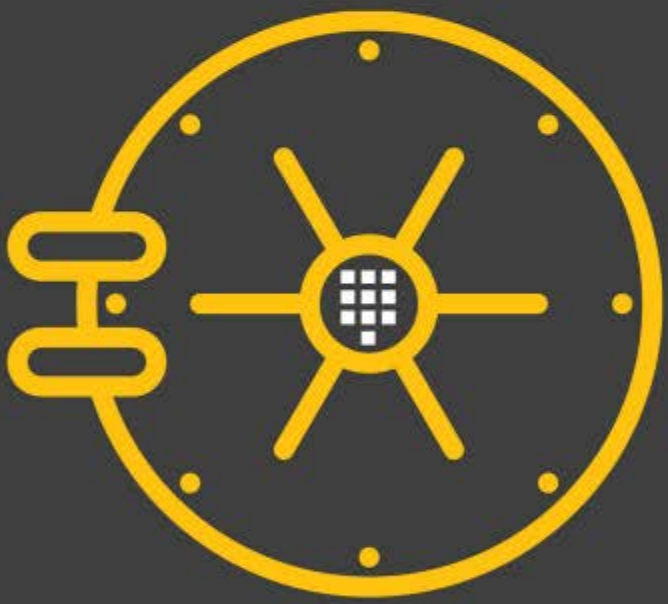
# RUNNING HASHICORP VAULT IN PRODUCTION



Foreword by ARMON DADGAR

DAN MCTEER
BRYAN KRAUSEN

# Running HashiCorp Vault in Production

Dan McTeer
Bryan Krausen

**FOREWORD BY** Armon Dadgar

**First Edition:** October 2020

**Copy Editor:** Evan Turro

# *Table of Contents*

# Preface

We wrote a book!

After many years of working with Vault in the real-world and learning it the hard way, we took the leap. We had both worked on large scale Vault projects and wanted to share this knowledge we've gained with the rest of the world. We decided that our combined experience could help folks looking to deploy Vault in their environment.

When we began outlining the content of the book, a common theme kept coming up – we wanted to use our experience with Vault to teach others how to use it while providing real-world examples, including customer examples and use cases. We wanted to include information to help others avoid the same mistakes we made and provide configuration files and code to help others get Vault up and running quickly. But we also wanted to make sure readers understood the processes involved with defining use cases, considerations when designing a Vault environment, and how to use some of the most popular components we often see with Vault customers.

In addition to teaching readers about Vault 1.5, we also found there was a gap in current literature about a solution that is gaining tons of market share. Vault solves many complex problems in a world full of containerization, multi-cloud, and the ongoing migration to microservices application architecture. However, there is no single place that one can learn about how to plan, design, implement, and manage the Vault service.

As we wrote the book, we had two audiences in mind: the reader who will use the book as a reference guide and the reader who will read the book front to back. Pleasing both audiences meant repeating ourselves enough to be helpful for reference readers without boring the front-to-back readers with repetition.

We hope this book inspires you to accomplish extraordinary things with Vault.

Thanks for reading!

***Bryan Krausen***
***Dan McTeer***

# *Acknowledgements*

*A huge thank you to my family for allowing me to spend the time writing this book, recording HashiCorp training courses, and just putting up with me in general. Your support not only allowed me to do this but also pushed me through until the end!*

*Also, thank you to everyone who watches my training courses, the customers who entrust me to design/implement solutions for them, and of course, my co-author for dealing with me throughout this whole thing - it probably was not easy!*

Bryan Krausen
@btkrausen

*Thank you to my loving wife, Abby, who has always pushed me to be a better person, to achieve greater things, and who is always there to help me get back up when I fall. Thank you to my kids, Matthew, Samuel, Sean, and Mason, who always keep life exciting and memorable (mostly in a good way).*

*Finally, thank you to Justin Barlow, Max Robison, Marius Barbulescu, Chandler Allphin, Scott Webster, Radu Puscas, Nico Grandori, and Tyler Jacobsen – the best damn Vault team (and friends) on the planet. Much of the content in this book came as a result of our dedicated work and collective learning.*

*To those who read this book: Thank you! I hope that at least some of this information can help make your life easier.*

Dan McTeer
@danmcteer

# About the Authors

## *Bryan Krausen*

Bryan Krausen is a consultant with over 20 years in enterprise IT, specializing in Amazon Web Services and HashiCorp tools. He uses his expertise to help customers with a variety of design, implementation, and integration decisions. He facilitates customer workshops to understand business and technical requirements and designs solutions to allow businesses to take advantage of public cloud and HashiCorp tools.

Bryan often speaks at community events, such as HashiConf (2019, 2020), HashiTalks (2019), and many HashiCorp User Groups across the eastern US. Bryan is also the first person to earn the HashiCorp Vault Expert partner certification.

## *Dan McTeer*

With an education in Information Security and over two decades of technical operations experience, Dan is passionate about DevSecOps and simplifying the implementation of security in technical operations processes. He recently moved over to HashiCorp from Adobe, where he managed a global SRE team responsible for the technical operations of many of Adobe's security-related services.

Dan is married to an amazing and loving wife who is always pushing him to be a better human being. In his spare time, you can find Dan cheering on his crazy kids, traveling the world, flying drones, riding his mountain bike, or just gazing at the stars.

# Foreword

It is often said that necessity is the mother of invention. If that is the case, there are any number of candidates for the father, including frustration and fear. Certainly, all the above were major considerations for why Vault was conceived and built.

In 2012, the public cloud was being rapidly adopted by smaller companies, and early signs were there for larger enterprises, but tooling for operators persisted as a big gap. When Mitchell and I founded HashiCorp, we wanted to provide tools for fellow operators of infrastructure to make it simple to operate a modern cloud infrastructure. We imagined a portfolio of products which would each solve a piece of the problem, and together would provide a holistic solution.

When we started building the initial products, they were roughly built in the same order that you would need them if you progressed through a software delivery lifecycle. Vagrant was first, allowing users to quickly jump into a development environment, write code, and test with parity to production environments. Packer was next, allowing users to build machine images (and later container images) that were suitable for production environments. Serf and Consul both followed, trying to solve the networking challenges of a distributed application, namely, how to discover and route between many services. Terraform provided a simple way to declare infrastructure as code and offload the day one creation and day two convergence. Later, Nomad would provide an application deployment tool that could handle modern container applications along with static binaries, JARs, VMs, and other packaging formats.

The one piece we did not anticipate early on was Vault. Given the long history of privileged credentials, we made the assumption that great solutions already existed, and we had just failed to discover them. As HashiCorp grew, we had an increasing customer base who trusted us with TLS certificates, usernames and passwords, API keys, and more. There was a certain *necessity* to us solving the challenge of handling these securely, and certainly a fear over the risk of a breach was on our minds. As we evaluated various solutions, this quickly turned into frustration. Most approaches had fundamental security issues, were difficult to automate, or seemed overly complex.

These forces culminated in driving us towards building Vault as a first-class product. We often talk about the Tao of HashiCorp, which is the shared ethos behind the design of all of our products. This leads us to emphasize certain things, like workflows over technologies, simplicity and composability, automation, and high availability. Many of our products also have deep roots in academic research and can point to specific research papers that guided our thinking or approach.

As we started to think about Vault, we tried first to get to a crisp definition of the problem and workflows involved. At the simplest, it is about allowing access to systems and services but only for authorized users and services. While this sounds simple enough, things get complex as you pull the thread of each requirement. As an example, humans and machines should not necessarily authenticate the same way; otherwise, you end up with a bad experience for human operators or, conversely, a strange design for API based integrations. Workflows around password or credential rotation also tend to be manual and onerous, so we wanted to find elegant solutions to avoid that complexity.

Very early in the design process, we started doing research on other systems. It is no accident that Vault has many similarities to Kerberos in approach. Although originally designed in the 1980s, Kerberos addressed many challenges around common means of authentication and brokering access to a wide range of systems. The weakness of Kerberos was the complexity of the system and the difficulty of integration. Kerberos required every system to integrate using the GSS-API, which was a difficult task that few modern tools implemented. Probably the second weakness of Kerberos is the terminology can be quite confusing when getting started and presented a barrier for new users.

We borrowed the general approach of Kerberos but made a major change of inverting the integration model. Rather than have every system speak a common protocol and integrate with a central service, we would build a robust plugin system and allow Vault to speak the native API of each system we are managing. Today Vault is able to manage credentials on a wide range of systems, including most common RDBMS', NoSQL systems, message queues, public cloud providers, Active Directory, LDAP, and much more. This is possible because of the plugin system around secret engines, where a Vault secrets engine is able to speak the native API of those systems without any code changes to them.

With a pluggable notion of secret engines, Vault could manage almost anything that has an API and credentials. Given our goal was to support a wide range of clients, both humans and machines, Vault also has pluggable authentication methods. These enable human operators to authenticate via traditional usernames and passwords or modern Single Sign-On (SSO) providers such as Active Directory or Okta. Application or machine workloads could authenticate using certificates, bearer tokens, native identity provided by cloud vendors like IAM, or platform identity provided by Kubernetes, Nomad, or Cloud Foundry.

What started to emerge was that Vault would provide a consistent workflow for authentication of clients, a consistent way to define authorizations, and a consistent API for getting credentials and performing operations. This consistent workflow and API would make it simple to integrate with Vault, while internally, the plugins would support a huge ecosystem of existing products and solutions.

A great property of Kerberos is that access is granted in a way that is ephemeral, meaning you are never granted a long-lived credential but only short-term access that can be renewed as needed. This was possible because all systems that integrated with Kerberos had to respect and enforce the ticketing mechanism. We felt strongly that this notion of ephemeral, time-bounded access was crucial to preserve. It seems like weekly there is a major breach announced caused by credentials that are leaked or stolen that are valid for months or years.

We struggled with how to bring this functionality to Vault, given that most of the systems we are integrating with have no similar concept. Ultimately, the idea of a “dynamic secret” was born. Vault would leverage the secret engine to create an entirely dynamic username and password, or API token depending on the system. However, it would restrict the permissions based on a static policy. The lifecycle of that credential would be managed by Vault, meaning it is created on demand when a client requests access and automatically destroyed after a pre-configured time to live.

This idea of a dynamic secret, or a credential that was only “leased” to a client rather than permanently provided, allowed us to provide an elegant solution to several problems. Day one, an application or a user needs certain privileged credentials to perform their function. However, day two and beyond, we have to worry about the rotation of credentials, revocation of access, and offboarding. Those problems become quite challenging if clients are provided long-lived credentials because there is no way to force a client to “forget” and no guarantee they do not keep a copy. However, with a dynamic secret, the credential is

automatically destroyed at the end of its time to live. It does not matter if a client is misbehaving or stores a copy because the target system will reject their access.

The short-lived nature of a lease forces clients to periodically come back to Vault to renew a lease or fetch a new dynamic secret. This periodic interaction allows Vault to then automate credential rotation very easily and provides an elegant workflow for the day two challenges, in addition to the day one initial distribution.

Once we had this core architecture in place, it became clear that the flexibility of Vault enabled it to solve other security challenges. A common challenge we faced internally was encrypting customer data and key management for applications that needed access. We quickly discovered that we could implement a secret engine (today called the “transit” engine) which would hold the encryption keys within Vault, but expose an API to perform high-level operations such as encryption, decryption, signing and verifying transactions, and more. The Vault client could just invoke the API with the high-level operation they needed and defer the key management and cryptography to Vault. This made life simpler for our developers and more sane for our security teams.

In the last five years, since we first built Vault, it has grown and matured in many ways. While all the initial capabilities and design are still there, it has grown to support a huge number of authentication methods and secret engines. It has added richer automation capabilities. It has become much easier to use with a great developer CLI and a web-based user interface. There are broad ecosystem integrations for Vault from configuration management systems, to application platforms, to low-level data management and encryption solutions.

When I am asked about what makes Vault such a successful product, the one-word summary is simplicity. Those who have used Vault may laugh, knowing it is not exactly a simple tool, and the fact there are multiple books on it is probably a fair testament to that. However, when you consider the complexity and breadth of the problems Vault is attempting to solve, it succeeds in packaging that into a tool that is (relatively) simple to understand, deploy, integrate with, and use.

The HashiCorp ethos is very much focused on workflow over technology. It is that focus on the challenges our users face and trying to solve those in as elegant a fashion as possible that leads to the design of the products. I believe that has set Vault apart and allowed it to be successful. I hope you find the book useful on your journey to understanding the product more deeply and using it to solve complex problems with a solution that is reliable, secure, and scalable.

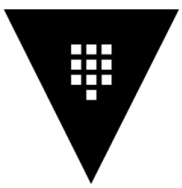
***Armon Dadgar, Co-Founder and CTO***
***HashiCorp***

# 1 Introduction to HashiCorp Vault

***This Section Covers:***

- How Vault Works
- Interacting with Vault

The first chapter of this book focuses on what Vault is and how it can be used. For those unfamiliar with Vault, this chapter will provide a high-level overview of HashiCorp Vault and the process of secret retrieval. One of the most powerful aspects of Vault is that it can be consumed in many different ways, so this chapter closes by discussing a variety of different ways to interact with Vault.



## *How Vault Works*

Imagine that someone is taking a vacation to a beautiful and exotic location. Their trip would begin with their arrival at the hotel to check in to their room. At the front desk, the receptionist asks for the person to prove their identity by providing credentials. The receptionist would then check their credentials against the hotel record on file and determine that they are whom they say they are. After verifying the guest's identity, the receptionist would issue the guest a room key. The specific level of access granted to the guest would depend on the guest's relationship with the hotel. Where a first-time guest might only receive access to their room and the hotel gym, a returning guest may also receive access to the guest lounge for their loyalty.

The workflow for a Vault client communicating with HashiCorp Vault is similar to the check-in process at the hotel. Like Vault, a hotel has many entrances for a guest to access the lobby and guest room. When a Vault client needs access to Vault, it does so through a variety of interfaces, including a robust **API**, a **CLI**, and a **UI**. Protecting Vault is done through a **cryptographic barrier**, similar to how a hotel builds walls and doors to protect guest rooms and amenities. This cryptographic barrier is responsible for encrypting all information stored by Vault.

A Vault client's objective is to access data from a **secrets engine** the same way our guest's objective is to access their room. However, before a Vault client is permitted access to a secret, it must authenticate with Vault to prove its identity, like our guest who had to check-in before accessing their room. Just as the front desk checked the customer's ID, Vault validates the credentials of the user through a configured **auth method** to ensure those credentials are valid. Upon successful validation, Vault issues a **token** to the client based on a defined **policy** for that user. Similar to the guest's room key that allows them to access various places in the hotel, this token (based on the defined policy) permits the Vault client to access different endpoints within Vault, called **paths**. The token is valid for a set amount of time using a **time-to-live (TTL)** and is no longer valid after expiring, just as the guest's key will no longer work when their stay has ended.

While these permitted services might not be a luxurious gym or cushy guest lounge, they may include retrieving static secrets, generating new credentials, or encrypting data.

## The Basic Components

In order to understand how Vault works, it is vital to understand the necessary pieces of Vault. These pieces consist of the user or machine authentication process, the secrets engines available once a user or machine is authenticated, the policies that dictate access to those secrets engines, and the interfaces available to interact with Vault.

### *Paths*

Paths are the basic building blocks of Vault. Like the hallways of a hotel, every endpoint in Vault is accessible through a path. These paths provide access to everything within Vault, including secrets engines, policies, auth methods, static secret values, and system health information. After a component is enabled in Vault, all interactions with that component are performed using the associated path. Even Vault's administrative functions are configured using a path.

### *Secrets Engines*

Secrets engines provide Vault's core functionality, and without secrets engines, there is no point in deploying Vault. The specific function of each secrets engine, however, can vary. Some secrets engines store static secret data, while other secrets engines can generate a set of short-lived, dynamic credentials. Some can even encrypt plaintext data in transit. All other Vault components can be seen as supporting components to secrets engines.

### *Auth Methods*

Auth methods are responsible for assessing identity and assigning a set of policies to a user or machine. Like the front desk at a hotel, auth methods validate authentication requests through the configured identity provider to ensure the credentials are valid before granting access to services. Examples of auth methods include Active Directory, LDAP, GitHub, Kubernetes, Okta, and identity management services on the major cloud providers.

### *Tokens*

Tokens are the core method of authentication within Vault. Vault can be configured to use tokens directly as an authentication mechanism, or an auth method can be used to generate dynamic tokens based on external identities. Regardless of how Vault clients authenticate to Vault, a token will be used for all subsequent requests.

### *Policies*

Policies determine the level of access an entity has to a particular path or the data contained therein once the entity has authenticated. Permissions defined inside of these policies follow the typical "CRUD" model of access: **C**reate, **R**ead, **U**pdate, **D**elete. These permissions or "capabilities" are applied to a particular path and associated with specific Vault clients or applied service-wide. Certain "parameters" can be associated with the policies to tighten security controls around specific actions further. In summary, while auth methods handle authentication to Vault, policies control *authorization* to Vault components once a client has successfully authenticated.

## The Vault Service Process

There are many aspects to preparing Vault to handle client requests on a day-to-day basis. Generally, there are three phases to prepare a production-ready Vault environment: the configuration, the client interaction, and post-client events. Each phase is uniquely essential to building a working Vault environment.

### *Phase 1 - Preparation*

The first phase involves the actions performed by an administrator to enable user interaction. Before a Vault client can retrieve a database credential from Vault, for example, there are configuration items that must be in place:

- The database ***secrets engine*** must be configured with an administrative credential capable of creating dynamic user accounts with appropriate permissions on the target database.
- An ***auth method*** must be configured to permit access to the entity requesting access to the database.
- A Vault ***policy*** that grants permission to the database secrets engine needs to be created and attached to the entity.

### *Phase 2 - User Interaction*

The second phase of the Vault service process involves interaction from the user. When a client sends a request to Vault, the request is initiated using TLS to verify the identity of the

Vault service and establish secure communication with it. The basic workflow of client interaction with Vault includes:

1. The client sends an authentication request to Vault, specifying the auth method and credentials to be used.
2. Vault forwards those credentials to the appropriate authentication backend for verification.
3. Vault receives approval from the appropriate authentication backend and returns an access token to the requestor based on the policies associated with the requestor.
4. The Vault client uses the access token to issue a read request to the associated path to generate database credentials.
5. Vault validates the token and the associated policy to determine whether access to a database credential should be granted.
6. If the token is permitted to access the requested path, Vault will use pre-configured database credentials to generate temporary database credentials based on the policy associated with the requestor. The database credentials are returned to the requestor.

### *Phase 3 - Cleanup*

Once user interaction is complete, Vault needs to clean up the token. When a token or other credential is provisioned by Vault, it is associated with a TTL or "lease." This lease may be time-based, such as 24 hours, or defined by a set number of uses. After the lease expires, the tokens are revoked, and the associated credentials are removed automatically by Vault.

Figure 1-1 illustrates the database example, where a Vault client needs access to database credentials to read data inside the database. Phase 1 of this example includes the configuration of the LDAP and Cloud auth method, the policy creation, and the database secrets engine configuration. When a Vault client sends an authentication request using LDAP, for example, the workflow in Phase 2 is executed. The client obtains a token and requests database credentials.  Once the client interaction is completed, the token will expire, and the database credentials will be revoked.

Note that the intent and philosophy of Vault is to build security into automated processes that utilize temporary credentials - whether for a specific duration or one-time use. The ultimate goal is to migrate from static credentials and identities that require periodic manual

rotation. Keep in mind, though, that this workflow can still be leveraged for static secrets as well.

