm standby and holds a complete copy of everything. A DR Replica is not able to answer requests until promoted. From an operational perspective, a DR Replica would provide service continuity in case of:

- **Data tampering:** Unintended or intentional manipulation of the storage backend, which holds binary encrypted data. This aspect can be somewhat mitigated through backup / restore techniques, with a considerable impact on RPO and RTO, which a DR replica is not subject to.
- **Seal Tampering:** The key, being integral to cryptographic assurances, is not recoverable. If the encryption key were to be deleted or tampered, Vault would not be able to recover. A DR Replica has a separate seal, to which access should be segregated.
- **Infrastructure failure:** If the Primary cluster were to become unavailable from a Network or Compute perspective, a DR Replica can be promoted to provide service continuity with no impact.

More commonly, organizations with presence in multiple geographical locations will create a set of Performance Replicas, spread geographically. While this setup can provide replication of secrets and configuration within locations, most importantly it unifies a single Keyring (Master Key & Storage Key) across locations, which is paramount to reduce operational overhead when using Vault at scale.

In Performance Replication mode, selected paths are replicated to different clusters, and operations as much as possible are carried out in Secondary Clusters to minimize latency. Both replication modes can be combined, which would allow a broad scale of both operational efficiency and resiliency, as illustrated in the attached diagram.



[5] https://www.vaultproject.io/docs/enterprise/replication/index.html

Day One

Deployment Patterns

The preferred deployment pattern for the recommended architecture is to perform immutable builds of HashiCorp Vault and Consul, and perform a Blue/Green deployment[6]. As such, tools like HashiCorp Packer[7] are recommended to build immutable images for different platforms, and HashiCorp provides a number of examples[8] in regards to how to build these elements through existing CI/CD orchestration.

HashiCorp recognizes that there are organizations that have not adopted those patterns, and as such there are a number of configuration management patterns to install, upgrade and configure Vault, available in different repositories:

- In Ansible Galaxy, the Vault role by Brian Shumate, available in https://galaxy.ansible.com/brianshumate/vault
- In Ansible Galaxy, the Consul role by Brian Shumate, available in https://galaxy.ansible.com/brianshumate/consul
- In the Puppet Forge, the Vault module, available in https://forge.puppet.com/jsok/vault/readme
- In Chef Supermarket, the hashicorp-vault cookbook, available in https://supermarket.chef.io/cookbooks/hashicorp-vault

It is recommended to restrict SSH access to Vault servers, as there are a number of sensitive items stored in volatile memory on a system.

Vault Release Cycle

HashiCorp releases major versions of Vault quarterly, as well as minor releases as required, though at the very least monthly, so the process of updating Vault in an organization should be fairly streamlined and highly automated.

[6] Blue/Green deployment refers to the pattern where a straight copy of the infrastructure gets deployed ("Blue") and the previous production infrastructure ("Green") gets decommissioned when the service is failed over to the new copy.
[7] https://www.packer.io/

[8] https://github.com/hashicorp/guides-configuration/tree/master/vault

Monitoring

Vault listens for requests on a single port (both service, and management), which as mentioned previously is an HTTP REST endpoint. By default this is TCP port 8200, and there is an unprotected status endpoint that can be used to monitor the state of a cluster, as shown below:

```
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"
}
```

The individual seal status of a node can also be queried, as shown below:

```
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
}
```

In a best practice setup, HashiCorp Consul would monitor the status of Vault, and can provide either Service Discovery via DNS[9], or automatically configure a number of popular open source load balancers, as documented in the official Reference Architecture Guide[10]. Detail step by step guides are also available in the HashiCorp Vault documentation[11].

Telemetry

When running Vault at large scale, it is recommended to profile the performance of the system by using the Telemetry output of Vault in conjunction with a telemetry analysis solution such as StatsD, Circonus or Datadog.

Audit

From a Vault perspective, the use of a SIEM system is a requirement to keep track of access brokering. Vault provides output through either Syslog, a File, or a Unix socket. Most organizations parse and evaluate this information through Splunk or using tooling from the ELK Stack. The Audit output is JSON data, enabling organizations to parse and analyze information with ease.

Backup and Restore

Backup of the solution is done through the Consul Snapshot Agent[12], which can either upload an encrypted backup automatically to an S3 bucket, or leave the backup on the file system to get shipped out by a traditional enterprise backup solution.