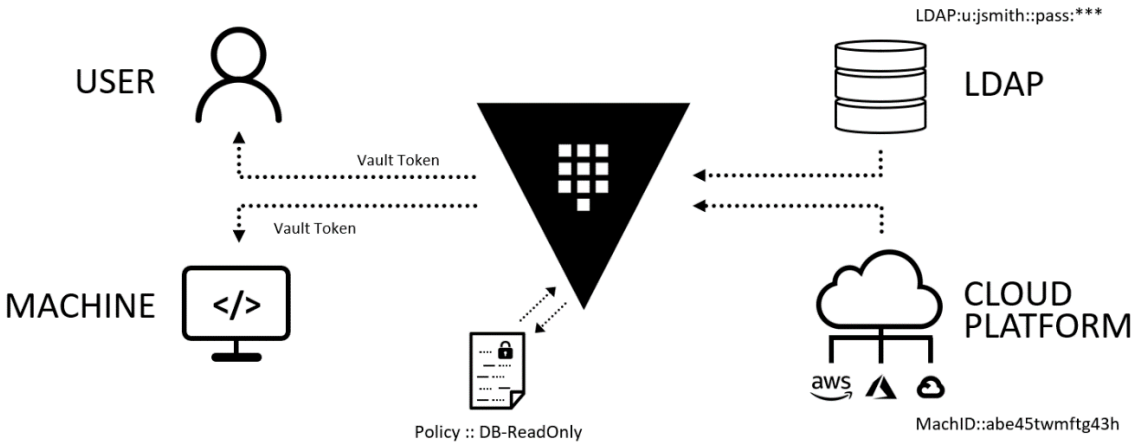



Figure 1-1

# *Interacting with Vault*

It is critical for developers and Vault operators to understand the details of the user interaction process as they add integrations. The previous section introduced this process at a high-level, but additional context on the day-to-day use of Vault will significantly benefit the consumers of the Vault service.

## The Vault API

Every interaction with Vault happens through the API, regardless of the entry point. Even when clients interact with the CLI or web interface, the actions requested are performed using the API. While the CLI and web interface are friendlier to end users, they do not support all the actions that can be made by using the API directly.

The API is divided into three main sections:

***Secrets Engines***
These provide the core functionality of Vault. Secrets engines can maintain static values, generate dynamic credentials, or perform cryptographic functions.

***Auth Methods***
These are responsible for validating authentication requests from Vault clients and authorizing the user to perform actions in Vault. HashiCorp Vault can use an external identity provider for authentication or provide internal identity management for granting access to secrets.

***System Backend***
This endpoint is intended primarily for service administrators to configure the components and features within Vault. The system backend can also be used by external services to monitor Vault, such as a third-party monitoring solution or a load balancer health check.

## Vault Interfaces

There are three avenues for Vault interaction, all of which occur through the Vault API.

### *Using the API*

Programmatic interaction with Vault occurs when an application or service calls the Vault API. This method should be the most common form of interaction with Vault. Interaction using the API is what Vault was designed for – a completely hands-off, automated secrets management experience. Automating complex tasks removes the human element, reduces organizational risk, and improves security.

### *Using the CLI*

Before configuring traditional application integration with the API, the CLI is often used to configure Vault. The CLI simplifies the administration of the Vault service when performing initial Vault tasks, such as migrating secrets to Vault. Once the initial configuration has concluded, most of the day-to-day interactions can move to a programmatic function. Almost all the accessible API paths are available in the CLI, with few exceptions.

The CLI is the ideal method for Vault operators or users who might be more comfortable using a command line to perform manual interactions with Vault instead of clicking around in the web interface. Interaction through the CLI occurs through the same binary that is used to operate the Vault service. A user may issue CLI commands directly on a Vault node, or the binary may be downloaded on a local workstation to streamline access to Vault.

### *Using the Web UI*

The Vault UI is accessible using modern web browsers. The UI can be accessed using the DNS name or hostname of the cluster and the configured port of the listener.

For some users, the web interface is the most appropriate choice for accessing Vault. Although Vault was not initially designed with a user interface in mind, the web UI has quickly become a focus of development. Recent Vault releases have included significant improvements to the UI, adding new features and capabilities. The most common tasks a user or administrator might need to perform can be executed using the Vault web UI.

## General Consumer Interaction

General consumer interactions consist of the basic human actions executed against the Vault service. This type of interaction is often how users first approach Vault when the

service is initially deployed. Under this category of interaction, a human user would authenticate with Vault, receive an access token, and use the token to manage Vault or read a secret.

Many of the standard authentication providers used today, such as directory services, OIDC solutions, or public cloud identity services, can be leveraged to validate human users.

***Example:*** Jane Smith needs to retrieve the latest value of the root password for one of the Linux hosts she manages. Jane uses the Vault CLI to log in to Vault using her LDAP username and password. Once authenticated, Vault returns an access token to Jane. Jane then uses that access token to retrieve the value of the desired secret from a specific path in Vault. This process is outlined in Figure 1-2.

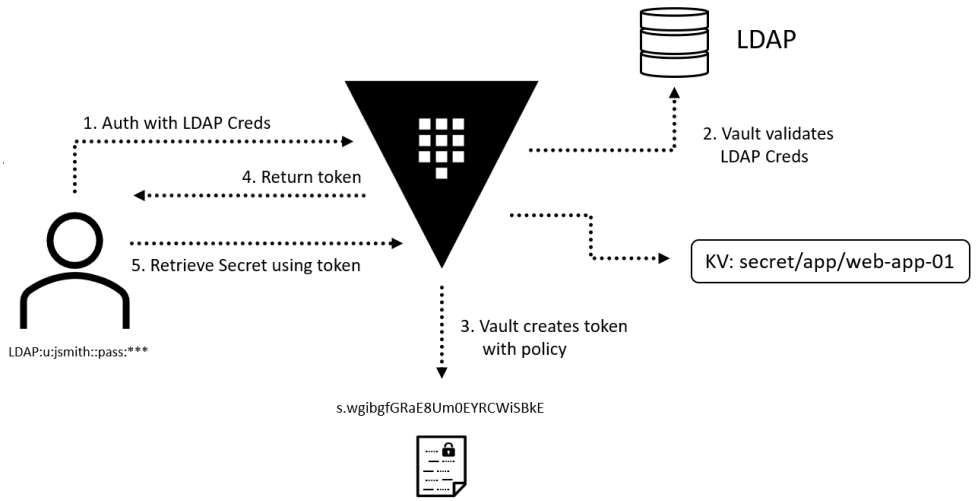



Figure 1-2

## Administrative Interaction

Like consumer interactions, a human entity usually performs administrative interactions during both the deployment and initial management phases of the Vault service. The administrative interactions also tend to shift to programmatic interactions as the service matures and grows. However, administrative interactions shift toward management of the Vault service rather than the management of secrets.

Vault administrators commonly use the same auth methods that are leveraged by other human users. However, it is common to use Endpoint Governing Policies (EGPs) to ensure the second factor of authentication is used for privileged users. For instance, all Vault users need to provide an LDAP username and password. However, Vault administrators must also use a second factor of authentication, such as an Okta Verify push notification. This strategy increases the security of privileged user access.

When using an EGP to configure the second factor of authentication, the policy can be written to trigger the second factor based on the path requested. For instance, configuring the Okta push to trigger at *auth/ldap* would initiate an Okta push for all users authenticating through LDAP. However, configuring the Okta to push to trigger at *auth/ldap/users/jdoe* would initiate an Okta push only for Jon Doe.

In this scenario, both the LDAP auth method and Okta MFA provider must be configured in Vault before the EGP can be used for triggering the second factor of authentication.

***Example:*** Jon Doe needs to create a namespace in Vault for a new team onboarding to Vault. Jon uses the Vault CLI to log in to Vault using his LDAP username and password. Jon receives an Okta Verify to verify his identity. Once authenticated, Jon receives an access token from Vault with administrative privileges. Jon uses his token to create and configure a new namespace in Vault. This process is outlined in Figure 1-3.

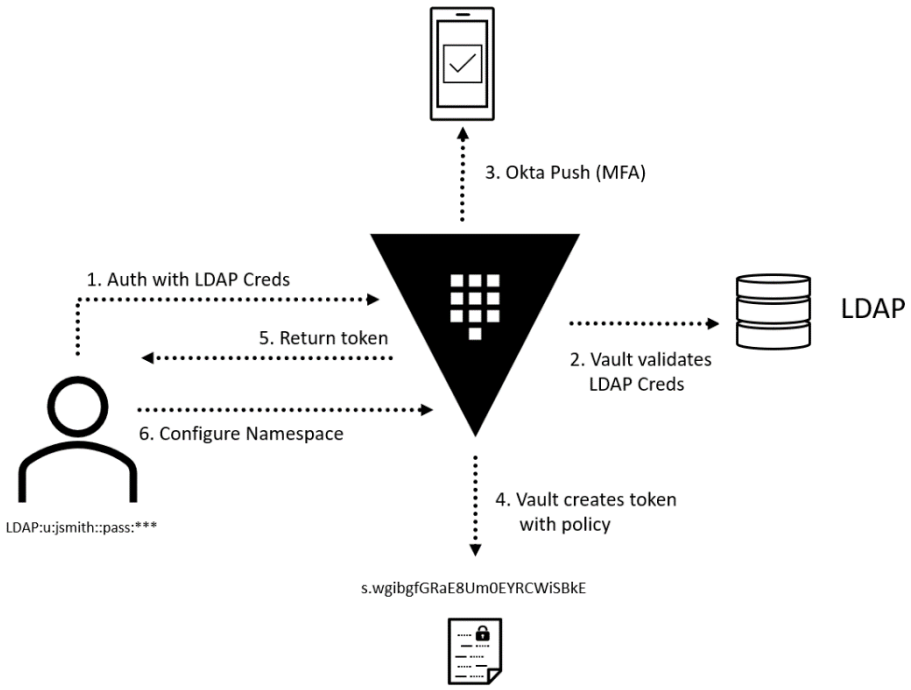



Figure 1-3

## Security and Compliance Interaction

While Vault can increase the overall security of integrated applications, security and compliance personnel must ensure the Vault service is secure. The security and compliance team may interact with Vault in a variety of ways. The first type of interaction is based on a need for secrets management and aligns with general consumer interactions. Typically, security and compliance personnel have secrets that need to be stored and managed. The security and compliance teams also need to ensure that the Vault service is being consumed in a secure and compliant manner along validating the security of the Vault service. This would include processes such as penetration testing or static code analysis.

To ensure Vault is running and being consumed securely, the security and compliance teams will frequently use the Vault audit data to monitor operations using log aggregation and analytics tools, such as Splunk or an ELK stack.

***Example:*** A Vault user authenticates with Vault and receives an access token. The access token is used in an attempt to retrieve a secret that the user does not have access to read. The user's request is denied, and a log entry for the event is generated through an audit device and ingested into Splunk. A notable security event is sent to the security operations center (SOC) for investigation. This process is outlined in Figure 1-4.

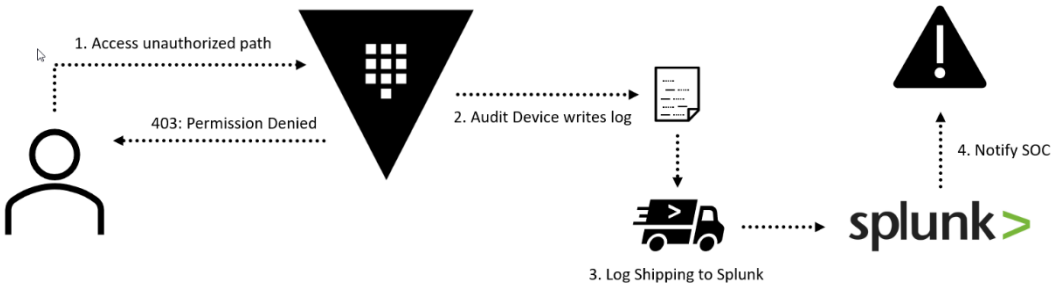



Figure 1-4

## Programmatic Interaction

One of Vault's most compelling features is its robust API, especially in organizations adopting a DevOps culture or migrating applications to the public cloud. The goal is to have the majority of interactions take place through automation using the API directly. This form of interaction requires the entity (usually an application or host) to authenticate with Vault, receive a token, and use it to retrieve a secret.

One of the most common auth methods used for this machine-to-machine interaction is the AppRole auth method. This auth method uses a RoleID (username) and a SecretID (password) to authenticate a non-human entity with Vault. As a result of authentication, the client receives a token in response that can be used to retrieve a secret.

***Example:*** A root password on a group of Linux hosts needs to be rotated. These hosts are using SaltStack for configuration management. On the Salt master, a process is triggered to rotate the root password on all Salt minions. The Salt master uses a RoleID and a SecretID, configured in environment variables, to authenticate with Vault and retrieve a short-lived access token. The Salt master then uses this token to read a secret value at a specified path in Vault and sends this value to all attached minions. Once received, each minion

updates the local root password with the specified secret value. This process is outlined in Figure 1-5.

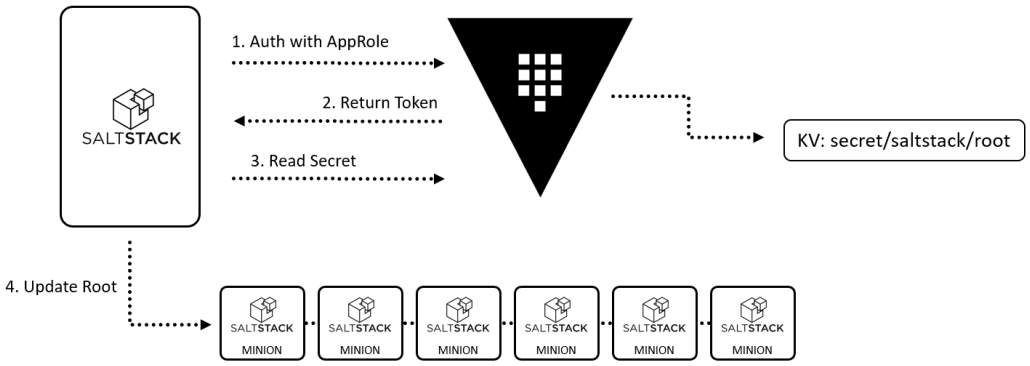



Figure 1-5

# 2 Planning A Vault Deployment

***This Section Covers:***

- Gathering Requirements
- Service Level Objectives and Indicators
- Designing the Vault Architecture
- Finalizing the Deployment Plan
- Choosing the Support Team

This chapter is all about Vault design. From gathering business and technical requirements to selecting where Vault will be provisioned, this chapter will describe a practical planning phase. With a proper Vault design, organizations can accelerate the deployment and adoption of Vault.

# *Gathering Requirements*

As with the introduction of any platform, organizations need to understand both the business requirements and technical challenges that Vault is intended to solve. Vault includes many features and integrations and can quickly become the focal point for services that organizations depend on to run applications successfully and securely. Planning a Vault deployment is not as simple as reading the technical documentation and following a step-by-step guide; it requires conversations with internal teams such as management, development, security, operations, automation, and even the network/security operations centers. Defining these collections of both functional and non-functional requirements will help ensure a successful project.

## Business Requirements

Requirement gathering happens in a layered approach, and each layer directly impacts the one beneath it. Commonly, requirements gathering starts with "the big picture" and works its way down to team-specific requirements. Therefore, the first functional requirements that need to be understood are the general business requirements. Business requirements are critical to understanding because they define the main objective of the overall project. Every organization is different, and each business has requirements that must be met due to events such as planning, the anticipation of new business or acquisitions, or the introduction of new offerings to its customers. Goals could include significant milestones such as obtaining PCI-DSS certification or passing a HIPAA audit down to internal goals such as improving security or consolidation of tools. Regardless of the goal, each should be taken into consideration when gathering requirements for a Vault implementation.

## Technical Requirements

Once the business requirements are fully understood, technical requirements are gathered to help drive the design of the overall Vault implementation. As the name implies, technical requirements are generally dictated by existing or inbound technologies and services used by the business, such as cloud platforms, container scheduling, or the workloads that interact with Vault. The technical requirements can also be driven by a service-level agreement (SLA), regional or international availability, or established security policies.

Technical requirements that may need to be considered include:

- Vault must be highly available and provide an SLA of 99.99%
- Database secrets must not be valid for longer than 4 hours
- Vault services must be available for both on-premises and cloud-based workloads

While gathering technical requirements, it is essential to discuss any constraints that may affect the design and implementation. An example of a constraint may be "Vault must be deployed in AWS," which may drastically change the Vault design instead of deploying Vault on-premises. Constraints can come in many different forms, including budgetary, technical, or time constraints.

In addition to constraints, risks must be identified and documented to ensure all parties understand and agree to these risks. The potential increase in the cost of the underlying infrastructure in a public cloud is an excellent example of a risk that should be considered. It is essential to verify any assumptions made before or during the requirement gathering process to ensure there are no "unknowns" when moving into the design phase.

Technical requirements that might need to be answered for a Vault design may include:

- Which version of Vault will be implemented, Enterprise or Open-Source?
- Will Vault be deployed on a public cloud, on-premises, or both?
- Will Vault be provisioned using Terraform, scripts, or manually?
- Will Vault need to be highly available and deployed in a cluster?
- Does Vault need to be available in multiple data centers or cloud regions?
- What storage backend will Vault use to store its data?
- What secrets engines will be initially consumed?
- What performance and operational alerts should be configured?
- Are consistent instance sizes available across all cloud providers?

## Security Requirements

Beyond the business and technical requirements, other requirements should be considered and identified for the design. Since Vault is a security product, security requirements must also be identified to ensure that the design not only follows the business' standards but may include extra precautions due to the confidential data stored and managed by Vault. These

may include additional techniques such as environmental hardening, operating system hardening, and following best practices for Vault configuration and operations. All these combined helps provide a defense-in-depth approach when designing and implementing Vault in an environment.

Security requirements that might be gathered may include:

- Will access to the Vault nodes be permitted, or will nodes be treated as immutable?
- How will the underlying operating system be secured?
- What methods will be used to protect system memory where encryption keys are stored?
- How are Vault audit logs ingested and correlated?
- What system account does the Vault service use?
- Are the Vault binaries, configuration files, and directories secured and audited?
- Is the storage backend secured from unauthorized access and modification?
- Is Vault secured by TLS?
- Who has access to unseal/recovery keys?
- Are TTLs minimized beyond a reasonable timeframe?
- Are Vault policies written to follow the principle of least privilege?

## End-User Requirements

It is essential to consider non-functional requirements as well. Often these can be determined by interviewing or participation of the end-users who will use Vault. These end-users could be various business units throughout the organization, DevOps teams who manage the CI/CD pipelines to automate application development or machine-to-machine authentication within an organization's application footprint. All these end-users are ultimately the Vault service consumers, and without understanding how these end-users will interact with Vault, it is impossible to run a thriving Vault service.

Requirements gathered for end-users might include:

- How will users or applications authenticate to Vault?
- What access is required within Vault once authenticated?

- Can applications be directed to performance standbys for read-only access?
- Does network and security infrastructure permit access to Vault from clients?
- Which secrets engines are required?
- What are the expectations around availability? How is degradation defined?
- Are there features or components not currently in place that may impact future Vault needs?

When gathering requirements from end-users, it might help to build a Service Level Requirements (SLR) template that teams can fill out before onboarding to Vault. The SLR helps the onboarding team consider how they will integrate with Vault but also helps the support team to understand the needs of each consuming team better.

## Requirements Gathering Process

In many organizations, requirements gathering can be an unnecessarily complicated process due to either the organizational structure or the number of participants wishing to voice an opinion. To minimize conflicts and increase efficiency, a workshop-type approach is generally recommended to streamline the requirements gathering process and design the Vault architecture. This Vault design workshop should be a collaborative effort between management, development, security, operations, automation, and network teams. If it is not feasible to include all parties in the workshop, representatives from each team can be included in the workshop.

> **Note:** Often, the workshop's participating members are defined as a virtual team that will see the project from the beginning to end. These virtual teams are created in place of a full-time team, as most participants still have day-to-day responsibilities. Members of this team frequently become part of the support team after the project is completed.

During the workshop, participants should provide input in determining the business, technical, security, and end-user requirements in which the Vault solution and architecture should meet. These critical decisions should be well documented and shared with all involved parties within the organization. After requirements are agreed upon and documented, the workshop should shift into the design phase. In this phase, the Vault

architecture and related services are designed to meet the requirements. This phase likely includes whiteboarding, discussions, and documenting decisions based upon HashiCorp best practices, available technology, internal capabilities, and documented policies and procedures. Any unanswered (parking lot) items should be documented, and action items should be assigned to the responsible parties with expected dates for follow-up.

> **Note:** In almost all cases, there will be an aspect of education during the workshop to ensure all parties understand how Vault works, what the Vault platform can do, and how Vault is deployed and operationalized. Providing some level of education allows all participating parties to help make educated decisions.

After the design has been agreed upon, it should be well documented and shared with all parties. Not only does this encourage open communication and collaboration among teams, it also assists in solving disputes over agreed-upon service levels, requirements, and design aspects. The documentation can also serve as a reference both during and after the Vault architecture build-out. Service teams can reference the document as they develop Terraform configuration files or automation scripts. New IT employees can reference the documentation to understand why Vault is used in the organization, how it is deployed, and where to seek additional information.

## *Service Level Objectives and Indicators*

Vault is an extremely flexible platform, and every organization will likely use it differently. As a result, the importance of the Vault service being deployed may differ significantly among organizations.

While some organizations may opt to use Vault for merely storing static secrets, other organizations may integrate Vault into complex applications, cloud platforms, and security and business processes. The service-level agreement (SLA) required for the Vault service would be vastly different in these two examples. Organizations using Vault for storing static secrets may not even have an SLA for Vault if their business applications do not rely on it to function correctly. However, another organization that has integrated it into critical applications may require Vault to be available 24/7, as applications might not function if Vault is unavailable. Therefore, establishing an SLA for Vault is critical for operational efficiency within that organization.

As an example, the following depicts two different scenarios for two different organizations using Vault:

> *Organization A has Vault deployed in its environment. Vault is used to store long-lived, static credentials for its applications. During application deployment, an engineer logs into the Vault UI, retrieves the database credentials, and adds the credentials to the connection string to authenticate and read data from the database. The application is not directly reliant on Vault and continues to function if the Vault service becomes unavailable. In this case, the SLA for Vault would be relatively low, or maybe even non-existent.*
>
> *Organization B has Vault deployed in its environment. Vault is used in various ways, including for applications accessing Vault using Vault's APIs to generate dynamic credentials for database access. Vault is also used to generate credentials for Terraform Enterprise to provision the application to the organization's preferred cloud provider. The engineers at this organization retrieve elevated credentials as needed from Vault, as well. In this case, Organization B's employees, processes, and applications are highly dependent on Vault's services, which means that the SLA for Vault will likely be stringent and require more "9s" of availability.*

Based on the examples above, it should be clear that when a critical application becomes dependent on the services and data that Vault provides, Vault becomes equally as critical as any other component within the application architecture. Vault should now meet or exceed the SLA associated with the dependent application to ensure its availability and uptime.

## Determining Service-Level Objectives

To ensure the Vault service meets and exceeds its SLA, service-level objectives (SLOs) must be established to provide indisputable and measurable characteristics. These measurements may include the availability, performance, and throughput available to integrated applications when connecting to Vault. For example, Vault's SLO may indicate that Vault has higher than a 99.99% uptime because many of the applications connecting to Vault have a similar SLO. Using a service-level indicator (SLI), the historical uptime statistics can be measured and reported on to ensure that they meet or exceed the SLO for availability.

When designing Vault, organizations should determine what SLOs and SLIs are critical to their infrastructure and operations and design the Vault infrastructure. Some of the SLOs commonly seen in Vault environments may include:

- **Availability** - Vault can respond to a client request within a given timeframe
- **Throughput** - Vault can handle the number of client requests within a given timeframe
- **Replication** - Vault is replicating data between clusters at an acceptable rate
- **Recoverability** - Vault data can be recovered in the event of a failure that meets or exceeds the organization's RTO
- **Response Time** – Vault can respond to requests within a specific timeframe, such as 20ms.

For example, availability is a service-level objective that can drastically affect how Vault is designed, deployed, and even how it is licensed. Referring to the examples above, Organization A likely needs only a single Vault cluster running Vault OSS to meet its requirements to store long-lived, static secrets using the K/V secrets engine. Assuming the use of the integrated storage backend, Organization A could provision five (5) nodes for Vault and would not need to obtain licenses. On the other hand, Organization B would likely need to provision clusters in multiple data centers or cloud regions for availability and

redundancy. Organization B may also purchase Vault Enterprise to use advanced features such as Performance or DR replication.

## Mapping Service-Level Indicators

Once SLOs have been defined and documented, the determination needs to be made on how they will be tracked. Service-level indicators (SLIs) are the data points that must be collected from the Vault environment to calculate how well the service and its performance is adhering to the committed SLOs. Data points may be derived from various sources, including, but not limited to, Vault telemetry data, third-party monitoring software, custom scripts, and automated jobs. More detail is provided in the section named Monitoring and Alerting for Vault.

After adequate samples have been collected, the measurement can be calculated and compared to the documented SLO to ensure it meets or exceeds the agreed-upon threshold. If the metric does not meet the standard, changes to the Vault architecture or service may be needed to ensure the service meets the agreed-upon SLO.

## Service-Level Reporting

It may be desirable to have SLA reporting readily available for management and service consumers. Almost all monitoring tools used for tracking SLIs offer functionality to generate dashboards and automated reports for the SLIs. It is recommended to make these available to all internal users to provide transparency of SLOs.

# *Designing the Vault Architecture*

Once all the requirements and service-level agreements have been gathered and documented, the Vault architecture can be designed. Although Vault has several different components to consider for design, a few key characteristics will shape the Vault environment. These include decisions such as the fault tolerance required, which storage backend is used to store Vault's data and whether replication is needed for Vault Enterprise deployments. Other critical Vault components, such as auth methods and audit devices, typically do not affect a Vault environment's overall architecture. However, certain secrets engines may affect the architecture if there are high-performance requirements, such as the Transit or Transform secrets engine.

## Fault Tolerance

One of the benefits of Vault is its flexibility to support a wide variety of failure scenarios. For example, Vault can service clients using a single node deployment or be deployed across multiple nodes to provide redundancy in the event of a failure. The data obtained in the requirements gathering session should determine the level of fault tolerance needed in an environment. Often, these decisions affect both the number of Vault nodes and the location where they are provisioned.

### *Vault Clustering*

Multiple Vault nodes configured to use the same storage backend automatically form a cluster for high availability, assuming the storage backend supports it. Nodes within the cluster are designated as either *Active* or *Standby* nodes. The first Vault node that grabs a lock in the configured storage backend becomes the active node. All other nodes in the cluster are designated as standby nodes. If the active node fails or becomes unavailable, a standby node within the cluster is promoted to replace the active node through a leader election process.

Vault clustering is available in both the OSS and Enterprise version of Vault. In a traditional Vault cluster running OSS, only the active node services client requests, and all standby nodes forward or redirect client requests to the active node. While this type of architecture provides high availability within the cluster, it does not increase scalability, meaning that more standby nodes do not improve the performance of a Vault cluster. In a Vault Enterprise cluster, the standby nodes in a cluster can be used as Performance Standby

nodes, which can service read-only requests for clients. Performance standby nodes continue to forward all write requests to the active node in the cluster. Regardless of the version, there is always one active node in a cluster (Figure 2-1).

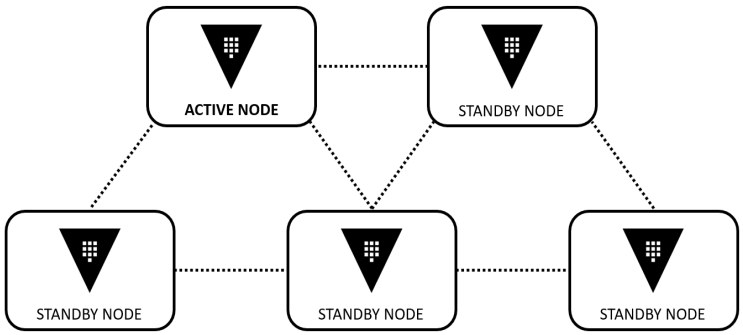



Figure 2-1

## *Accessing the Vault Service*

Many customers will front-end a Vault cluster with a load balancer to target the cluster's active node. Providing a front-end load balancer for a Vault cluster allows a reference with a 'friendly' DNS name, simplifying access to the cluster. This configuration is generally how load balancers are used in a multi-node environment.

In Vault's case, however, load balancers are used to provide high availability to the Vault service and not distribute requests across nodes. On the backend of the load balancer, health checks are used to determine the active node. Information on individual nodes can be obtained using Vault's dedicated health endpoint [/sys/health]. Each node responds with an HTTP status code that corresponds with its status and its node type within the cluster. As an example, a node that is initialized, unsealed, and is the active node returns an HTTP status code of 200. If a node is unsealed but a standby node, it returns a status code of 429.

The dedicated health endpoint has multiple default status codes (Table 2-1)

| Status Code | Description |
|---|---|
| **200** | Node is initialized, unsealed, and active |
| **429** | Node is unsealed and standby |
| **472** | Node is DR recovery mode secondary replication and active |
| **473** | Node is a performance standby |
| **501** | Node is not initialized |
| **503** | Node is sealed |

Table 2-1

As an example, Figure 2-2 represents a Vault cluster behind a load balancer. While the standby nodes return an HTTP status code of 429, the active node returns a 200 and is the Vault node to which all client requests are forwarded:

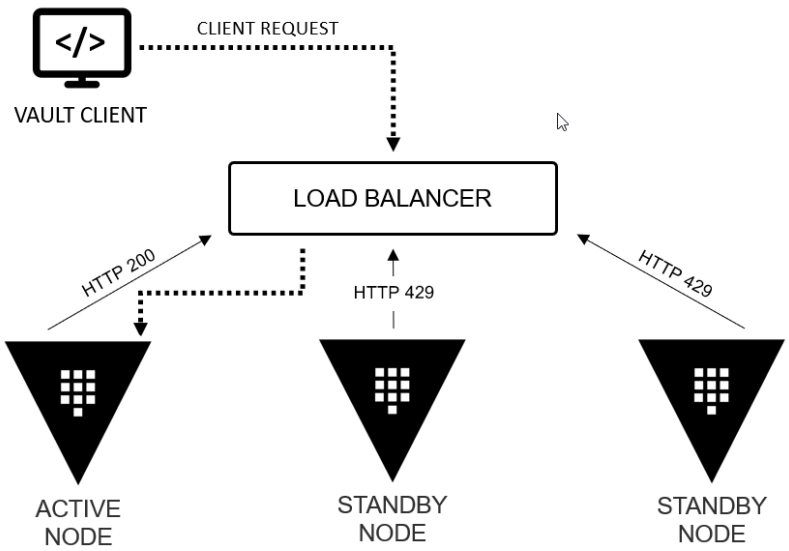



Figure 2-2

**Consul Service Discovery**
As an alternative to using a load balancer, organizations can opt to use HashiCorp Consul's service discovery feature to target Vault using DNS and service registration. Each Vault node would be configured to register the Vault service with Consul, and integrated health checks would allow Consul to determine each node's status and health. Consul automatically creates service registrations for both the active node as well as standby nodes. Once Vault nodes are registered, clients can access the service using the DNS records in which Consul is the authoritative server for responses to DNS queries. As an example, the Vault service could be accessed by sending a request to active.vault.service.consul, assuming the default Consul configurations are used.

Many organizations opt to go this route if service discovery or service mesh technologies are already in use, or they are looking for the opportunity to deploy such a solution. Keep in mind, though, that the Consul cluster serving as the storage backend for Vault should not be used for other Consul services, such as service discovery. A separate cluster should be provisioned to manage these services.

## *Node Placement*

Once the number of nodes has been decided based on the required fault tolerance, the individual nodes' placement must be considered to ensure the Vault service is as fault-tolerant as possible. Ideally, Vault nodes should be placed across several fault zones to safeguard all nodes' failure in the event of a failure of an underlying host or environment. Multiple fault zones are often easier to achieve in a public cloud environment, where a public cloud provider offers multiple availability zones. These public cloud-based availability zones are interconnected with high bandwidth, low latency networks to enable communication among resources residing in these different fault zones. For additional fault tolerance protection in the public cloud, Vault nodes can also use dedicated hosts or placement groups to ensure they do not run on the same physical host when provisioned in the same availability zone.

In contrast, organizations opting to deploy Vault on-premises may only have one or two physical data centers, and provisioning Vault nodes across multiple fault zones may not be feasible. However, most modern data centers use containerization or virtualization atop physical hardware to run and manage application workloads, which can help provide some level of fault tolerance. With this in mind, Vault nodes should run on different physical hosts/hypervisors, at a minimum. Technologies such as VMware DRS anti-affinity rules or

Kubernetes topology spread constraints should be used to prevent multiple Vault nodes from being hosted on the same physical hardware.

If an organization must use on-premises deployment, consider using network and power diversity. For instance, a standard Vault cluster architecture includes three Vault nodes and five Consul nodes. To achieve the best availability for on-premises deployment, place each node in a different rack in the data center to ensure each is connected to different power outlets and network switches. Network and power diversity are not as effective as availability zones in the public cloud, but it provides slight availability improvements over placing the entire cluster in a single rack.

As an example of node placement, a Vault cluster deployed in AWS should be provisioned across multiple availability zones, which provides the Vault cluster with numerous fault zones while providing proper connectivity between the Vault nodes. (see Figure 2-3)

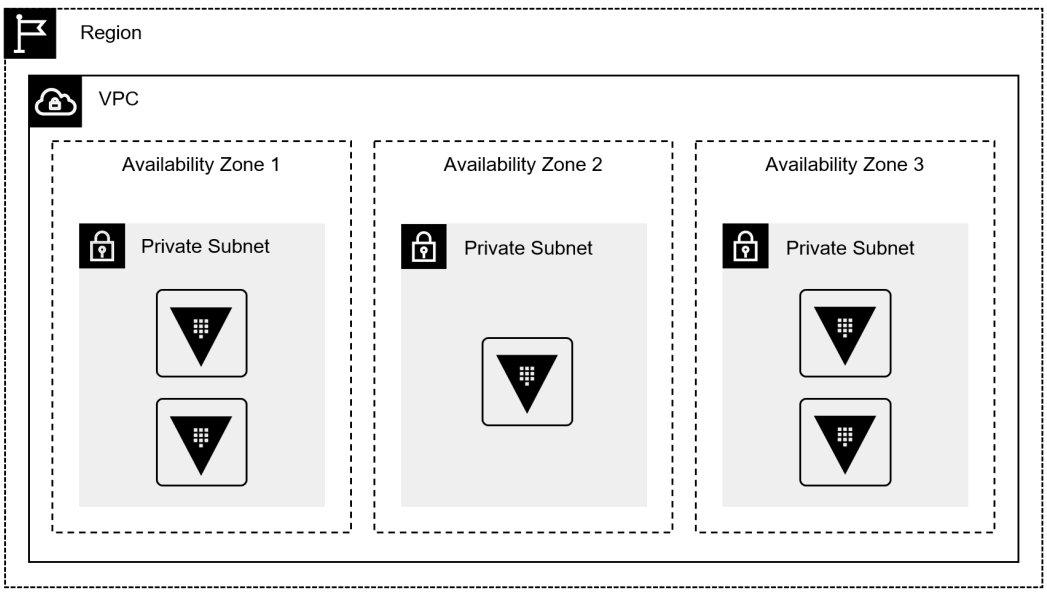



Figure 2-3

## Storage Backend

The storage backend is the most crucial Vault component, as it is the centralized location where Vault stores all its information. Therefore, the storage backend must be designed

with high-availability and fault tolerance as well. Out of the box, Vault supports a wide variety of options for the backend. Many of these storage backends support high availability, while some do not. For a Vault Enterprise deployment, a storage backend requires both high-availability and HashiCorp technical support. HashiCorp officially supports two options that provide both high-availability and are officially supported by HashiCorp's technical support: HashiCorp Consul and Integrated Storage.

### *Integrated Storage*

HashiCorp Vault's integrated storage is based on the Raft protocol ported over from Consul. It allows customers to maintain the encrypted data on the Vault nodes rather than using an external storage backend based on other technologies. Because Vault's data is stored locally, it eliminates the need to traverse the network for reads/writes. It also simplifies troubleshooting since integrated storage is configured in the Vault configuration file rather than on an external system like Consul. The fault tolerance supported by integrated storage (clustering and replication) also satisfies any requirements for a highly available storage backend.

> **Note**: Integrated storage is often referred to as *Raft* storage. The reference to *Raft* is found in both HashiCorp's documentation and the Vault configuration file.

Vault's integrated storage is essentially a distributed system using the Raft consensus protocol. It requires a relatively low latency connection between the nodes to ensure data is replicated correctly among the cluster members. While Vault nodes should be placed in multiple fault zones, it is essential to consider the network connectivity between the nodes. Due to this requirement, a single Vault cluster should not extend past the boundaries of a data center or public cloud region.

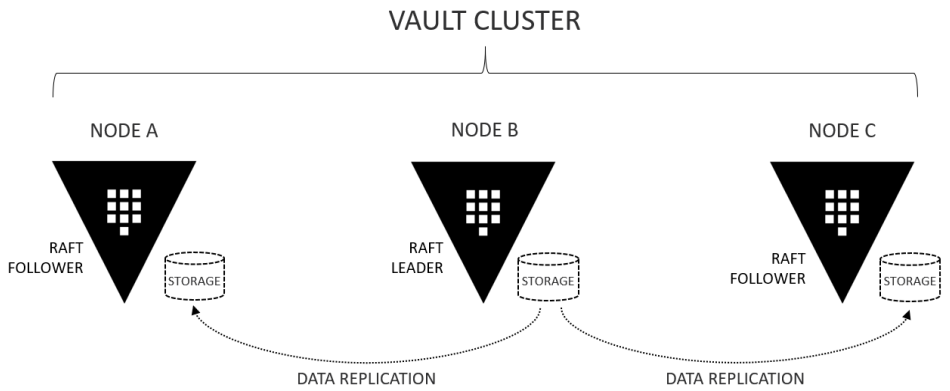



Figure 2-4

## *Consul Storage Backend*

HashiCorp also supports Consul as a storage backend for Vault. This storage backend is reliant on a separate HashiCorp Consul cluster, in which Vault can store persistent data using Consul's key/value store. The Consul cluster comprises of multiple nodes running the Consul agent in server mode and is frequently provisioned alongside the Vault nodes to minimize latency for communication.

Consul uses two main protocols to function: a *gossip protocol* and a *consensus protocol*:

- **Gossip Protocol**: Consul uses a gossip protocol based on Serf to manage membership and communication within the cluster. Consul uses two different gossip pools: LAN connectivity (local) and WAN connectivity (cross data center requests). For a Vault implementation, only LAN is used for the storage backend.
- **Consensus Protocol**: the consensus protocol is based on Raft and is responsible for managing the quorum, log entries, and cluster leadership.

A node in a Consul cluster is either a *leader* or a *follower*. When Consul is started, each node starts as a follower, but servers work together to elect a leader using the Raft protocol. The leader is responsible for processing all new transactions. Once transactions are committed, the leader is responsible for replicating that data to all other server nodes in the cluster.

Each Vault node runs the Consul agent in *client mode.* By using client mode, each Vault node joins the Consul cluster as a client. This method allows the Vault service to communicate with its local Consul client rather than directly with the Consul cluster when establishing connectivity to the storage backend. As a result, the Vault configuration file remains unchanged even if the underlying Consul nodes are modified or replaced during maintenance or upgrades. An example of the Vault architecture with a Consul storage backend can be seen in Figure 2-5.

> **Note:** If Consul is selected as the supporting backend, it is highly suggested that employees responsible for Vault/Consul fully understand the protocols used in Consul to support the storage backend. Vault is entirely reliant on the storage backend. Therefore, if Consul goes down, so will Vault!

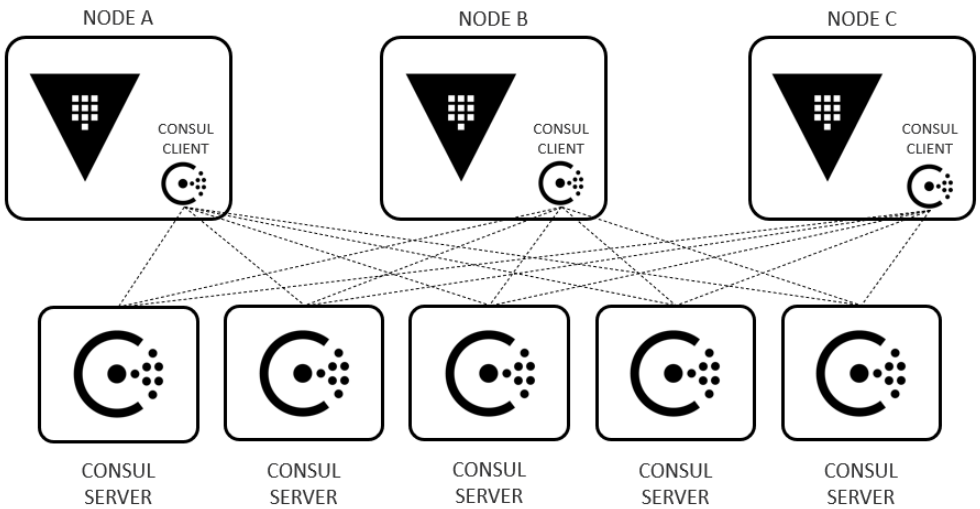



Figure 2-5

## Replication Design

In most cases, organizations have critical applications and infrastructure in multiple data centers or cloud regions. They may require that the Vault service be replicated and made

available to applications in their respective environments. This replication needs to include data, policies, and configurations originating in the primary Vault cluster to minimize the administrative overhead. Vault replication is a Vault Enterprise feature that provides this seamless functionality in an organization's Vault environment through log shipping. Organizations can opt to provision Vault clusters in multiple locations, generally close to application workloads, and use the Vault replication feature to keep these clusters in sync. For Vault replication, clusters are labeled as either a *primary* or *secondary* cluster:

- **Primary Cluster:** The primary cluster is the authoritative cluster, responsible for all write actions throughout the Vault environment while replicating changes to other clusters. In most cases, it is the first cluster that is deployed and configured within an organization.
- **Secondary Cluster:** A secondary cluster receives data from its primary cluster. Depending on the type of replication, secondary clusters are also responsible for servicing read requests from local Vault clients. Additionally, secondary clusters can be a secondary cluster while also serving as a primary cluster for other secondary clusters.