Failure Scenarios

- In case of an individual node failure, or up to two node failures, the solution will continue to run without operator intervention.
- If the Vault cluster fails, it can be re-provisioned using the same Storage Backend configuration.
- If the Consul cluster were to lose quorum, there are alternatives to regain service availability, although the recommended approach from an RTO/RPO perspective is to fail over to a DR Cluster or promote a Performance Replica.
- If the Seal Key was to be deleted or unavailable, the only supported scenario is failing over to a DR Cluster or Performance Replica.

Initialization Ceremony

While every action in Vault can be API driven, and as such automated, the initialization process guarantees the cryptographic trust in Vault. The Key Holders, which hold either the Unseal Keys, or most commonly the Recovery Keys, are the guardians of Vault trust. This process would only be carried out once for any Vault installation.

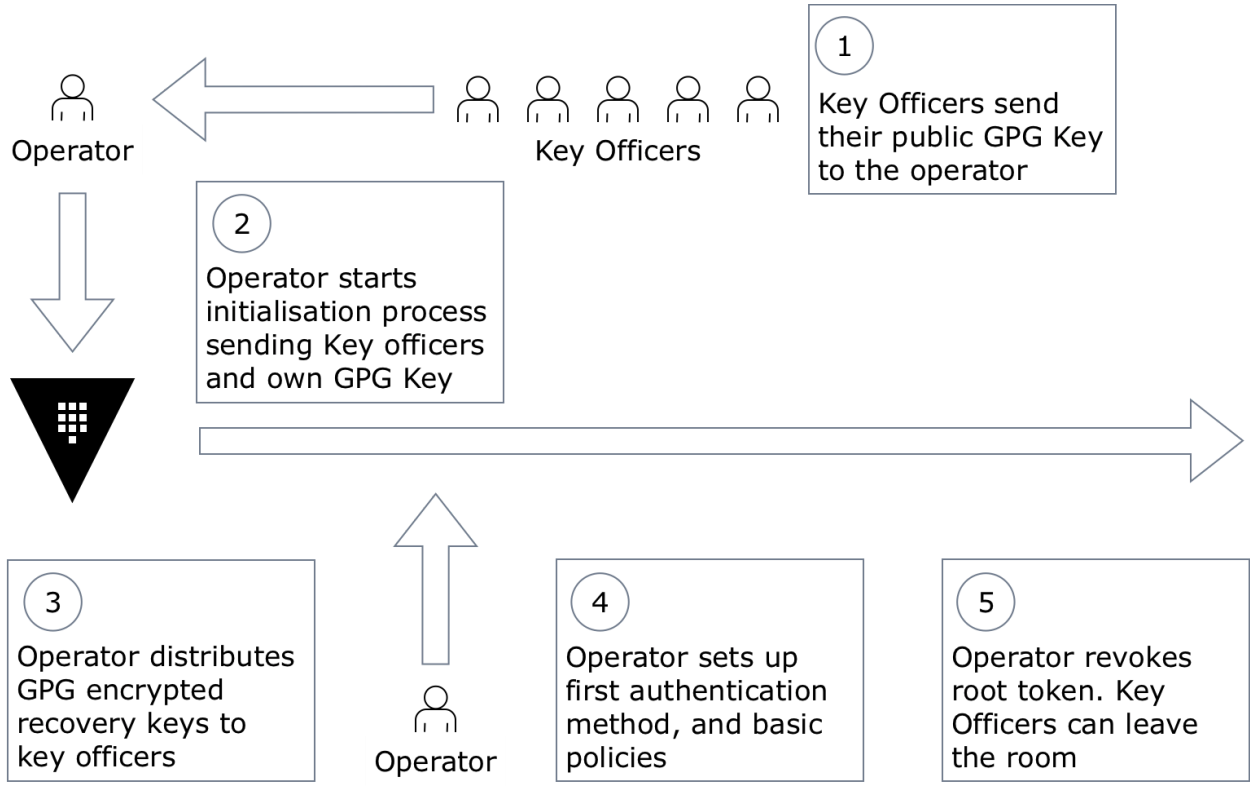
---

[9] https://learn.hashicorp.com/vault/operations/ops-reference-architecture#load-balancing-using-consul-interface
[10] https://learn.hashicorp.com/vault/operations/ops-reference-architecture#load-balancing
[11] https://learn.hashicorp.com/vault/operations/monitoring
[12] https://www.consul.io/docs/commands/snapshot/agent.html

While the initialization process goes through, Vault is at its most vulnerable, as a root token exist. This root token overrides the policy system and would only be used at an initial stage to configure the production authentication method and basic policy, or in a scenario where the primary authentication method is unavailable, in which case a root token needs to be regenerated.

It is recommended that the initialization ceremony is carried out on a single room, where an operator and the Vault key holders would be present throughout the process, which would be as follows:

1. The Operator starts the Vault daemon and the initialization process, as described in the documentation[13], providing the public GPG keys from the Keyholders, and the Operators own public GPG key for the root token.
2. Vault will return the GPG encrypted recovery keys, which should be distributed among the keyholders.
3. The operator uses the root token for loading the initial policy and configuring the first authentication backend, traditionally Active Directory or LDAP, as well as the Audit backend.
4. The operator validates they can log into Vault with their directory account, and can add further policy.
5. The operator revokes the root token. The Keyholders can now leave the room with the assurance that no one person has full and unaudited access to Vault.



[13] https://www.vaultproject.io/docs/commands/operator/init.html

Initialization

The full set of options for initialization is described in the Vault Documentation quoted in the footnote, though the following parameters should be considered:

- -root-token-pgp-key: The Operator Public PGP key that will be used to encrypt the root token
- -recovery-shares/threshold: Number of keys to provision (based on number of keyholders), and quorum of key holders needed to perform recovery operations (generally half of the keyholders plus one)
- -recovery-pgp-keys: Similar to the parameter above, list of Public PGP keys from the keyholders in order to provide an encrypted output of the Key Shares

Basic Configuration

In order to proceed with further configuration without the need of using a Root Token, an alternate authentication method must be configured. Traditionally this is done though the LDAP Authentication Backend configuring an Active Directory / LDAP integration, or most recently through OIDC or Okta support. A group or claim is defined as providing certain administrative attributes to Vault.

Furthermore, policy is defined in regards to 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 at a later time based on requirements. An example administrative policy could be defined as follows:

```
## Operations

# List audit backends
path "/sys/audit" {
  capabilities = ["read","list"]
}

# Create an audit backend. Operators are not allowed to remove them.
path "/sys/audit/*" {
  capabilities = ["create","read","list","sudo"]
}

# List Authentication Backends
path "/sys/auth" {
  capabilities = ["read","list"]
}
```

```
# CRUD operations on Authentication Backends
path "/sys/auth/*" {
  capabilities = ["read","list","update","sudo"]
}

# CORS configuration
path "/sys/config/cors" {
  capabilities = ["read", "list", "create", "update", "sudo"]
}

# Start root token generation
path "/sys/generate-root/attempt" {
  capabilities = ["read", "list", "create", "update", "delete"]
}
# Configure License
path "/sys/license" {
  capabilities = ["read", "list", "create", "update", "delete"]
}

# Get Storage Key Status
path "/sys/key-status" {
  capabilities = ["read"]
}

# Initialize Vault
path "/sys/init" {
  capabilities = ["read", "update", "create"]
}

#Get Cluster Leader
path "/sys/leader" {
  capabilities = ["read"]
}

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

# Manage Mounts
path "/sys/mounts*" {
  capabilities = ["read", "list", "create", "update", "delete"]
}
```

An example Auditor policy could be defined as follows:

```
## Audit
# Remove audit devices
path "/sys/audit/*" {
  capabilities = ["delete"]
}

# Hash values to compare with audit logs
path "/sys/audit-hash/*" {
  capabilities = ["create"]
}

# Read HMAC configuration for redacting headers
path "/sys/config/auditing/request-headers" {
  capabilities = ["read", "sudo"]
}

# Configure HMAC for redacting headers
path "/sys/config/auditing/request-headers/*" {
  capabilities = ["read", "list", "create", "update", "sudo"]
}

# Get Storage Key Status
path "/sys/key-status" {
  capabilities = ["read"]
}
```

An example Key Officer policy could be defined as follows:

```
## Key Officers
path "/sys/generate-root/update" {
  capabilities = ["create", "update"]
}

# Get Storage Key Status
path "/sys/key-status" {
  capabilities = ["read"]
}

# Submit Key for Re-keying purposes
path "/sys/rekey-recovery-key/update" {
  capabilities = ["create", "update"]
}

# Rotate Storage Key
```

—

```
path "/sys/rotate" {
  capabilities = ["update", "sudo"]
}

# Verify update
path "/sys/rekey-recovery-key/verify" {
  capabilities = ["create", "update"]
}
```

These policies are not exhaustive, and while three profiles are defined, in most organizations role segregation runs even deeper.