The primary cluster placement is important to consider, especially in cloud environments that incur costs for different types of network traffic. Many customers place the primary cluster in either the first environment that consumes the Vault service or the environment with the most applications accessing Vault. However, other customers who may be more cost-conscious might opt to place the primary on-premises and replicate to secondaries in the public cloud. This placement decision takes advantage of the fact that most cloud providers do not charge for ingress network traffic. Otherwise, when the primary is provisioned in a cloud environment and replication is configured, replication traffic would incur costs because it would consist of mostly egress traffic.

With that being said, the public cloud provides flexibility for scaling the resources for the primary cluster nodes. The same flexibility is harder to do on-premises when an organization may be at the mercy of hardware vendors and their associated supply chains.

When Vault replication is enabled and a cluster is configured as a primary, Vault creates a self-signed certificate. As secondary clusters are configured for replication, Vault generates a certificate and private key for the secondary cluster. Vault combines the certificate, private key, and other critical data, such as information on communicating with the primary, into a

replication bootstrapping bundle called the secondary token. The secondary token is submitted to the secondary cluster by the Vault administrator, and replication is initialized. The second cluster uses the certificate and private key to establish a mutual TLS connection to the primary cluster. This certificate and the associated private key are not the same TLS certificate and private key used for the Vault TCP listener.

> **Note:** If the Vault cluster is provisioned behind a load balancer and the secondary cluster cannot communicate directly with the primary cluster nodes, SSL offloading should not be configured on the load balancer. Such a configuration breaks the mutual TLS for replication. If such a configuration is required, the load balancer should be configured for SSL pass-through.

The workflow for replication configuration is as follows:

1. The primary cluster is marked as a primary cluster for replication.
2. The Vault administrator creates the secondary token on the primary cluster.
3. The secondary token is submitted to the secondary cluster.
4. The secondary cluster unwraps the secondary token from the primary cluster.
5. The secondary cluster communicates with the primary to initiate replication.
6. Data is replicated from the primary cluster to the secondary cluster.
7. The secondary cluster is ready.

> **Note:** Once a cluster is marked as a secondary, Vault removes all data and configurations upon initialization to start replicating data from the primary. If any data stored on the secondary cluster, make sure it is backed up or migrated off before activating the secondary cluster.

Vault replication uses two default TCP ports for replication, tcp/8200 and tcp/8201. While replication traffic only uses tcp/8201, the initial configuration of Vault replication requires that the secondary cluster can communicate to the primary on tcp/8200 to unwrap the secondary token. Once the replication is configured, replication traffic only uses tcp/8201 moving forward. The IP address or hostname used for these ports is configured using a Vault configuration file parameter and is required for highly available environments.

Vault Enterprise supports two options for replication: disaster recovery replication and performance replication. While both options replicate data from the primary Vault cluster, there are distinct differences between them. Application architecture and disaster recovery are factors that organizations must consider when deciding which replication option to use.

## *Disaster Recovery (DR) Replication*

When disaster recovery replication is configured, the primary cluster replicates the configuration, policies, and stored data to the secondary cluster. DR replication replicates all tokens and associated leases to the secondary cluster as well. Replicating the tokens and leases allows the DR cluster to immediately start serving applications if the DR cluster is promoted to a primary. Without these tokens and leases, requesting applications would need to reauthenticate to Vault as the existing tokens would be unknown.

DR clusters do not service read or write requests from clients. Therefore, DR replication is generally used for organizations needing a warm-standby if the primary cluster becomes unavailable (Figure 2-6). In many cases, a DR cluster is provisioned and maintained in each data center or cloud region where Vault services applications to provide continuous operations in the event the primary cluster fails.

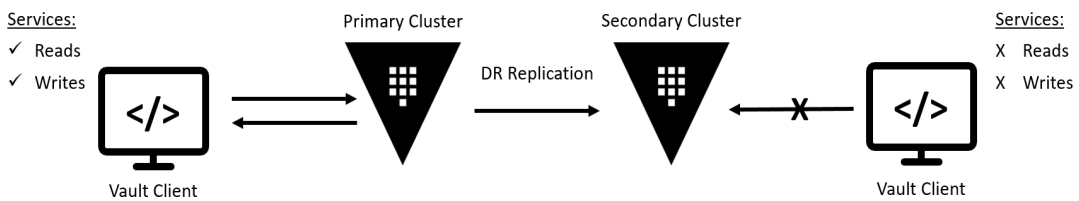



Figure 2-6

An important aspect to keep in mind is that DR clusters are not automatically promoted if a primary cluster fails. Different organizations may have different definitions of what an outage means to them, and Vault administrators should determine when a secondary should be manually promoted to serve applications. Allowing organizations to define an outage helps avoid a split-brain scenario where multiple clusters are marked as a primary cluster. This scenario would break replication and cause problems within the Vault architecture.

## *Performance Replication*

Like DR replication, performance replication copies the underlying configuration, policies, and data from the primary to the secondary clusters. A key difference in performance replication is that performance replicated clusters can service clients while keeping track of their own tokens and leases. Any client requests requiring a write are still forwarded to the primary cluster. The client's authentication requests are all handled locally, and the secondary cluster generates and issues tokens and associated leases that are valid only on the secondary cluster. Due to this capability, service tokens are not replicated from the primary cluster to the secondary. Any clients who have successfully authenticated to the primary cluster will need to re-authenticate to the performance replicated cluster if promoted to the primary due to an outage.

Seeing that performance replicated clusters can service client requests, performance replication allows organizations to extend the Vault service to other data centers or cloud regions where application workloads are deployed. Clients local to the primary cluster would send requests to the primary cluster, while clients local to the secondary cluster would make requests to the secondary cluster (Figure 2-7). This architecture reduces network traffic for client requests while keeping the latency between clients and Vault to a minimum.

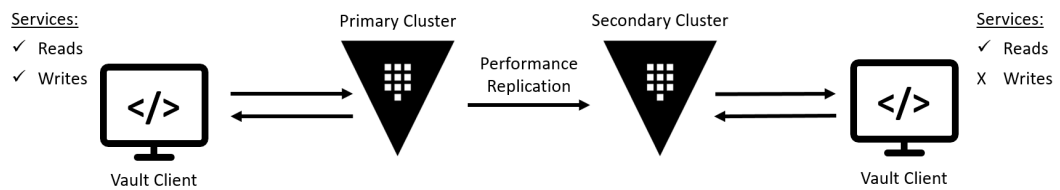



Figure 2-7

## *Choosing Between Replication Types*

When designing the Vault infrastructure and the desired replication type, an architect should use a few key characteristics to determine when to use performance replication or DR replication. The first characteristic is in regard to how the organization's application workloads are deployed and accessed. Organizations that use an active/passive architecture would likely choose DR replication since there are no active application workloads in the secondary data center or cloud region. If the organization needs to failover

to applications in the secondary data center, the Vault secondary cluster can be promoted to a primary to service the local applications. If there are active workloads in the second location, performance replication should be used to better service the local clients. If the original primary fails, the performance replicated cluster can be promoted to become the new primary and fully service Vault clients.

The other element that must be considered is licensing costs as DR replication and performance replication clusters have different pricing models. Since a performance replicated cluster actively services client requests, performance replicated clusters are deemed a production deployment of Vault and must be licensed as such. On the other hand, DR replicated clusters are usually labeled as a warm-standby cluster and are licensed with that in mind. Note that HashiCorp licensing models may change, so check with the local HashiCorp rep to verify.

Note that while this section focuses on deciding between performance and DR clusters, it is quite common for organizations to employ both types throughout its Vault infrastructure. The key differences between DR replication and performance replication are summarized using Table 2-2.

| **Characteristics** | **DR Replication** | **Performance Replication** |
|---|---|---|
| Replicates Configuration, Policies, and stored data | X | X |
| Replicates Tokens and Associated Leases | X | |
| Services Client Read Requests | | X |
| Requires Production License | | X |

Table 2-2

## Single Cluster

In general, deploying a single cluster in production should be avoided if possible, as it does not provide redundancy in a complete cluster failure. However, there are various reasons that a multi-cluster Vault environment might not be a good fit for an organization. Due to constraints around budget, resources, and procurement, some organizations may be unable to procure Enterprise licenses for multiple clusters. On the other hand, some organizations may opt to run open-source. Assuming the risk is acceptable to the business, a single Vault cluster can be deployed and service its applications with little downtime.

For Vault deployments using a single cluster, organizations should apply as many of the characteristics for fault tolerance as possible to provide a highly available service. For open-source clusters, keep in mind that Vault is a scale-up application since only the active node services client requests. However, enterprise customers can quickly scale out the cluster using performance standby nodes, which reduces the workload on the active node. Additional DNS records and load balancers can be configured to direct read-only applications to the performance standbys. In either case, Vault clustering increases local availability for the Vault service.
Many times, the storage backend is the limiting factor regarding overall performance in a cluster. When scaling the Vault cluster/nodes for performance, the storage backend must be scaled to handle the additional read and write requests. Performance metrics such as CPU and memory must be closely monitored to ensure the desired performance.

Node placement during implementation is even more critical for organizations running a single cluster. Use numerous fault zones, where available. Organizations should also ensure that the provisioning process for Vault and the storage backend is automated to decrease the time to recover in the event of a failure. Data protection, which is discussed later in this section, is crucial for service recovery.

If a single-cluster deployment is sufficient for an organization, it should ensure that a DR replicated cluster is also deployed and frequent snapshots are stored in a secure location. Having both in place helps ensure that the cluster can be recovered quickly and that any errors that cause data loss can be resolved.

> **Customer Story:** An e-commerce firm embraced an open-source first strategy when identifying solutions for use within its microservices

environment. The firm adopted Vault as the primary platform for storing and generating credentials required for its application stack, all of which were containerized and deployed on Amazon EKS. Since open-source Vault was selected, a single Vault cluster was provisioned in its AWS environment so applications could access the required secrets. For fault tolerance, Vault was provisioned across multiple AWS availability zones, along with the Consul cluster serving as the storage backend. Data protection was achieved by automating Consul snapshots to be taken every hour and saved to Amazon S3. Since Vault and Consul were provisioned using Terraform, a new Consul and Vault cluster could be quickly deployed in the event of complete cluster failure. This architecture is an excellent example of how Vault can properly serve high-throughput applications while understanding and accepting the risks of running Vault with only a single cluster.

## Multi-Cluster - Single Region/Data Center

When moving from a single cluster to multiple clusters, Vault's design increases in complexity. However, this opens the door to endless configuration possibilities to serve applications better. Multiple clusters within a single region can be as simple as a primary cluster and a warm-standby with DR replication. It can also be as complicated as multiple performance clusters deployed throughout a large enterprise's data center and accessed by thousands of applications.

> **Note:** This section defines a multi-cluster deployment as multiple Vault clusters configured as a single, connected environment using either performance or DR replication. It does not refer to deploying multiple, disparate Vault clusters.

### *Redundancy*

The first significant benefit of using multiple clusters with a single region or data center is service redundancy (Figure 2-8). If the primary cluster becomes unavailable, a second cluster can be promoted to be the new primary. This design not only provides the organization with fault tolerance within the cluster by deploying multiple nodes, but it

provides redundancy at the cluster level, as well. Keep in mind the differences between the replication types before selecting the replication type to use for redundancy. Also, remember that licensing costs come into play here, as well.

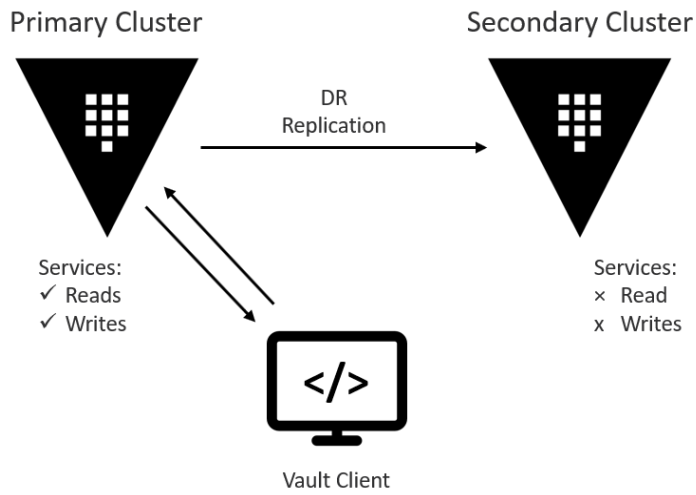



Figure 2-8

### *Scale*

The second significant benefit of using multiple clusters within a single region or location is to scale the Vault service using performance replication. Every time an application consumes the Vault service, a read or write request is made to Vault. For example, retrieving a secret stored in the K/V secrets engine would be classified as a read request. However, if an application is sending data to be encrypted or is requesting a PKI certificate, the request would be considered a write request. The key distinctions of whether to scale out with additional clusters sit with the request type and the volume of those requests.

In the case of servicing read-heavy applications, performance replicated clusters can be provisioned throughout the infrastructure to handle the requests. This design would provide a fully scalable and flexible Vault environment that would support an exceptionally high SLA. Each of the performance replicated clusters would have an ingress endpoint and DNS record that applications use to connect (Figure 2-9). Thanks to replication, each performance cluster would have an identical configuration and contain the same secrets needed by applications to target any accessible clusters. That said, applications should

target the same cluster, where possible, to prevent errors due to re-authentication requirements. An alternative would be to build this cluster targeting and re-authentication error handling into the applications. By doing this, the organization could ultimately treat entire clusters as immutable.

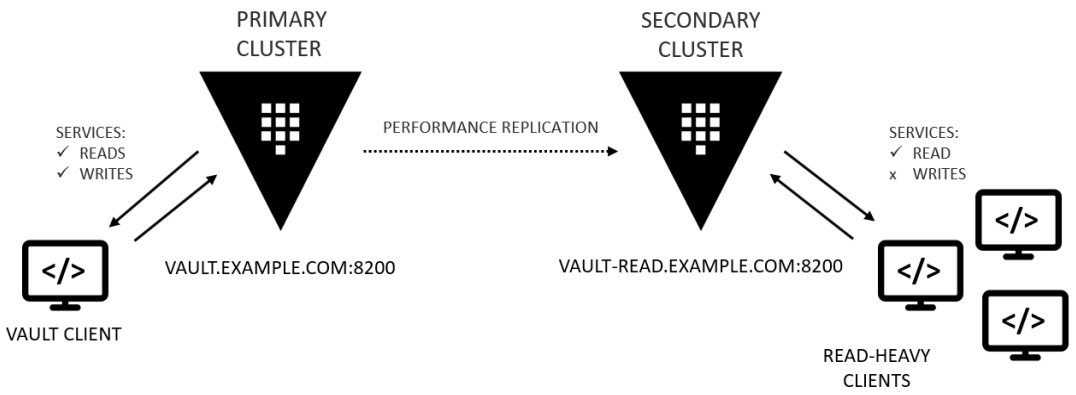



Figure 2-9

### *Limitations*

While a multi-cluster strategy is ideal, the most significant limitation of running multiple clusters in a single region or data center is the lack of site redundancy. Suppose an entire cloud region or an organization's entire data center would become unavailable, conceivably due to a natural disaster. In that case, it might be challenging to bring the Vault service back online. That said, this risk may be acceptable to organizations whose policies mandate that applications only span multiple fault zones for redundancy. Since availability zones are, by AWS's definition, isolated data centers, a single cloud region with multiple availability zones would satisfy the policy.

> **Customer Story**: A utility company that provides electricity and natural gas to millions of customers was rearchitecting its legacy, customer-facing application to a modern, cloud-native application using microservices and AWS EKS. To centralize the storage of critical secrets for applications and generate dynamic credentials for its CI/CD pipeline and Terraform Enterprise, the company adopted HashiCorp Vault Enterprise. During the Vault design phase, it was determined that

Vault would only be used for newer, cloud-native applications that were being deployed to a single public cloud region due to its feature-rich API and capabilities. The Vault architecture needed to be fully redundant at both the node and cluster level. Vault was implemented in the public cloud in a single region but across multiple availability zones. A DR cluster was also deployed in the same region to ensure the configuration, policies, tokens, and KV data were replicated to the secondary cluster. In the event of a failure, the company can quickly failover to the other cluster, and applications can continue functioning as usual. Future design stories for this utility company include extending to a second region, to which Vault will also be extended.

## Multi-Cluster - Multiple Regions/Data Centers

A highly available Vault implementation ready to service robust organizations may include multiple clusters and multiple regions in its architecture. Vault deployments that incorporate multiple clusters in multiple regions or data centers provide the most flexibility and most significant availability. Organizations using this design can quickly scale their infrastructure using multiple regions or data centers.

### *Redundancy*

While redundancy can be achieved within a single region or data center, performance and DR replication can protect against a full region or data center outage. As discussed, performance replication clusters can be used to extend the Vault service to additional sites. However, a typical pattern is for each performance replicated cluster to be configured with its own respective DR cluster in the other region or data center (Figure 2-10). With this model, the configuration, policies, and data are replicated to the additional sites so applications can access the Vault service. However, because tokens are not replicated using performance replication, DR replicated clusters can be deployed to ensure that each cluster's tokens and associated leases are replicated elsewhere. Keep in mind, though, that different use cases may alter the use of the DR clusters. For example, if long-lived TTLs are not being used or the applications can easily handle re-authentication, DR clusters might not be required.

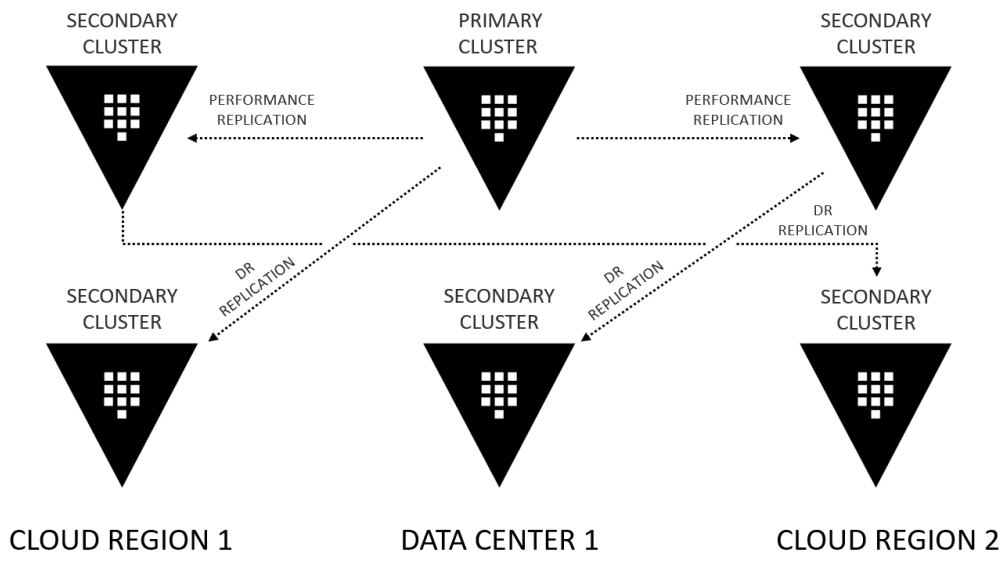



Figure 2-10

### *Scale*

Scaling to multiple data centers and regions for applications is a typical design pattern used by many large enterprises but it can be adopted by organizations of any size. To provide highly available services to their customers, organizations frequently provision applications in an active/active deployment in multiple data centers or public clouds (Figure 2-11). When these applications rely on the Vault service to retrieve secrets, generate dynamic database credentials, or encrypt data on the fly, Vault needs to be scaled up as the number of requests increases. If Vault needs to be scaled due to a large number of reads, performance standby nodes can be used. If Vault needs to support a larger number of writes, performance replication clusters can be used since secondary clusters can reduce the load on the primary cluster.