It's also worth noting that due to the sensitivity of certain endpoints, most organizations chose to use Control Groups[14] in order to require an approval workflows for some configuration changes, for example, when adding or modifying policy, like in the example below:

```
# Create a Control Group for approvals for CRUD operations on policies
path "/sys/policies*" {
    capabilities = ["create", "update"]
    control_group = {
        ttl = "4h"
        factor "tech leads" {
            identity {
                group_names = ["managers", "leads"]
                approvals = 2
            }
        }
        factor "CISO" {
            identity {
                group_names = ["infosec"]
                approvals = 1
            }
        }
    }
}
```

In this way, policy changes would require two approvers from either the "managers" or "leads" group, and one from the "infosec" group.

[14] https://www.vaultproject.io/docs/enterprise/control-groups/index.html

Root Token Revocation

As mentioned previously in the guide, the root token should only be used for initialization and emergency "break glass" operations. Upon configuring Vault, the root token should be revoked and normal authentication methods should be used to introduce changes.

An operator along with a quorum of key holders can re-generate the root token in an emergency scenario.

Organizational Roles

In most organizations where Vault has been deployed at scale, there is no requirement for extra staffing or hiring. In terms of deploying and running the solution. Vault has no predefined roles and profiles, as the policy system allows very granular definitions of the duties for different teams, but generally speaking these have been defined in most organizations as follows:

Consumers: Individuals or teams that require the ability to consume secrets or have a need for a namespaced Vault capability.

Operators: Individuals or teams who onboard consumers, as well as secret engine capabilities, policies, namespaces and authentication methods.

Crypto: Individual or teams who manage key rotation and audit policies.

Audit: Individual or teams who review and audit usage.

Day Two

Namespacing

Vault Enterprise allows an organization to logically segment a Vault installation.

Traditionally, in organizations where Vault is run as a central capability, certain teams that are required to maintain a large set of policies would get their own namespace. This can also be segmented, aligning for example, with a set of Kubernetes namespaces.

Tokens obtained at top namespaces can be segmented to traverse multiple namespaces.

For some organizations where there is a dedicated team running Vault, namespaces may not add significant advantages, and could potentially increase complexity.

Namespaces are generally provisioned in an automated manner, in line with team or project onboarding, for example when a new Kubernetes namespace or AWS account is provisioned.

Secure Introduction

Ahead of being able to consume secrets, a user or an application has to login into Vault, and obtain a short lived token. The method used for applications is generally based on either the platform where the application is running *(AWS, GCE, Azure, Kubernetes, OIDC)*, or the workflow used to deploy it *(AppRole with a CI tool, like Jenkins, or Circle CI).*

Token must be maintained client side, and upon expiration can simply be renewed. For short lived workflows, traditionally tokens would be created with a lifetime that would match the average deploy time and left to expire, securing new tokens with each deployment.

Authentication methods are generally configured by an operator at initial configuration time.

Most commonly, systems perform an authentication process automatically, though responsibility of carrying out the process is generally agreed as part of a handover, when multiple teams take responsibility of provisioning a system or deploying an application.

Based on existing attributes (like LDAP Groups, OIDC Claims, IAM roles, Google Project ID, etc...) roles are created in the different authentication backends, which map to policies (that ultimately grant access to secrets).

Secure introduction procedures are documented in detail under each authentication method[15], and there are a number of guides available providing step by step documentation.

Storing and Retrieving Secrets

Vault will either store or dynamically generate secrets that can be accessed programmatically or in an interactive manner.

The most common pattern for adoption, is starting by storing secrets in Vault that have traditionally been spread across files, desktop password management tools, version control systems and configuration management, and utilize some of the consumption patterns described below as means to get the secrets to the applications consuming it.

Moving forwards most organizations start adopting Secret Engines as a way to reduce overhead, such as using PKI as an intermediate CA and generating short lived certificates automatically, or short lived database credentials.

API

As described before, Vault provides a HTTP Restful API that allows applications to consume secrets programmatically. There are a large number of client libraries and in-language abstractions that allow for simpler programming.