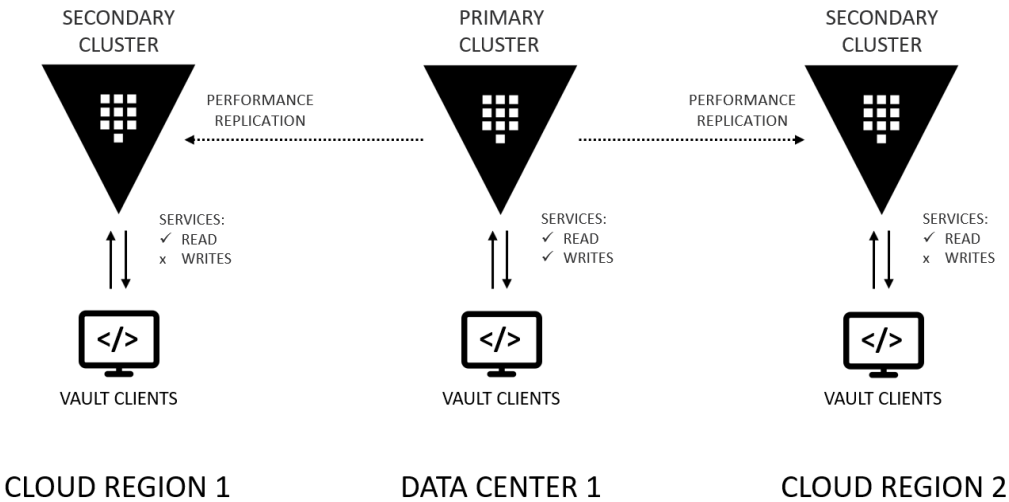



Figure 2-11

**Customer Story:** A massive tech company needed a performant secrets management tool that could protect sensitive data being collected and meet the demands of the trillions of transactions being processed globally every year. This company was running applications out of data centers and multiple cloud providers in various global regions, so the secrets management experience needed to be consistent across regions and cloud providers. The company deployed many Vault clusters in a replication set across multiple cloud providers and regions, making all clusters accessible to all applications as a way to ensure applications were always able to retrieve secrets regardless of the status of any single cluster. This model also allowed them to ensure the secrets retrieval process was consistent, even in regions like China, where cloud provider APIs are multiple versions behind the rest of the world.

## Disaster Recovery

As with any critical application or service, disaster recovery must be considered part of a Vault design. Since Vault stores all its data on the storage backend, its nodes are often

treated as immutable and is why the primary focus of disaster recovery should be placed on the storage backend. The storage backend should also be protected against data loss due to corruption or problems with the underlying hosts or operating system. The process to back up the storage backend differs depending on the storage backend being used for Vault.

### *Integrated Storage*

Based on the Raft protocol, Vault's integrated storage provides the ability to take snapshots for data protection. Snapshots are point-in-time backups that include all Vault data and configurations and should be the primary method for protecting data in a cluster. The snapshots can be used for data recovery if it ever becomes necessary or for testing purposes.

Once snapshots are taken, they should be copied or migrated to external storage, cloud-based storage, or enterprise-level storage designed to provide high-availability and durability. Frequent snapshots should be scheduled, and the frequency of those snapshots depends on how often data on Vault is being created or modified. The organization's data retention policies should dictate how many snapshots to maintain and how long those snapshots should be kept.

### *Consul Storage*

Like integrated storage, Consul includes a built-in 'snapshot' command for saving, restoring, and inspecting the state of Consul nodes for disaster recovery. This method should be the primary method for protecting data in Consul. These snapshots are point-in-time backups that include all data in the key/value store, the service catalog, prepared queries, sessions, and ACLs configured in Consul. Snapshots should be taken frequently to help ensure Vault and Consul is recoverable if all nodes become unavailable.

In Consul, there are two ways to initiate a Consul snapshot: a manual command or the Consul Snapshot Agent provided in Consul Enterprise. If the cluster is running Consul Enterprise, it is recommended to use the Consul Snapshot Agent for automated backups.

For Consul Enterprise clusters, the Consul Snapshot Agent runs as a separate service. The service is configured to run on each node, and the Consul snapshot agent obtains leadership before executing a snapshot. Obtaining leadership ensures that only a single

snapshot is executed per the scheduled configuration rather than having each node saving a snapshot. The Consul snapshot agent can also be configured to automatically save the snapshot on cloud-based storage, such as Amazon S3 or Azure Blob Storage, and maintain a pre-configured number of snapshots on the storage solution.

# *Finalizing the Deployment Plan*

Building a shared service like Vault does not happen overnight and delivering a quality product will take time. Often, large Vault projects take months to complete, depending on the number of resources available. A deployment plan should be in place to define the scope, approach, and execution plan that the project team will follow.

## Preparation

Deployment plans start with preparation activities, including gathering requirements, identifying the service level agreements, objectives, and indicators, and finalizing the Vault design. Concurrently, the support team needs to be identified and presumably trained on Vault to prepare for deployment and support of the service after it is launched. It is also important to remember that backlog creation or grooming sessions should be scheduled to determine the tasks required to build the Vault foundation and deployment.

## Building the Foundation

After preparation has been completed, the deployment plan shifts focus to the buildout of supporting services and technologies that will directly manage the Vault environment. These services may be as simple as creating a code repository dedicated to Vault configurations and policies. They can also be as complicated as building a deployment pipeline used to provision Vault and any future modifications to the service. These tasks are generally defined and tracked in a project tracking application that Vault team members can follow to complete the assigned work. External resources throughout the organization may be engaged, depending on the task at hand.

Potential build phase tasks include:

- Obtaining access to environments where Vault will be deployed
- Creating TLS certificates for Vault and Consul nodes, if applicable
- Requesting firewall changes to enable communication to Vault
- Developing Terraform or other code for provisioning and configuring resources
- Writing systemd service configurations
- Creating storage resources to store Consul snapshots

## Deploying Vault

After supporting services have been built to specification, code has been committed, and deployment pipelines have been built, Vault can be deployed. While organizations may differ, Vault is predominately deployed in a development environment before any other environment. Deploying in a development environment not only allows the team to test items such as configurations, pipelines, and jobs, but it also allows development teams to start integrating applications into Vault. While improvements are made to the deployment process, the Vault support team can engage with development teams to enable auth methods, secrets engines, and write policies for application integration. Organizations may also opt to provision a Vault cluster for operational testing, feature testing, and upgrades.

After validating the deployment of the development environment, the team can provision and configure the resources for the production environment. The production environment should be provisioned and configured based on the architectural design from the preparation phase. If modifications are needed or requested, the design document should be updated to ensure it contains the most up-to-date design. Assuming a successful deployment, the Vault support team and application developers can promote code to production and interact with the live environment.

# *Choosing the Support Team*

The success of any shared service is dependent on the team that manages or supports it. In Vault's case, having a strong technical team in place will help ensure the success of the various teams using the service and the success and efficacy of the security program. Choosing a support team requires selecting the optimal number of people to manage Vault, selecting members who have the necessary background and experience, and scaling the support team as the Vault service grows and matures. A support team may look different for each organization, which can vary based on the size of the organization or the experience level of the individuals selected for the team.

## Support Team Experience

Most support efforts will be spent either performing system administration work or helping software/system engineers understand their Vault integration. Thus, it is vital to hire support personnel who have a strong understanding and experience of Vault and integrated technologies. While it seems obvious to put a security team in charge of managing Vault, this is often not ideal. Although security is an essential partner for a successful implementation, security personnel are often not equipped to deal with issues or support requirements that come with managing the Vault service. The ideal model is to hire team members with automation and application integration skills, such as Site Reliability Engineers (SRE) and DevOps Engineers. This support team can then partner with security to ensure the service is meeting security requirements.

Below are some descriptions of the key roles of a Vault support team and descriptions of the ideal candidates to fill them.

### *Site Reliability Engineer - Required Experience*

The goal for this senior-level role is to design the Vault service, provide technical guidance, drive internal programs surrounding the Vault service, and handle second-tier support for more complicated issues. This role should also provide mentorship to the less experienced personnel on the team. These senior team members should be involved throughout the entire project, starting from the early phases of Vault design through implementation. This continuity will provide the engineers with an intimate understanding of the service.

When recruiting site reliability engineers, the ideal candidates should have ample experience working in fast-paced environments. An example of the desired experience is listed below:

- 5-7 years experience managing Linux environments
- 3-5 years experience with Python scripting
- 3-5 years experience managing public cloud infrastructure
- 3-5 years experience managing shared, multi-tenant services
- 3-5 years experience with configuration management tools
- 2+ years working with CI/CD tools
- 2+ years working with version control tools
- Deep understanding of system/service monitoring
- Basic understanding of the Software Development Life Cycle
- Basic understanding of identity management services

## *DevOps Engineer - Required Experience*

While the term DevOps is more of a culture and process than a job, many engineers responsible for automation and integration commonly find themselves working under a DevOps Engineer title. On a Vault support team, the DevOps Engineer should be the front line of support, spending most of their time supporting users or tuning the Vault service. The DevOps Engineer role will also be responsible for onboarding and monitoring the Vault service health.

The ideal candidate for a Senior DevOps engineer should have the following experience:

## *Senior DevOps Engineer*

- 5-7 years of experience with software development
- 3-5 years of experience managing Linux environments
- 2+ years of working with system/service monitoring
- Deep understanding of software development lifecycle
- Experience managing shared, multi-tenant services

- Experience managing public cloud infrastructure
- Experience with configuration management tools
- Basic understanding of identity management tools

Candidates with similar skill sets but less experience can be considered for junior-level roles to fill out the rest of the team. These junior-level candidates should work under the guidance of the senior-level support personnel.

### *Team Composition*

It is worth mentioning that the Vault "team" should not be defined as a single individual. There should be a minimum of two individuals on a team to ensure that support requirements are met. However, the ideal starting point in mid-sized to large organizations is at least three members. A team of this size will help ensure on-call needs are met without negatively impacting the ability of members to maintain a healthy work/life balance.

An ideal support team would be comprised of one Senior SRE and one Senior DevOps Engineer. Additional vacancies can be filled with DevOps Engineers to accommodate the workload. As additional applications and integrations are added, the support team should grow accordingly. If an organization can only equip the team with a single member, the Senior SRE candidate will likely provide the best experience to support the Vault service.

## Support Phases

It is not necessary to hire all support personnel at once. The team should start small, filling two to three positions, and scale up as needed. Thinking of Vault support in terms of phases is a helpful approach because it acknowledges how support needs change as Vault is implemented.

### *Phase One*

Typically, a Vault deployment starts with a preparation phase. During this phase, the base configuration, related automation, and telemetry integrations are built. However, traditional users do not yet consume the Vault service while this work is being performed. During this phase organizations can enlist a small team that will be responsible for building the Vault service.

### *Phase Two*

After Vault is up and running and supporting tools and processes are ready for use, the service is ready for production use. However, it is ***highly recommended*** to start with a few targeted use cases to get a clear understanding of how the service will be used. This approach also allows the support team to learn what support requests may take place during normal operations. The same team will likely continue to support and maintain the Vault service during this phase.

During this initial integration phase, the Vault team should work closely with the product teams to understand the requirements for integration. With each successful onboarding, the team should be building internal advocates who can help champion the service to other teams. This strategy can boost adoption and bolster the organization's return on investment

### *Phase Three*

Once the first few use cases are successful, the service should be ready for general onboarding and organizational adoption. Depending on the level of technical savviness of the user base, the same support team may handle onboarding new Vault clients and continue to support the service. If the newly onboarded use cases are complex or require additional time and effort, additional personnel may need to be added to the support team.

As adoption continues to increase throughout the organization, various teams will approach the support team to use Vault. Most of these teams will understand the value of the service and will want to consume it in some way. If done correctly, little will need to be done to find new use cases for Vault.

As these new use cases continue to increase, it is essential to keep track of the incoming workload. The support team needs to communicate expectations to requesters, and corresponding tasks will need to be controlled to prevent the support team from being overwhelmed. The support team will need to maintain a 50/50 split between onboarding/maintenance and service improvements.

### *Phase Four*

During this last, ongoing phase, late adopters will be onboarded to Vault. Since these are likely teams that operate as laggards on the technology adoption lifecycle, they may require

additional technical assistance. It will likely be necessary to expand the support team because of this additional work.

These late adopters will need to be approached in an organized manner. This work will require the support team, perhaps with the assistance of a project manager, to determine which teams have not yet been onboarded and proactively reach out to them to begin the process.

### *Staffing Recommendations*

It is recommended that a full-time employee be hired for every 15-20 teams that will be using Vault. This approach will right-size the support team so that appropriate tasks can be performed for support/maintenance and continuous service improvements.

The number of support team members should be calculated based on the number of teams using Vault and not necessarily the number of applications. Regardless of the number of applications each team manages, teams generally only need to be trained once to understand how to integrate with Vault. It is uncommon that the number of applications that a team manages will create an additional support burden on the Vault support team.

# 3 Building a Vault Service Foundation

### *This Section Covers:*

- Operating System Configuration
- Vault Service Configuration
- Monitoring and Alerting for Vault
- Automating Vault Operations
- Support Documentation
- Executing the Deployment Plan

A well-configured Vault service is only as effective as the foundation on which it is built. In Vault's case, a strong foundation includes precise technical configuration and robust support documentation to match. This chapter covers how to lay this foundation, all the way up to Vault deployment.

# *Operating System Configuration*

A well-defined operating system provisioning process is critical to ensuring the Vault service runs in a secure, consistent, and performant manner. This provisioning process can improve many elements of the Vault service lifecycle. These benefits include the deployment of validation pipelines to test configuration changes and manage operating system upgrades. Building and maintaining a process to provision the underlying operating system is essential because it sets the standards for the security of the Vault service.

Vault supports a variety of underlying operations systems, but this section provides guidance for CentOS 8. For specific details on any of these commands or the flags issued, consult the command's manual page.

## System Hardening

Since Vault is a security service, it is critical to secure the underlying operating system. Fortunately, many security organizations align with standardized guidelines and controls, such as CIS, to ensure that an operating system is adequately hardened. There are many ways to secure an operating system, and the strategies used will depend on the technologies and skillsets available at each organization. That said, this section covers some of the basic operating system configuration items.

### *Configure /tmp*

During the initial installation process, the */tmp* directory should be created as a separate partition. If this partition was not created during installation, then a partition will need to be created after the installation. Keep in mind that if the entire disk were used during installation, the creation of this directory would be a much more complicated process.

After the partition has been created, add a mount entry to */etc/fstab* with the *nosuid* and *noexec* flags set to prevent execution within the */tmp* directory, as shown in Figure 3-1.

```
$ <PartitionPath> /tmp ext4 loop,nosuid,noexec,rw 0 0
```

Figure 3-1

Manually mount the new partition. This action only needs to be done once since the operating system will automatically mount the partition on reboot.

```
$ mount -a
```

Figure 3-2

### *Disable Auto-Mounting (autofs)*

AutoFS is a common Linux service for automatically mounting removable media and often comes standard on many Linux distributions. However, allowing auto-mounting of such media could compromise the security of the host and should therefore be disabled. AutoFS can be disabled using the commands in Figure 3-3.

```
$ systemctl stop autofs
$ systemctl disable autofs
```

Figure 3-3

### *Configure SSH*

For organizations that permit remote access to the Vault nodes, the following configurations should be updated in *sshd_config* to secure access to a host. Note that the configuration suggested in Figure 3-4, *PermitRootLogin* is disabled, as most security experts concur that the root account should not connect via SSH. In most situations, administrative tasks should be performed by a configuration management tool. SSH should be reserved only for break-glass situations.

```
PermitRootLogin no
ClientAliveInterval 300
ClientAliveCountMax 0
PermitEmptyPasswords no
```

Figure 3-4

It is also recommended to authenticate to hosts using SSH keys rather than local user accounts. Local user accounts tend to be shared, which eliminates accountability.

Credential rotation is often neglected on these static accounts, posing another security concern. By using SSH keys, a configuration management tool can easily upload the team member's public keys to the associated *authorized_keys* file to enable access. This change can be achieved by setting the value in the *sshd_config* file, as found in Figure 3-5.

```
PasswordAuthentication no
```

Figure 3-5

### *Service Accounts*

When configuring a local service account, it is best to permit only the necessary permissions required to operate the service itself. In the case of Vault, a service account should be created as a system account with no shell, as it should never be used to log into the system. An example configuration is shown in Figure 3-6.

```
$ useradd --system --home /etc/vault.d --shell /bin/false vault
```

Figure 3-6

As service accounts are created for other tools, such as configuration management solutions, their permissions should also follow the principle of least privilege. By default, many of these tools are far too permissive, often permitting full root access of the entire Vault infrastructure from a single host.

### *Local User Accounts*

While SSH keys are the recommended method for access, some organizations may opt to use local user accounts. When local user accounts are used, regular password changes should be enforced. To enforce a password change on a particular service account every 60 days, issue the command depicted in Figure 3-7.

```
$ chage –M 60 –W 7 <user>
```

Figure 3-7

### *Disable Command History*

By default, Linux records each command in the user's *.bash_history* file in the home directory. If Vault operators use SSH to manage the Vault service, sensitive information could be exposed in the command history. For example, credentials used for Vault authentication, secrets used for the configuration of a secrets engine, or the addition of a static secret could undesirably end up in the command history.  Use the command in Figure 3-8 to disable the recording of command history globally.

```
$ echo 'set +o history' >> /etc/bashrc
```

Figure 3-8

### *Disable Core Dumps*

Core dumps create a snapshot of the memory contents in the event an application crashes or exits abnormally. Since an unsealed Vault node maintains the encryption key in memory, a core dump could contain this key or other sensitive data. This scenario is a security concern because any user with access to the core dump could retrieve this data. The commands for disabling core dumps can found in Figure 3-9.

```
$ echo '* hard core 0' >> /etc/security/limits.conf

$ echo 'fs.suid_dumpable = 0' >> /etc/sysctl.conf

$ sysctl -p

$ echo 'ulimit -S -c 0 > /dev/null 2>&1' >> /etc/profile
```

Figure 3-9

## Additional Configuration

### *Log Rotation*

Regardless of which audit device(s) is used for logging activity in Vault, it is a good idea to ensure data is also being saved locally. This practice helps prevent service disruption and

ensure audit data continuity when all other audit devices are unavailable. However, since Vault does not manage log rotation itself, it is crucial to ensure those local log files are being rotated in a timely fashion. Rotating logs will help prevent the local disk from filling up, causing a service disruption. Figure 3-10 illustrates how to install LogRotate, which is one of many tools frequently used for log management.

```
$ yum install logrotate
```

Figure 3-10

Once installed, a new configuration file needs to be created in the directory */etc/logrotate.d*. It should be named after the log file that needs rotating. For example, if the Vault audit log is configured to store data at */var/log/vault-audit.log*, the LogRotate configuration file would exist at */etc/logrotate.d/vault-audit.log*. The cadence of log rotation should depend on the amount of activity in Vault.

```
/var/log/vault-audit.log {
     rotate 7
     missingok
     notifempty
     compress
     size 1G
     daily
     create 0600 vault vault
}
```

Figure 3-11

The LogRotate configuration file displayed in Figure 3-11 will manage the logs using the following actions:

- Rotate logs either daily or after they reach 1GB in size
- Compress logs at the time of rotation
- Do not rotate empty log files
- Begin a new log file regardless of whether the previous log exists
- Keep seven versions of the logs (whether daily or at 1GB)

- Set permissions to the *vault* user each time a new log file is created

## Configuration Management

When building a standard image, operating system configurations should be built using a configuration management tool rather than manual configurations. Using a configuration management tool, such as Puppet, Chef, or SaltStack, facilitates granular changes to operating system configuration without the need to make updates to the base image. These tools establish consistency throughout the base image installation to ensure standardization for existing and new infrastructure alike.