This is traditionally the most secure pattern to retrieve secrets from Vault, as these are generally stored as a variable in memory, and get cleaned up when the application is not using them.

This pattern is considerably invasive, as it requires modifying the application to retrieve secrets from Vault (generally upon initialization, or consuming an external service).

Non-invasive patterns

HashiCorp provides two applications that provide workflows to consume credentials without modifying the application, using the most common pattern for secrets consumption, via a configuration file or environment variables.

These applications either render a configuration file template[16] interpolating secrets, or pass environment variables[17] with values obtained from Vault.

[15] https://learn.hashicorp.com/vault/identity-access-management/iam-secure-intro
[16] https://github.com/hashicorp/consul-template
[17] https://github.com/hashicorp/envconsul

Response wrapping

Where a third party software generally provisions credentials, this system can request wrapped responses, in which the secrets are ultimately unwrapped in the system consuming the secret, without the need to disclose credentials to the automation system that is configuring the application.

Encryption as a Service

Vault provides cryptographic services, where consumers can simply encrypt / decrypt information by virtue of an API call, and key lifecycle and management is generally managed by an external team (often assisted by automation).

Each key generated has separate API paths for management, and each service action (encrypt / decrypt / sign /verify), allowing policy to be set at a very granular level, aligning to roles existing in the organization. As an example, while the security department may have the following permissions over a particular transit mount:

```
## Crypto officers
# Create key material, non deletable, non exportable in unencrypted fashion, only aes-256 or
rsa-4096
path "/transit/keys/*" {
  capabilities = ["create”, “update"]
    allowed_parameters = {
      "allow_plaintext_backup" = [“false”]
      "type" = ["aes256-gcm96", "rsa-4096"]
      “convergent_encryption” = []
      “derived” = []
    }
}

# List keys
path "/transit/keys" {
  capabilities = ["list"]
}

# Rotate Key
path "/transit/keys/*/rotate" {
  capabilities = ["create"]
}
```

Whereas a consumer would just have access to the service:

```
## Consumers
# Encrypt information
path "/transit/encrypt/keyname" {
  capabilities = ["create”]
}

# Decrypt information
path "/transit/decrypt/keyname" {
  capabilities = ["create"]
}

# Rewrap information
path "/transit/rewrap/keyname" {
  capabilities = ["create"]
}
```