For example, during a typical provisioning process using SaltStack, a host is provisioned using the latest base image. The most current Salt minion is then installed, and a basic Salt minion configuration is added. Shortly after, the latest copies of Salt states are copied to the host, and a *state.apply* is executed locally to apply the necessary configurations and bring the host to the desired state.

Configuration management tools can also help validate the integrity of the Vault binary being deployed to Vault nodes. Using the checksum provided by HashiCorp, organizations can ensure that an unmodified version of Vault is being deployed. An example of a checksum for Vault 1.5.3 can be found on HashiCorp's website (*https://releases.hashicorp.com/vault/1.5.3/ vault_1.5.3_SHA256SUMS*).

## Building a Base Image

A secure and well-defined base image is a critical foundation for building a robust automation process. This image provides a single element where the security of the operating system can be defined, and changes to the operating system can be validated. Once the base image is created, a configuration management tool can manage post-installation and long-term management of the instance.

Fortunately, HashiCorp provides a free, open-source tool called *Packer* to assist with this process. Packer is a tool designed to automate the creation of operating system images that can be used on any platform, from on-premises platforms to the public cloud. Packer is packaged as a precompiled binary that can be easily downloaded to any operating system and immediately executed as-is. Packer uses a simple JSON configuration file to build a machine image according to requirements and specifications. It can create images based

on many sources, such as a newly downloaded ISO or an existing cloud image. Packer provides its users with tremendous flexibility in the image creation process.

For a simple tutorial on using Packer to build machine images, check out the HashiCorp Learn platform (*https://learn.hashicorp.com/packer*).

## Maintaining a Base Image

Once each base image has been created, regular maintenance should be scheduled to ensure the images continue to meet standards as new security and business requirements are developed. While there are many reasons that images should be updated, image maintenance falls into three primary categories:

- **Operating System Patches:** Operating system vendors frequently publish security patches to address vulnerabilities. Image patching should be automated based on a recurring schedule. More critical patches should be addressed sooner on a case-by-case basis.
- **Changes in Business Requirements:** Most organizations have default applications installed across all its workloads, such as security tools, data collection services, or other management tools. These tools should be applied post-provisioning using configuration management. However, updates to applications that cannot be automated should be managed using an out-of-band process.
- **Bug Discovery:** Out-of-band maintenance may be required when security or performance bugs are discovered. This type of image maintenance is often a manual process.

Once the initial process is developed for creating images in Packer, this process should be migrated to a CI/CD tool such as *CircleCI* or *Jenkins*. Integration with a CI/CD tool simplifies image maintenance by automating approved maintenance tasks. For example, a well-defined pipeline can determine when new patches are available, apply them to the image, and validate the install without user intervention.

Organizations can be confident that new workloads are provisioned with an up-to-date image by using this approach. Automation is the key to maintaining base images. Below is a high-level description of what that build process could look like:

1. The base Packer template is created and managed through a repository in a version control system (VCS), such as Git.
2. A user submits a proposed change to the image template through the VCS as a pull request.
3. A CI/CD process automatically picks up the proposed change.
4. The CI/CD process uses the updated Packer template to deploy an existing image, apply the new changes based on the specifications in the Packer template, and create a new image file based on those changes.
5. Afterward, the CI/CD process installs the new image based on the proposed changes.
6. A configuration management tool is used to deploy the configuration changes.
7. The CI/CD process runs a series of validation tests on the virtual machine to ensure that the new image meets the same standards and benchmarks that the previous image met.
8. After all validation tests have been completed, the changes are submitted to a final peer review.
9. Once the final peer review is completed, template changes are merged into the main branch. An updated image is built, and the image is stored in an artifact repository for general use.

## Deployment

When building the provisioning process, the end goal should be a fully automated deployment of a Vault cluster in the desired location specified by the administrator. There are a countless number of tool combinations that can accomplish this to various degrees of completion. Terraform alone could accomplish this goal, but the addition of a CI/CD process can simplify this even further. The following example demonstrates a simple deployment workflow using CircleCI, Terraform, and SaltStack:

1. A CircleCI pipeline is executed to deploy a Vault cluster in the AWS US-West-2 region,
2. The CircleCI pipeline triggers a *terraform apply* to deploy properly sized instances in US-West-2 using the standard base image.

3. Once the instances are running, Terraform triggers a provisioner to install and configure a Salt minion on the instances.
4. After the Salt minion has been configured, Terraform triggers a provisioner to pull the necessary state files from a Git repo and execute a *state.apply*.

This approach has two advantages. First, having formal automation in place will simplify the initial deployment and subsequent deployments to new platforms and regions. This level of flexibility allows the service team to move in tandem with business demands. Second, this automation alleviates the effort involved with tedious operating system patching. When a deployment process is well-built, the support team can create a new Vault cluster in less time than it would take to patch and validate the existing servers manually.

In the long run, less time will be required to build the automation than will be spent manual patching and troubleshooting inconsistencies over the lifetime of the Vault service. After well-defined deployment pipelines are in place, administrators should not require more than an hour to get a Vault cluster up and running. If cluster deployment takes longer than this, continual developments should be made to improve the automation process.

# *Vault Service Configuration*

Now that the operating system is secured and ready to support the Vault service, configuration files must be created for both Vault and the service manager. While the Vault configuration file might be the most critical configuration that comes to mind, the service manager's configuration file is also vital to controlling the Vault service on the operating system.

## Vault Configuration File

The Vault configuration file is a flat text file that configures Vault components, such as the storage backend, listener, seal type, audit device, and more. The configuration file can be formatted in either HCL or JSON, depending on the organization's preference. A similar configuration file is created for each Vault node in the cluster to ensure consistency among the Vault nodes. When the Vault service is configured and started, the configuration file is referenced using a flag within the executable statement.

While the configuration file defines Vault's critical components, it does not configure Vault features such as secrets engines, auth methods, or other features of Vault, which are configured after Vault has been deployed. Table 3-1 lists the features configured using the Vault configuration file compared to the features that are configured within the Vault platform itself.

| Configured in Configuration File | Configured in Vault |
|---|---|
| Storage Backend | Secrets Engines |
| Listeners (IPs) and Ports | Auth Methods |
| TLS certificates | Policies |
| Seal Type | Entities and Groups |
| Cluster Name | Key/Value Structure |
| Log Level | Replication |
| User Interface | Namespaces |
| Cluster IP and Port | MFA |
| Telemetry | Audit Devices |

Table 3-1

The Vault configuration file is composed of multiple parameters to define various configuration options applied to the Vault service. While some parameters are required for Vault to start, several are optional and enable additional functionality. The essential parameters for the Vault configuration are grouped into stanzas and define the primary components of Vault. Each stanza has a set of valid configurations based on the option being configured. Stanzas are written as follows:

```
stanza1 "name" {
     <parameter1> = <value1>
     <parameter2> = <value2>
     <parameter3> = <value3>
}
stanza2 "name" {
     <parameter1> = <value1>
     <parameter2> = <value2>
     <parameter3> = <value3>
}
```

Figure 3-12

The parts of a configuration file may include:

- **storage** – [required] this stanza configures the storage backend
- **listener** – [required] this stanza defines the IP addresses and ports
- **seal** – this stanza specifies the seal type used for seal and unseal operations
- **telemetry** – this stanza configures Vault to publish metrics to upstream systems
- **service_registration –** this stanza instructs Vault to register with the specified platform
- **entropy –** this stanza configures Entropy Augmentation
- Additional parameters such as a cluster name, enabling the user interface (UI), or configuration for high availability

## *Storage Backend*

The storage backend stanza defines where Vault reads and writes its configuration and other information, such as the data written to the KV store. The storage backend configuration is one of the two stanzas that must be declared in the configuration file. Each node within a cluster should have an identical configuration for the storage backend.

HashiCorp Vault supports over twenty storage backends to choose from depending on an organization's preferences and ability to support a platform. In the case of a Vault Enterprise deployment, the storage stanza would use either Consul or raft, depending on the use of a Consul storage backend or an Integrated storage backend. These are the only two storage backends that support high availability and are officially supported by HashiCorp technical support.

**Consul Storage Backend**

In the example provided (Figure 3-13), the storage backend is defined as *Consul*, which is the name of the storage backend being used. To configure the Consul storage backend, Vault needs to understand how to communicate with the Consul cluster to store its encrypted data.

```
storage "consul" {
  address = "127.0.0.1:8500"
  path    = "vault/"
  token   = "1a2b3c4d-1234-abdc-1234-1a2b3c4d5e6a"
}
```

Figure 3-13

As previously discussed, the Consul agent runs locally on each Vault node in client mode, simplifying the connectivity to the Consul cluster. Consul uses TCP port 8500 for connectivity by default. Therefore, the configuration file above instructs Vault to connect to the Consul service using its loopback address and TCP port 8500. The *address* parameter defines this connectivity.

The *path* parameter defines what path Vault should use to read/write data on the Consul key/value store. The path can be configured to an organization's preference, but

deployments frequently use "*path = /vault"* as the default path. This path is also referenced in a Consul policy if Consul ACLs are enabled.

Consul ACLs should be enabled to protect from unauthorized changes and direct modifications when using a Consul storage backend. Even though this data is encrypted, users may be able to connect to Consul and delete the */vault* path, which would be detrimental to the Vault service.

The *token* parameter defines the Consul service token that permits Vault to read/write data in Consul. The token is created in Consul and added to the Vault configuration file.

**Integrated Storage**
For clusters using Integrated Storage as the storage backend, the storage stanza defines parameters specific to the Integrated Storage configuration. Figure 3-14 depicts an example configuration for Integrated Storage.

```
storage "raft" {
  path = "/opt/vault/"
  node_id = "node-a.example.com"
  performance_multiplier = 1
}
```

Figure 3-14

The first parameter for Integrated Storage is the path. This parameter, again, instructs Vault where to read/write data. However, since Integrated Storage stores data local to the Vault node, this path should be a valid and secured path on the local host itself. This path should be available on each node in the Vault cluster to properly replicate data between the hosts for redundancy.

Each node in a raft cluster requires a unique identifier. If a node attempts to join the cluster using a duplicate identifier, raft does not permit that node to join the cluster. For Integrated Storage, the *node_id* parameter defines the identifier to be used within a cluster. The *node_id* value is often set to the node's hostname since the hostname is likely unique within the environment.

The *performance_multiplier* parameter is used to tune raft's performance, including how quickly it detects failures and initiates leader elections. A lower value configures raft to perform these tasks faster but at the cost of CPU and network resources. This value is set to 5 by default, which is generally acceptable for development or test environments. However, production configurations should be set with a value of 1, which is raft's highest-performance mode. Omitting this parameter in the raft configuration results in the default value of 5, so it is vital to define this parameter for a production cluster.
Additional configurations for Integrated storage can be found in HashiCorp's documentation (*https://www.vaultproject.io/docs/configuration/storage/raft*).

### *Listener*

The *listener* stanza is all about communication. It is a required stanza that determines how Vault responds to requests, such as addresses, ports, and TLS behavior. Unlike other stanzas in the configuration file, the listener stanza only has one option: TCP. The Vault TCP listener can be configured to listen on a single interface or multiple interfaces. While there are many parameters available for the TCP listener, Figure 3-15 shows what a listener stanza might look like for a Vault cluster.

```
listener "tcp" {
 address = "0.0.0.0:8200"
 cluster_address = "0.0.0.0:8201"
 tls_disable = "false"
 tls_cert_file = "/etc/vault.d/client.pem"
 tls_key_file = "/etc/vault.d/cert.key"
```

Figure 3-15

The first parameter in Figure 3-15 is the *address* parameter. The address parameter specifies what IP address and port that Vault uses to listen for requests. The default port for Vault is 8200, and from experience, most organizations use the default port for Vault communication. This parameter's value should be set to the IP address of the desired interface to instruct Vault to use a specific interface on the host

For example, if the node is assigned an IP address of 10.0.0.10, the value would be 10.0.0.10:8200. If the Vault node has multiple interfaces, the value can be set as

0.0.0.0:8200 to listen for requests on all interfaces. Using 0.0.0.0:8200 can help standardize the configuration file, but this may be undesirable from a security perspective if additional interfaces are added to the host.

> **Note:** If the address parameter is set to 127.0.0.1:8200, Vault will not respond to external requests. This configuration is a common mistake with new users as 127.0.0.1:8200 is used throughout HashiCorp documentation.

The next parameter in the listener stanza is the *cluster_address*. This optional parameter sets the address for communication with other nodes within the Vault cluster. Like the *address*, *cluster_address* binds this communication to an interface. When this parameter is not defined, Vault defaults to one port higher than the address port. Therefore, this port defaults to tcp\8201, as the default address port is tcp\8200.

The *tls_disable* parameter specifies whether the Vault cluster uses TLS for communication. Vault assumes TLS for secure communications by default, so this parameter is optional unless explicitly disabling TLS. In a larger organization, obtaining TLS certificates is often a grueling task. In such cases, Vault might be initially deployed without TLS certificates to help accelerate Vault's deployment and configuration. While disabling TLS may be undesirable, this use case may be justifiable, assuming certificates are implemented before any data is written or requests are made to Vault.

The *tls_cert_file* and *tls_key_file* parameters are reasonably self-explanatory, but they point Vault to the location of the TLS certificate and the private key so Vault can secure the listener. The TLS certificate should be created by a trusted certificate authority (CA), and a TLS certificate and private key are needed for each Vault node. The certificate is usually requested using the node name, but it may include additional subject alternative names (SANs) depending on the configuration. For example, suppose Vault's front-end load balancer is configured for pass-through. In that case, the certificate will likely have a SAN for the 'friendly' DNS name of the Vault cluster to prevent certificate validation errors on Vault clients.

### *Seal Type*

The *seal* type stanza in the Vault configuration file determines how Vault encrypts and decrypts the master key to read the data stored on the storage backend. When Vault is first initialized, Vault starts in a sealed state: it knows where the data is, but it does not know how to decrypt it. Vault is unsealed if the nodes can decrypt the master key and maintain the encryption keys in memory. Until Vault is unsealed, it cannot respond to any client requests for authentication or secret retrieval. However, Vault can respond to a limited number of queries about its health and the status of the cluster.

The seal type stanza is an optional configuration. Most customers opt to use something other than the default seal type of key shards to provide increased flexibility and automation of the unsealing process. Additional options frequently seen in production include cloud auto unseal options such as AWS KMS, Azure Key Vault, or Google Cloud KMS. For organizations with access to a hardware security module (HSM), Vault can be configured to use the HSM to encrypt and decrypt the master key as another supported option. Each of these options has parameters that apply to the supporting seal type. For example, the AWS KMS seal type has parameters that define the KMS key ID, the region that the key was created in, among others. Figure 3-16 shows what a seal type stanza might look like for a Vault cluster using AWS KMS for unseal operations.

```
seal "awskms" {
  region = "us-east-1"
  kms_key_id = "1a2b3c4d-abcd-1a2b-abcd-1a2b3c4d5e6f",
  endpoint = "example.kms.us-east-1.vpce.amazonaws.com"
}
```

Figure 3-16

Most of the seal types can be used by a Vault cluster regardless of where Vault is deployed. For example, a Vault cluster running in an on-premises data center can still use the AWS KMS seal type, assuming it can access the AWS KMS public APIs. In such a scenario, the Vault cluster would need to authenticate to the AWS service using an AWS access key and secret key. While the "awskms" service provides the optional parameters for authentication, it is generally not recommended to store any credentials in the configuration file to prevent unauthorized access. Instead, use environment variables to set these values. This same

strategy should be used for any seal type where integrated authentication cannot be used, such as Amazon EC2 service roles or Azure managed identity.

Since each seal type has parameters based upon the underlying platform or solution, this section does not go through the individual parameters in the example provided for AWS in Figure 3-16. HashiCorp's documentation should be referenced to understand the required parameters for the desired seal type (*https://www.vaultproject.io/docs/configuration/seal*).

### *Additional Parameters*

Aside from the configurations defined within the stanzas, there are additional parameters that are critical to the Vault configuration as well. These include general node configurations such as enabling the user interface (UI) and setting the log level. High availability parameters defined here include the address to advertise to other Vault servers for redirection (*api_addr*) and request forwarding (*cluster_addr*).

As an example of the additional parameters configured, Figure 3-17 displays several standard configurations seen in many deployments.

```
api_addr = "https://vault-us-east-1.example.com:8200"
cluster_addr = " https://node-a-us-east-1.example.com:8201"
cluster_name = "vault-prod-us-east-1"
ui = true
```

Figure 3-17

The *api_addr* and *cluster_addr* parameters are critical to high availability and are required if Vault is deployed with replication in mind. The value for the parameter *api_addr* is a full URL that identifies the address to advertise for client redirection. The value of this parameter is commonly the DNS name used by clients to access the Vault cluster. This hostname is also the URL the secondary cluster uses to unwrap the secondary token.
The *cluster_addr* parameter value is a full URL that identifies the address to advertise to other Vault servers in the cluster for request forwarding. This parameter's value is frequently the hostname or the IP address of the Vault node.

The *cluster_name* parameter is optional, but it defines the identifier for the Vault cluster. If this value is omitted, Vault generates a random value for the identifier. Often, the value

matches the friendly DNS name or a unique name that simplifies the cluster's identification and the location of where the cluster is provisioned.

For an organization new to Vault, enabling the UI might help visualize configuration, but it is not required. Many security-conscious organizations opt not to use this Vault interface. By default, the Vault UI is disabled. The UI must be explicitly enabled using the *ui* parameter. This parameter enables the UI on the primary interface and port configured on the listener. This parameter accepts a Boolean value and can also be set using the environment variable VAULT_UI.
Additional parameters for the Vault configuration file can be found on HashiCorp's website (*https://www.vaultproject.io/docs/configuration*).

### *Putting It All Together*

Vault requires a configuration file for each node. The configuration file must consist of the required parameters and stanzas along with the optional configurations. Many of these configurations were based on the functionality required to meet the technical requirements.

Figure 3-18 is an example of a complete configuration file based on the configurations discussed in this section. This configuration file assumes a vault cluster deployed on AWS, a Consul storage backend, and ready to be configured for replication. This configuration file can also be found on GitHub (*https://github.com/btkrausen/hashicorp*) and other references throughout the book.

```
storage "consul" {
  address = "127.0.0.1:8500"
  path    = "vault/"
  token   = "1a2b3c4d-1234-abdc-1234-1a2b3c4d5e6a"
}
listener "tcp" {
 address = "0.0.0.0:8200"
 cluster_address = "0.0.0.0:8201"
 tls_disable = 0
 tls_cert_file = "/etc/vault.d/client.pem"
 tls_key_file = "/etc/vault.d/cert.key"
 tls_disable_client_certs = "true"
}
```

```
seal "awskms" {
  region = "us-east-1"
  kms_key_id = "12345678-abcd-1234-abcd-123456789101",
  endpoint = "example.kms.us-east-1.vpce.amazonaws.com"
}
api_addr = "https://vault-us-east-1.example.com:8200"
cluster_addr = " https://node-a-us-east-1.example.com:8201"
cluster_name = "vault-prod-us-east-1"
ui = true
log_level = "INFO"
```

Figure 3-18

## Vault Service Configuration

In a production environment, Vault needs to be configured to be run and managed as a service. Most modern operating systems, such as RedHat and Ubuntu, use *systemd* as the default system and service manager to manage critical services deployed on the server. Although *systemd* is not the only option for managing services, it tends to be the most popular, so this section focuses on using *systemd* to manage the Vault service.

### *Vault User*

Security is always in inherent concern when configuring and managing the Vault service on each node. Running Vault as a non-privileged user reduces risk to the operating system and data by limiting the user's privileges to only those needed to run the Vault service. While each organization may differ, many deployments will use a system user named *vault* to run the Vault service.

```
$ useradd --system --home /etc/vault.d --shell /bin/false
    vault
```

Figure 3-19

While this *vault* user is used to run the Vault service, keep in mind that the user needs access to other folders and files on the Vault node to properly run the Vault service. For example, the *vault* user needs access to the directory where the configuration file is stored,

access to a directory for Vault plugins (if used), and access to a directory or file for writing audit logs.

### *Vault Service File*

Like other systemd services, a service configuration file needs to be created and added to the */etc/systemd/system* directory to create and manage the Vault service. In a Vault deployment, the file is commonly named *vault.service* so the service can be managed as *vault*. This file can be manually created on the node, or it can be copied from an available source during the automated job that provisions Vault.