Day N

Storage Key Rotation

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    01 May 17 10:30 UTC
```

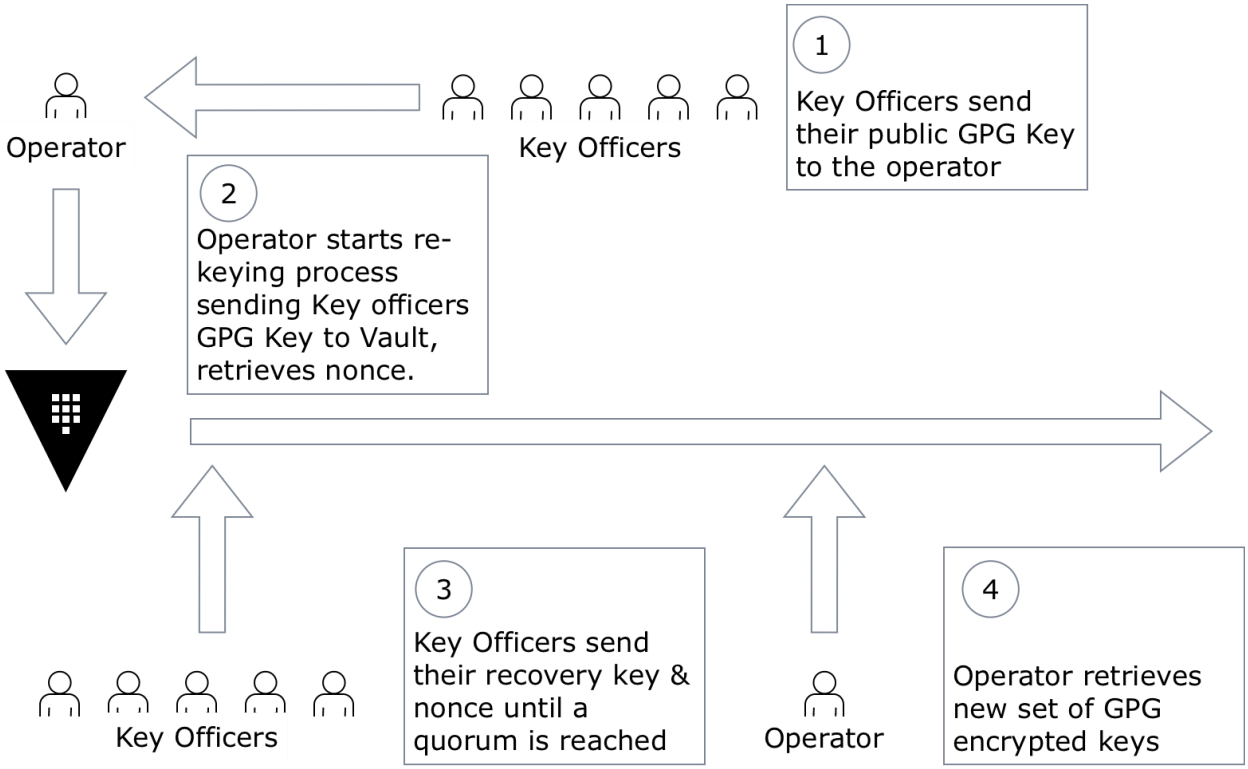
This requires the right privileges as set on the policy. From the point in time of rotating the key every new secret gets encrypted with the new key. This is a fairly straightforward process that most organizations carry out every six months, unless there is a compromise.

Master Key Rotation

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's time bound. The way this procedure is carried out would be 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 proceed 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 of time. Traditionally in organizations there is a level of collaboration with a Human Resources department, or alternatively these procedures already exists for organizations using HSMs, and can be leveraged for Vault. An illustration of the procedure can be found below.



This procedure is generally carried out by organizations yearly, unless there is a compromise.

Seal Key Rotation

This procedure varies from seal to seal.

For a PKCS#11 seal, traditionally an operator would simply change the key_label and hmac_key_label parameters. Upon detecting that the label has changed, and it does not match the label used to encrypt the Master Key, Vault will simply re-wrap it:

```
2018-09-05T11:56:22.892Z [DEBUG] cluster listener addresses synthesized:
cluster_addresses=[127.0.0.1:8201]
2018-09-05T11:56:22.892Z [INFO ] core: stored unseal keys supported, attempting fetch
2018-09-05T11:56:22.892Z [DEBUG] sealwrap: unwrapping entry: key=core/hsm/barrier-unseal-keys
2018-09-05T11:56:22.893Z [DEBUG] sealwrap: unwrapping entry: key=core/keyring
2018-09-05T11:56:22.895Z [INFO ] core: vault is unsealed
2018-09-05T11:56:22.895Z [INFO ] core: post-unseal setup starting
2018-09-05T11:56:22.895Z [DEBUG] core: clearing forwarding clients
2018-09-05T11:56:22.895Z [DEBUG] core: done clearing forwarding clients
2018-09-05T11:56:22.895Z [DEBUG] sealwrap: checking upgrades
2018-09-05T11:56:22.895Z [DEBUG] sealwrap: unwrapping entry: key=core/hsm/barrier-unseal-keys
2018-09-05T11:56:22.896Z [TRACE] sealwrap: wrapping entry: key=core/hsm/barrier-unseal-keys
2018-09-05T11:56:22.898Z [DEBUG] sealwrap: unwrapping entry: key=core/keyring
2018-09-05T11:56:22.899Z [TRACE] sealwrap: wrapping entry: key=core/keyring
2018-09-05T11:56:22.900Z [DEBUG] sealwrap: upgrade completed successfully
```

—

```
2018-09-05T11:56:22.900Z [DEBUG] sealwrap: unwrapping entry: key=core/wrapping/jwtkey
2018-09-05T11:56:22.902Z [TRACE] sealwrap: wrapping entry: key=core/wrapping/jwtkey
```

The existence of the key can be verified by using HSM tooling or the pkcs11-list command:

```
$ pkcs11-list
Enter Pin:
object[0]: handle 2 class 4 label[11] 'vault_key_2' id[10] 0x3130323636303831...
object[2]: handle 4 class 4 label[16] 'vault_hmac_key_1' id[9] 0x3635313134313231...
object[3]: handle 5 class 4 label[11] 'vault_key_1' id[10] 0x3139303538303233...
```

DR Promotion

The DR promotion procedure would be carried out to continue service upon a catastrophic failure that would leave the primary cluster unusable. This procedure shouldn't be carried out if the Primary cluster is in service as it may have unintended consequences.

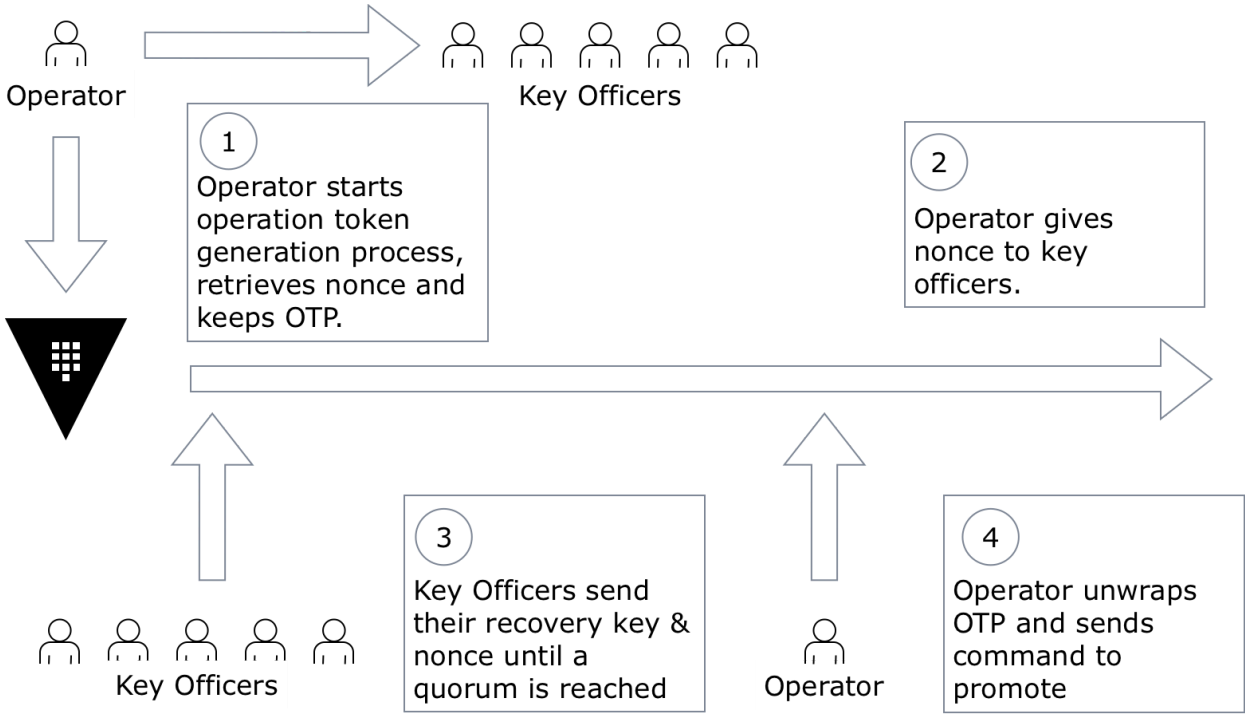
Possible causes to trigger a DR promotion would include, but not be limited to:

- Primary Datacenter failure
- Data corruption on Primary Cluster
- Seal key loss
- Seal key tampering

This procedure includes a Vault operator and the Key Officers, and is similar to an unsealing or key rotation, in the sense that a quorum of officers need to approve the operation by virtue of submitting their key share.

1. The operator submits a request for promoting the Secondary cluster to Primary. This is one of the only endpoints available to a DR Cluster, which is not operational under normal parameters. It will retrieve a nonce for the operation.
2. The Key Officers would proceed to submit their key shard.
3. Once a quorum of Key Officers have been reached, the operator can retrieve an operation key.
4. This operation key is one time use, and the operator will use it to start the promotion process.

The procedure is illustrated in the diagram below.



Policy Maintenance Patterns

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

Policy templates are also used as a way to reduce the amount 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 via 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 regular operation in Vault.

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

- What is going to be the consumer involvement into the process. Will be the consumer be responsible from 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 secure the token ahead. The handover point needs to be agreed 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 supposed to create a role for the new application.
- If using static secrets, is the consumer expected to manually store the secret into Vault, or will it be refreshed by a process automatically.

Most commonly, as described previously, organizations start with auditing static password 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