Figure 3-20 is an example of a service configuration file. This file can be found on GitHub (*https://github.com/btkrausen/hashicorp/blob/master/vault/config_files/ vault.service*)

```
[Unit]
Description="HashiCorp Vault - A tool for managing secrets"
Documentation=https://www.vaultproject.io/docs/
Requires=network-online.target
After=network-online.target
ConditionFileNotEmpty=/etc/vault.d/vault.hcl
StartLimitIntervalSec=60
StartLimitBurst=3

[Service]
User=vault
Group=vault
ProtectSystem=full
ProtectHome=read-only
PrivateTmp=yes
PrivateDevices=yes
SecureBits=keep-caps
AmbientCapabilities=CAP_IPC_LOCK
Capabilities=CAP_IPC_LOCK+ep
CapabilityBoundingSet=CAP_SYSLOG CAP_IPC_LOCK
NoNewPrivileges=yes
ExecStart=/usr/local/bin/vault server -
config=/etc/vault.d/vault.hcl
```

```
ExecReload=/bin/kill --signal HUP $MAINPID
KillMode=process
KillSignal=SIGINT
Restart=on-failure
RestartSec=5
TimeoutStopSec=30
StartLimitInterval=60
StartLimitIntervalSec=60
StartLimitBurst=3
LimitNOFILE=65536
LimitMEMLOCK=infinity

[Install]
WantedBy=multi-user.target
```

Figure 3-20

Once the service configuration file has been created, the Vault service should be configured to start whenever the Vault node has been restarted. To ensure the Vault service starts automatically, use the *systemctl enable* command.

```
$ systemctl enable vault.service
```

Figure 3-21

# *Monitoring and Alerting for Vault*

The classification of Tier 0 is given to services that are critical to an organization's operations. Organizations take special care when designing the performance, reliability, and disaster recovery around these services because of their importance. When organizations adopt HashiCorp Vault, they tend to classify it as a Tier 0 service because of its integration with business-critical applications in their environment. Because its applications rely on Vault's services, the Vault infrastructure requires an equal or greater uptime than the integrated applications.

Critical services such as Vault must be proactively monitored to ensure the highest availability. Enterprise deployments of Vault can be complicated, and there are a variety of aspects that must be monitored. Monitoring within a Vault deployment can be grouped into three general categories:

- Operational Monitoring
- Security Monitoring
- Performance Monitoring

These categories focus on the monitoring of the Vault service itself. General security measures related to elements such as the underlying operating system, infrastructure platform, and identity provider are not included in this section. Some of these topics are briefly discussed in *Operating System Configurations* and *Designing the Vault Architecture*.

## Operational Monitoring

In the context of this section, operational monitoring is defined as how an organization monitors and responds to real-time events within the Vault infrastructure. These events are often related to environmental changes, such as issues with the underlying infrastructure. However, they can also be related to changes requested by Vault operators or users as well.

Environmental changes or failures can wreak havoc on the Vault deployment since the Vault platform's underlying infrastructure is critical to its performance. Overutilized hosts can introduce latency to client requests, hardware failures can invoke Vault cluster leadership changes, and network errors can hinder cluster replication. While there is an infinite number

of possible concerns, their impact on Vault can be monitored using both built-in and third-party solutions. Operators can be alerted when events are logged, and decisive action can be taken to ensure the Vault service's availability.

Many of the events that operators should be aware of are available through the Vault audit logs, Vault telemetry logs, or operating system logs. While these logs are often written directly to the local disk by default, logs should be collected and sent to a centralized log aggregation platform. These platforms can parse through the logs and send alerts or take automatic action based on important events or keywords. Examples of events that should be monitored for operational efficiency include:

### *Telemetry*

- **Duration of Vault requests** - determines how quickly Vault is responding to client requests
- **Write-ahead logs** – ensures that cluster replication is functioning as expected
- **Vault leadership changes** – may indicate an underlying problem with the operating system, host, or platform
- **Seal status of Vault** – may indicate recent service or host failure
- **Raft leadership changes** – may indicate a problem with the storage backend if using Integrated Storage (Consul has telemetry if using Consul)
- **Audit log failures** – may indicate a local disk problem or issues connecting to a remotely configured Syslog server
- **Resource quota violations** – could indicate abuse of the service by a user or application

### *System Logs*

- **License expirations** – may indicate an upcoming renewal requirement. Vault Enterprise stops the agent if the license expires
- **Consul logs** – could point to Consul client or cluster problems

### *Audit Log*

- **Updates to Vault policies** – could indicate additional and unwarranted permissions for tokens
- **Transit key deletions** – could indicate an accidental or nefarious key deletion
- **Transit minimum decryption version configurations** – might indicate a potential problem with decrypting older data, if it exists.
- **Number of active tokens** – this can sometimes reflect overuse or incorrect use of tokens by a user.
- **Number of secrets created** – long-term trends could indicate a potential abuse of the service

## Security Monitoring

HashiCorp Vault is a security platform. Its ability to store, generate, and decrypt sensitive data makes Vault a prime target for bad actors. While every critical piece of an organization's infrastructure should be adequately secured and monitored, Vault's unique capabilities demand that exhaustive security practices and monitoring be enforced for the service. Nefarious actions may not only come from beyond the walls of the organizations; internal threats can pose significant risks to organizational security as well. While Vault operators and users should be trusted entities to the business, detailed auditing, logging, and alerting should be enabled to ensure actions within Vault fall within acceptable guidelines. Any actions falling outside of the defined threshold should trigger an alert for the security and operations teams.

The first line of defense for organizations is to follow the principle of least privilege. Vault administrators, operators, and users should only be granted the permissions that align with their responsibilities. Permissions granted beyond these roles often invites curiosity and undesirable Vault activity. For example, Vault operators with excess permissions might create a root token as a test but forget to delete it. The result is an unmanaged root token with unlimited access that could be accidentally shared or unintentionally obtained by a bad actor. Without proper security monitoring and alerting, the Vault administrators or security team would never know about this substantial risk to the environment.

The simplest way to start auditing actions in Vault is to enable Vault audit devices. By enabling one or more audit devices, all authenticated interactions to Vault are securely logged for future inspection. As these audit logs are sent to a data aggregation tool, alerts should be configured for specific actions that could negatively impact Vault's security. Often, these security logs are also sent to a SIEM for correlation with other logs, such as logs from the public cloud provider or a CI/CD tool. Detailed auditing information can be found in the chapter on Audit Devices. Some actions that should be closely monitored are:

- Use of a root token
- Creation of a new root token
- Vault policy modification
- Transit key deletion
- Enabling a new auth method
- Modification of an auth method role
- Creation of a new auth method role
- Permission denied (403) responses
- Creation of a new token
- Use of Vault by human accounts outside of regular business hours
- Vault requests originating from unrecognized subnets

While HashiCorp Sentinel can restrict offending clients' actions, the evaluation of denied requests may help organizations understand where internal and external threats may exist.

## Performance Monitoring

In addition to monitoring for operational concerns, performance must be monitored to ensure the Vault infrastructure meets the organization's needs. Standard metrics (CPU, memory, disk) are a critical aspect of performance monitoring, and organizations should define baselines based on the average workload. Peak workloads should also be identified and taken into consideration when monitoring an environment's performance.

Different Vault components have varying requirements for foundational resources such as CPU, memory, and disk performance. Vault clusters performing a larger number of encryption or decryption operations have a higher requirement for CPU processing than

clusters that are merely consolidating secrets. Organizations should provision Vault using HashiCorp's recommended reference architecture and adjust based on the current workloads. (*https://learn.hashicorp.com/tutorials/vault/ reference-architecture*)

In addition to the workloads generated by the integrated applications and end-users, performance requirements may differ depending on the architecture, namely the storage backend. Vault deployments using Consul must closely monitor memory usage on the Consul nodes, as Raft depends on having sufficient memory to store data. HashiCorp once shared that over 80% of HashiCorp support's SEV1 cases involved Consul running out of memory. On the other hand, Vault's Integrated Storage feature requires high-performing disks since it writes all data directly to disk. Organizations deploying Vault using Integrated Storage should be sure to allocate disks that can support high read/write performance while continuing to meet other minimal requirements.

Vault telemetry data can also be used as a measurement of how Vault is performing. Metrics such as Vault's response time (*vault.core.handle_request*) can be used to measure the time it takes Vault to respond to a client request. An increase to that time may indicate resource contention or other problems within the Vault infrastructure. All these metrics can be easily visualized by sending telemetry data to an upstream platform. Custom dashboards can also be created to create a general overview of health for the Vault environment.

Performance metrics that should be monitored for a Vault environment include:

- CPU, memory, and disk
- Storage backend transactions
- Disk capacity
- Response time for requests
- Consul memory and persistent storage
- Integrated Storage last contact
- Audit log request/response failure (*a non-zero value indicates a problem*)
- Policy operations (*more complicated policies take more time to parse*)

To find more information about specific telemetry metrics that Vault supports, make sure to look at the official telemetry documentation on the HashiCorp website (*https://www.vaultproject.io/docs/internals/telemetry*)

HashiCorp has additional documentation on Vault and Consul performance, both as a white paper and its Learn platform. These resources are invaluable to a Vault operator and should be reviewed before deploying Vault into a production environment.

*(https://learn.hashicorp.com/tutorials/vault/monitoring)*
*(https://learn.hashicorp.com/tutorials/vault/monitor-telemetry-audit-splunk)*

There are also many HashiCorp community members, such as Martez Reed, who have written blog posts about monitoring and alerting as well. (https://www.*greenreedtech.com/detecting-hashicorp-vault-root-token-generation)*

## *Tracer Bullets*

A tracer bullet is a projectile that uses a small pyrotechnic charge to leave a visible trail indicating the ammunition's flight path. The trail left behind is intended to help determine the projectile's accuracy and adjust the firing mechanism accordingly.

This same concept can be leveraged in monitoring and improving shared services. With shared services, the intent is to mimic a typical transaction of a client consuming the service to collect data about that type of transaction. For example, a support team creates a Python script that does the following:

1) Authenticate with Vault and retrieve a service token
2) Record elapsed time
3) Use the service token to write a secret to a K/V secrets engine
4) Record elapsed time
5) Use the service token to retrieve that secret from the same K/V secrets engine
6) Record elapsed time
7) Store timestamp data along with response codes in a JSON object

This script can be automated and scheduled to run as frequently as needed. The data from this job provide critical insights into the health of the service. Not only is the support team aware of whether the service is running, but they are also aware of ***how*** the service is running.

Why is this beneficial in optimizing the service? Assume all users at an organization are only using the K/V secrets engine. The support team has collected a lot of data on the health of the service using tracer bullets. One day, the support team decides to enable and allow the use of the Transit secrets engine. From the moment the Transit secrets engine is enabled, there is a marked difference in completion times across all transactions. As a result of the data the support team has gathered, they can quickly see the impact on the service due to enabling this new functionality. This information arms the support team with information to make educated decisions on adjustments to the Vault service to ensure performance can be realigned with expectations. These changes may require adding resources to the cluster, separating Transit from K/V, or other adjustments.

## *Automating Vault Operations*

Change is a necessary and frequent element of delivering applications requiring an exceptional user experience. Before the DevOps model was widely adopted, change was handled through laborious change control or change management processes. These generally involved weeks of development, manual testing, and change approval processes before even minor adjustments could be implemented. These tedious processes impeded an organization's ability to transform the way they do business.

The rise of DevOps and Agile practices has brought a renewed focus on reducing delivery time while increasing reliability with automation tools. These practices operate under the theory that the approval and validation of changes can happen as changes are built, increasing the volume and frequency of enhancements. Moving to such a model allows organizations to introduce changes more often since unit testing can validate changes before implementing them. This process enables organizations to move faster to keep up with market demands.

Once integrated with business-critical applications, these same methodologies must extend to the Vault shared service as well. Defining a baseline using a repeatable framework, such as Terraform, can reduce operational overhead and human error for the initial Vault deployment. After the service is deployed, operational tasks should be performed using a configuration management tool or infrastructure as code.

## Tools for Success

The creation of a pipeline is an essential part of software development. The primary objective of the pipeline is to execute a set of steps for a given objective, such as deploying a new application on a Docker container. Many different pipelines often exist within an environment, each supporting a critical function within the software development lifecycle.

Using a pipeline can provide the following benefits:

1. Eliminate the need for human beings to perform mundane tasks repeatedly.
2. Eliminate the opportunity for human error that could potentially compromise a service.
3. Ensure that a specific checklist is followed each time the process is run.

When adopting a pipeline for infrastructure-related automation, a standard process should be established. This process will become the basis for all required pipelines with the goal of automating the organization's change control process. Approval processes can be easily integrated into the process, enabling change control personnel to be involved as changes are happening. This strategy can significantly reduce the time required to deliver a change and provide change tracking for all changes. *Jenkins* or *CircleCI* are examples of tools that can be used to automate the individual steps of each job.

As an example, a base-level process might include the following steps. The objective is to map out what the process would look like if it were to be performed manually and then convert it into an automation job in Jenkins.

1. A site reliability engineer (SRE) clones an internal Git repository.
2. The SRE modifies the configuration file in a branch of that repository.
3. The SRE submits this change in the form of a pull request.
4. A Jenkins job configured to look for pull requests to this branch notices that a pull request has been submitted.
5. The Jenkins job creates a virtual machine, applies a base service configuration, and initializes the service on the new host.
6. The Jenkins job runs unit testing tests against the service to gather a baseline for the current state.
7. The Jenkins job then applies the proposed changes based on the submitted pull request, restarts the service, and runs the same panel of tests against the changes.
8. If all tests complete successfully, the pull request waits for a peer review to be performed by a fellow SRE. If any of the tests fail, the submitter can look at the results and make further changes before submitting an update to the pull request.
9. Any unneeded resources are cleaned up.
10. The peer SRE will look at the changes being made in Git and the data generated by the Jenkins job to ensure that the desired result was reached.
11. If the final peer review is successful, the peer SRE will approve the pull request and merge the changes into the specified branch in the repository.
12. The change is immediately applied to the environment associated with the repository branch.

During pipeline execution, multiple checks should be enabled to validate the desired configurations, such as:

- Syntax
- Security Practices
- Access Credentials
- Scheduled Change Freezes
- Change Control Approvals
- Resource Cleanup

A well-defined pipeline can ensure services are deployed quickly and uniformly each time the job is run. Organizations should ensure these processes are built before the service is officially launched, even if it results in a slight delay to production release.

## Automated Deployments

The first phase of the operations lifecycle where automation is adopted is the initial deployment. This phase involves building out configuration and supporting scripts that will be used to provision the base infrastructure that will support the Vault service. The resulting artifacts will be stored in a VCS repository. Some of the most common tools used for this phase include:

- Terraform (HashiCorp)
- CloudFormation (AWS)
- Resource Manager (Azure)
- Cloud Deployment Manager (Google)

Note that while all these tools are widely adopted, Terraform is currently the industry-standard since it is a cloud-agnostic tool explicitly built for multi-cloud infrastructure deployments. However, many of the configuration management tools available today, such as SaltStack and Ansible, can also facilitate cloud provisioning in some form.

Once the tool of choice has been selected, the deployment process should always begin with the code defined in the version control system. The code and configurations should be

executed through a pipeline and use the base, as discussed in the Operating System Configuration section. The pipeline should contain various tests to ensure the integrity of changes. This deployment process should be limited to deploying the base image and a limited set of packages. Most of the configuration tasks should be managed in the configuration management phase.

Organizations may need to deploy infrastructure across many different data centers or cloud providers. When building the pipeline, the objective should include extensibility into the pipeline so that it is configurable based on the desired target location and environment type. For instance, the deployment pipeline should allow a user or process to easily deploy the service in the US-West-2 development environment or the US-East-1 region for the production environment. This flexibility should be provided using dropdown menus or command-line flags.

If the pipeline and related processes have been built appropriately, it should be simple for service administrators to redeploy a service when changes or upgrades are needed. This strategy is common in organizations using the concept of immutable infrastructure, where a "known good" resource is deployed rather than attempting to troubleshoot or upgrade existing resources.

For example, suppose a Vault cluster runs in the US-West region of AWS, supported by a front-end load balancer to facilitate service access. Without warning, a new vulnerability is discovered that will negatively impact the security of the service. In the past, the process to patch the Vault cluster would have involved taking down each host one-by-one, installing the patch, and bringing it back online. This process could take hours and posed risks to Vault service availability.

By using a pipeline, the process would now shift to a much simpler approach. The base image would be patched, and new Vault nodes would be deployed with the updated image. As the new nodes come online, the existing nodes can be decommissioned. The load balancer will automatically direct traffic to the new nodes based upon the configured health checks. This process would be similar when requiring updates in a containerized environment. The Docker image would be updated, a new pod would be deployed, and network configurations would be updated to point production traffic to the new pod.

## Configuration Management

Configuration management is at the heart of the operations lifecycle, as this is where most of the day-to-day service administration occurs. The services provided by configuration management tools are commonly triggered after the initial deployment of the underlying infrastructure and base image. Not only do these tools apply the initial configuration, but they also ensure a consistent state for these configurations during the lifecycle of the service. There are many tool options available in this category that offer a variety of features for supporting services and underlying infrastructure, such as:

- Ansible
- Puppet
- Chef
- SaltStack

Features within these tools that maintain infrastructure and services in the desired state should always be leveraged. For example, SaltStack has a feature known as a *highstate* that enforces the desired state of a particular host or service. Highstate can manage the version and configuration of installed software on the host. If an administrator user logs into a host and initiates a manual change, the changes will be automatically reverted on the next run of the highstate.

Using features such as *highstate* are critical to the security and continuity of the Vault nodes. The following workflows should be created to manage the underlying operating system for Vault nodes:

- Set Root/Admin Password
- Create Local Accounts
- Network Configuration
- Install/Update/Configure Support Packages (Logging, Monitoring)
- Configuration Change Validation

The following configuration management jobs can also be created to manage the Vault service as well:

- Vault/Consul Service Account Creation
- Vault/Consul Service Directory Creation
- Environment Variables
- Installation/Configuration of Vault/Consul Binaries
- Vault Initialization
- Configuration Change Validation
- SSL Certificates
- Storage Snapshots

In some cases, these jobs can be executed in succession for increased efficiency. In other cases, the jobs may need to run on their own. It is generally recommended to start with smaller jobs and then integrate them where possible. This strategy is more efficient than having to split up larger jobs by avoiding unnecessary and complex work.

In containerized environments, containers usually receive any necessary configuration upon deployment. If the configuration needs to be updated, the container is often redeployed with the correct settings. With non-containerized environments, it is common to update the configuration even after initial deployment regularly.

## Service Validation

Once the Vault service is up and running, several pipelines should be created to support and validate the day-to-day operational tasks required for the Vault service. Automating these support tasks ensures that proper validation is occurring the same way each time. These jobs are generally built and managed by using a VCS and a CI/CD tool.

Although there are many great candidates for automated validation, there are four primary aspects of the service that need to be validated regularly.

### *New Build Validation*

When new Vault and Consul deployments are provisioned, the operations team must define a production-ready state, meaning what a production-ready cluster should look like to the organization. A new build validation pipeline can validate the new builds against this state to determine when the builds are ready for production use. At a high level, the pipeline should

be triggered after the Vault binaries are installed on a host, and the production configurations are applied using configuration management. This job should validate production use cases and benchmark tests and notify the team that the new deployment is ready.

### *Service Benchmarking*

Performance and load testing are essential to ensure different components of the Vault environment can handle the volume of estimated requests. These types of tests are commonly performed before integrating production workloads. However, an organization may need to run tests before making substantial changes to the Vault environment, like moving to a different storage backend or changing the resources of the Vault nodes. Running regression tests against Vault can be performed within a pipeline designed to run benchmarking tests against the Vault service in a specified scenario. For new environments, the pipeline can deploy a specified build of Vault, apply the Vault configurations, and perform load testing the various features and functionality.

### *Feature Validation*

Beyond performance-related testing, a Vault operation team often needs to test new features against current production configurations. These new features may be a result of a new internal use case, or it could be to test a newly released feature. For example, if an application team wants to encrypt data before being written to the database, the Transit secrets engine could be provisioned for testing purposes. Enabling this functionality can be done by a pipeline to quickly launch a new environment for configuration validation or benchmarking tests against the new Vault feature.

### *Data Restoration*

Throughout the lifecycle of a Vault environment, disaster recovery plans should be tested frequently to ensure they are working as intended. Rather than manually performing this work, a pipeline can deploy a new environment using the latest production data snapshot. Once the new environment is up and running, validation tests can be performed to ensure the data is valid and available. Such a pipeline enables service operators to validate snapshots quickly and frequently.

## Incident Management

Technology or software components deployed within an organization can encounter operational issues that risk impacting the availability of the service. Regardless of whether the change is directly related to Vault or environmental changes, organizations need to detect and resolve operational issues quickly. Automating operational monitoring should not only be a reactionary process. It is just as critical to detect and solve potential issues to resolve service disruptions proactively.

Automating Vault incident management might include a workflow such as the following:

1. An anomaly is detected for the Vault service
2. An alert is triggered in the incident management tool
3. An incident ticket is automatically created
4. An incident resolution job is triggered
5. If the anomaly is cleared, the alert is cleared inside the incident management tool, and the incident ticket is closed automatically.
6. If the anomaly is not cleared, the alert is escalated. This escalation may be an additional incident resolution job, or an alert sent to support staff.
7. Once the issue is resolved through automation or human intervention, the alert is cleared, and the incident ticket is automatically closed.

The two components for incident management automation include a tool for incident management and another for incident resolution. For incident detection, tools such as OpsGenie, PagerDuty, or Iris, can be used for receiving alerts from monitoring/telemetry tools and routing them to the appropriate resolution tool.

For incident resolution, a pipeline can be triggered by the incident detection tool. Automation tools that include an API are excellent additions to such pipelines. Such automation tools include:

- Jenkins
- CircleCI
- Rundeck
- SaltStack

# *Support Documentation*

Before the Vault service is launched, organizations should ensure that proper documentation is created and readily available. This documentation is just as critical as the operational processes or service health monitoring because it reduces support overhead and service incidents.

The following are rules to follow for anyone who wants to write effective documentation:

1. ***Map It Out:*** It helps to outline the flow of the document and the specific subject matter before creating the documentation. A brainstorming exercise can ensure the documentation stays on topic and addresses as much of the user base as possible.
2. ***Keep It Simple:*** The documentation should be easy to consume. Build the documentation with novice readers in mind to ensure the best possible coverage for all users.
3. ***Be Concise:*** Be sure to stick to the topics at hand. Try not to get into the "technical weeds" unless there is additional information that must be provided.

Complete Vault documentation will include detailed information about the following topics:

- Onboarding to Vault
- Interacting with Vault
- Integration with of applications and services with Vault

Once it has been created, it is common for documentation to go out of date, especially in a fast-paced and growing environment. Internal business processes, Vault features, and Vault best practices can all change as time passes. Organizations should follow a regular schedule to ensure their documentation stays up to date. As a general rule, schedule documentation updates every six months to ensure that it stays relevant.

## Onboarding Guides

An onboarding guide should be created to provide users with a clear idea of how they can begin using the Vault service. The audience of the Vault onboarding guide will determine

what information to include in the guide. For example, a support technician would need different information from an onboarding guide than a developer would.

When creating this document, the entire process should be mapped out, including authentication and secret retrieval. The simplicity of the document can determine how much effort will be required by Vault operators to respond to simple onboarding questions and requests. This document should include:

- A brief description of the Vault service and the value it can provide
- Links to the SLA and associated SLOs
- An outline of the onboarding process
- Links to the Quick Start Guide, Best Practices Guide, and code examples
- Estimated turnaround for onboarding requests
- Methods of communicating with the Vault support team

Along with the onboarding documentation, it may also be beneficial to include a Service Level Requirements (SLR) document. The SLR document allows the Vault user to provide specific details relating to the Vault use case. These details help users define their Vault integration plans. This information can also prove to be valuable to the Vault operations team to better understand the needs of the business.

## Best Practices Guides

As users begin learning how to consume and interact with Vault, providing "best practices" will improve user efficiency and reduce support workload. Best practices may differ between organizations, but they should always include internal standards that users can follow to improve the Vault experience for all involved. While it is impossible to provide a comprehensive list in this section, below are some general best practices topics to consider for Vault consumers.

### *Reducing Footprint*

Encourage users to consider their use case's impact on Vault before implementation. Use cases that may seem simplistic can potentially generate tens of thousands or even millions

of secrets and associated leases. Managing this amount of data can have a significant impact on Vault and the storage backend.

### *Managing TTLs*

While requesting an extended long time-to-live (TTL) for secrets may simplify application integration, unnecessarily long TTLs are a common factor in service degradation. Vault must maintain each secret and associated metadata in storage until it expires and is garbage collected. The more secrets and leases that Vault needs to manage, the higher the impact on the underlying disks supporting Vault. Long TTLs are a stealthy antipattern because most users are unaware of the impact they can have. This negative impact makes it essential to provide guidance for TTL management.

For example, consider the scenario of a new team being onboarded to Vault. The team configures a new Jenkins job that runs every five minutes. The Jenkins job authenticates with Vault and retrieves a token to be used on each run. Rather than defining a token TTL, the default TTL is accepted. Over the course of 32 days, close to 10,000 tokens are generated and are remain unused in Vault. In this scenario, there may not be a noticeable impact on Vault if there is only one team involved. However, as more teams begin adopting the service, the impact on Vault becomes significant.

### *Managing Connections*

Despite what is widely considered best practice, some developers may decide that connections to the Vault API need to persist using an HTTP keep-alive. Using HTTP keep-alives can cause service disruptions for all other Vault users. Each connection should be stateless, and Vault operators should encourage consumers to ensure they are appropriately closing out each request they make to Vault.

Provide guidance on how to consolidate connections to Vault where possible. For instance, if 100 nodes comprise an application and those nodes are connected to a Salt master, use the Salt master for secrets retrieval rather than reaching out to Vault separately across 100 nodes. If retry logic is used in the code responsible for retrieving a secret from Vault, ensure that the developers know that interval between each subsequent retry should increase to prevent overloading the service.

### *Coding for Reliability*

Integrating with the Vault service should be as simple as possible. However, Vault environments may include multiple clusters or environments to provide high-availability and disaster recovery. It might be necessary for Vault users to implement additional functionality or code to improve the reliability of secrets retrieval if a primary or performance replicated cluster fails.

As applications or services interact with Vault, they require workflows to authenticate with Vault and retrieve a secret. The concept of coding for reliability adds functionality so the application or service is aware of multiple Vault clusters that can be contacted to retrieve a secret.

A detailed guide on this concept can be found in the *Coding for Reliability* section.

### *Segregate Actions*

Vault provides a robust auditing system that allows administrators and security personnel to track Vault consumption. The audit system can be leveraged to ensure that each action taken with a secret is associated with a single access token. Following this method helps ensure that each action can be traced to a particular Vault client. Additionally, separating actions can simplify debugging processes when troubleshooting issues with secrets.

## Code Examples

Provide the developers with documentation that includes examples in all relevant programming languages. The code examples should include code snippets for authentication, retrieving the token from the response, and using the token to retrieve a secret.

For example, consider an organization that uses SaltStack for configuration management and Java for customer-facing applications. The Vault operations team can provide code examples for Python and Java to its internal development teams.

An example of this can be found in the *Coding for Reliability* section of this book.

## Quick Start Guides

New users of Vault looking for an initial guide to the tool can find information and hands-on exercises available on HashiCorp's Learn platform (*https://learn.hashicorp.com*). However, in many cases, a typical user may not be interested or have the time to invest to view all the content on the Learn platform. Therefore, creating an internal, custom quick start guide can help users get the information needed to get started with Vault. When creating a quick start guide, be sure to include information about how to authenticate with Vault, navigate the Vault path system, and the basics of how to interact with the secrets engines.

This document should include:

- Configured authentication methods and how to authenticate with Vault
- The purpose of the Service Token and how to use it
- Defining Service Token TTLs that align with user or application needs
- Writing and reading a secret based on the organization's commonly used secrets engines
- Links to the Best Practices Guide and code examples

This document should not take longer than about 20 minutes for the user to consume before being able to interact with Vault.

## Namespace Management

For those organizations allowing teams to manage their namespaces within Vault, it is valuable to create a guide like the Quick Start Guide that helps teams become familiar with low-level administrative tasks. These tasks include:

- Enabling and managing auth methods
- Enabling and managing secrets engines
- Creating security policies

It is highly recommended that a "train the trainer" program be implemented for this, allowing the support personnel to instruct these people directly on managing a namespace through a hands-on approach.

## Frequently Asked Questions (Optional)

As the Vault service matures in an organization, it is sometimes helpful to curate commonly asked questions into a single document or knowledge base. This document can sometimes give users quick answers to some of the most common support issues, solving their problems before needing to reach out to the support team.

## *Executing the Deployment Plan*

As one of the founding fathers of the United States, Benjamin Franklin once said, "By failing to prepare, you are preparing to fail." While this statement applies to any project, it is especially true for organizations deploying Vault considering how many components are involved. Without a proper deployment plan, organizations may find that their Vault deployment is stalled or even unsuccessful. The deployment plan must be created carefully and include clearly defined goals and delivery dates.

A well-defined deployment plan includes many tasks, ranging from the initial preparation phase to the launch of the Vault service. Some additional tasks will even be defined after the launch date, such as new integration, technical support, and maintenance requests. Vault deployments are broken up into four distinct phases:

- Preparation Phase
- Build Phase
- Deploy Phase
- Launch Phase

Each of these phases includes critical tasks required to move to the next deployment phase. A realistic launch date should be identified, and timelines to complete each phase should be outlined. An example timeline can be found in Figure 3-22.

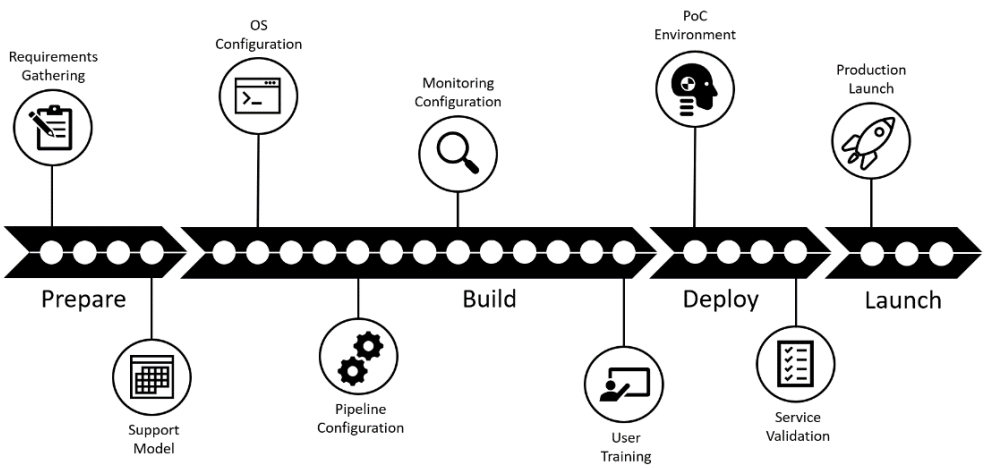



Figure 3-22

## Prepare

There are a few essential tasks that must be completed to prepare for a Vault implementation project. The tasks in this phase ultimately define the strategy for the remaining phases of the project and set the stage for the buildout of the service. Tasks in this phase include:

- Requirements Gathering
- Defining SLO, SLA, and SLI
- Choosing a Support Model
- Designating a Team for Operational Support
- Vault Architecture and Definition

Overall, this phase can take around one month to complete, with most of the time being consumed by requirements gathering and architectural design.

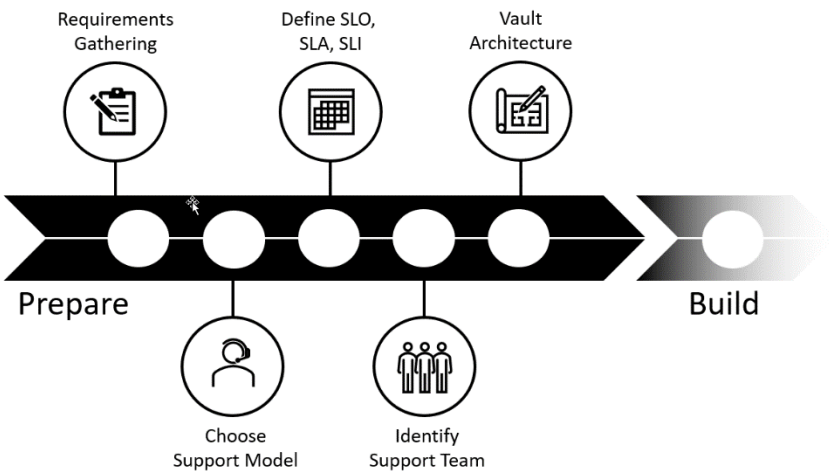



Figure 3-23

## Build

Now that the organization has a clear definition of the project, engineers can begin preparing and building the environment that directly supports Vault. Although there are many infrastructure-related tasks in the build phase, engineers should recognize that there

are many ancillary technologies and processes that contribute to a successful Vault deployment. Some of the supporting tasks may include creating Git repositories, designing automation pipelines, creating auto-healing workflows, and the configuration of monitoring and log aggregation solutions.

Tasks that may be incorporated into this phase include:

- Configuring Git Repositories
- Operating System Configurations
- Creating a Hardened OS Image
- Service Configurations
- Configuration Pipelines
- Deployment Templates
- Deployment Pipelines
- Monitoring and Logging Configuration
- Auto-Healing Jobs
- Terraform Development
- Initial User Training

During the build phase, keep in mind that many of the tasks can be worked simultaneously, as they are likely to involve specialists throughout the organization. While this phase could take upwards of two months, both the quantity and quality of work impacts the ability to deploy, launch, and support the environment going forward.

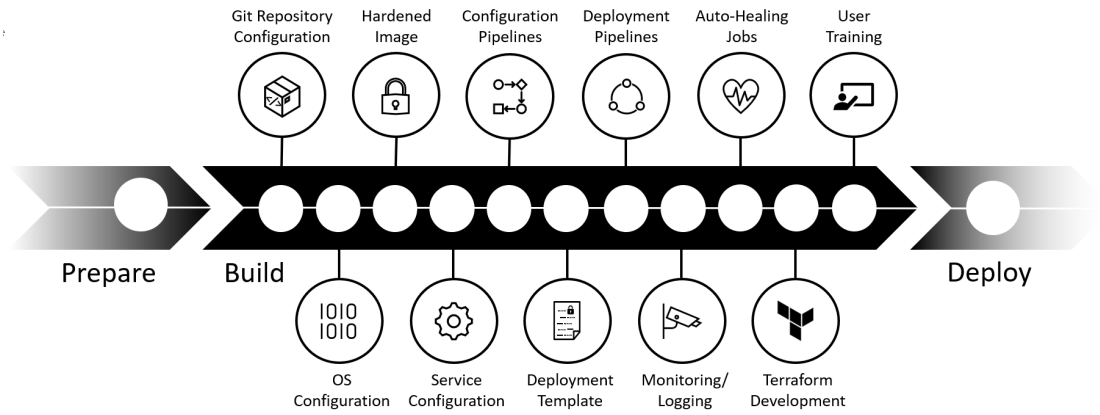



Figure 3-24

## Deploy

When preparations are complete and all environments and supporting components are ready, the initial Vault environment can be deployed. In almost every organization, Vault is first deployed in a proof of concept to validate the work completed in the build phase. Organizations should look for validation to ensure that integrated components are functioning as intended, and the Vault environment can successfully be deployed. Once the proof of concept has been completed, organizations need to provision and configure the infrastructure required for each Vault environment. This deployment should include a development environment at a minimum, but it might be as involved as deploying multiple clusters for production across data centers or public cloud regions. The unique deployment for each organization, of course, depends solely on the initial Vault architecture.

Some of the tasks that may be involved in the deploy phase could include:

- Deploying Proof of Concept
- Provision Infrastructure for Each Environment
- Configure Vault
- Service Validation

When the proof of concept phase is complete, most organizations start deploying Vault in development and test integrations within the development environment itself. Once integrations are proven, organizations can move up the software development stack until

they deploy the production environment. Depending on the complexity of the Vault environment, this phase could take upwards of one month to complete.

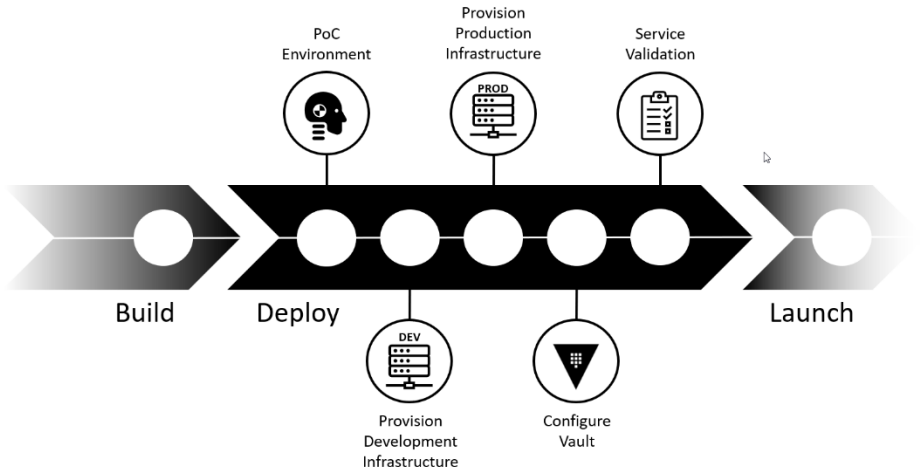



Figure 3-25

## Launch

All the hard work up until this point can now be put to the test. Now that Vault is up and running in production, the initial application and end-user onboarding can commence. Regardless of the use case, secrets can now be migrated to Vault, applications can generate dynamic credentials for required resources, and Vault can begin its journey to becoming an integral part of the organization's identity and security processes. In addition to the initial onboarding, the support team should be fully trained to provide technical support to developers and other end-users, especially during the initial launch. Providing some initial user training ahead of the launch can significantly reduce the initial support workload.

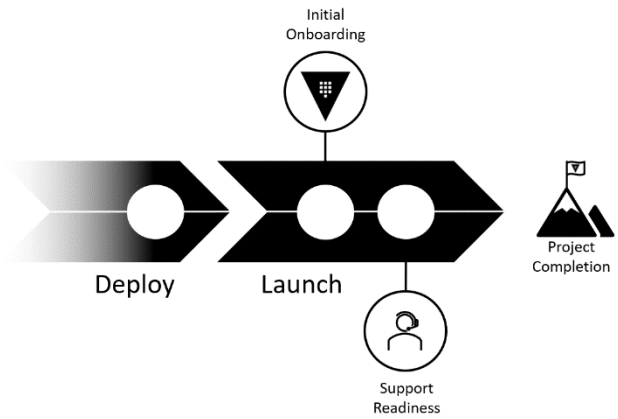

Initial
Onboarding
Deploy
Launch
Project
Completion
Support
Readiness

Figure 3-26

# 4 Configuring Vault Components

***This Section Covers:***

- Auth Methods
- Secrets Engines
- Vault Policies
- Audit Devices

This hands-on chapter focuses entirely on Vault's core functionality in practice. Official HashiCorp documentation does an excellent job explaining every knob and lever on every secret engine or auth method. However, this chapter will provide readers with a review of the basics, along with some valuable "hows" and "whys."

# *Auth Methods*

Vault uses auth methods to validate authentication requests from clients. These requests may come from a user logging into the UI or an application making an API request to generate dynamic credentials. Regardless of the interface or origin of the request, data within Vault is not accessible without a proper authentication request or a valid token. Authentication requests are verified with either an internal or external identity provider. After Vault has verified the supplied credentials against the identity provider, Vault issues the client a token associated with the configured policy. Note that auth methods provide *authentication*, while the policy attached to the resulting token provides *authorization*.

Vault supports various auth methods out of the box to meet many technical and organizational requirements. The auth methods that an organization selects depend on multiple factors, including the architectural or organizational goals for Vault, the environment where Vault will be used, and what existing processes and systems are in place for identity management. For example, if an organization has consolidated its user identity management in Microsoft Active Directory, it might make sense to enable the *LDAP* auth method for user or application authentication.

Multiple auth methods can be enabled or disabled as needed depending on the business and technical requirements. Organizations tend to use multiple auth methods based upon the use cases identified for Vault. Multiple auth methods can be used to provide access to the same data in Vault. The decision of which auth method to use depends on who or what needs access to the data. At a minimum, most customers enable an auth method for user (human) access to Vault and one or many auth methods for application or host (machine) access to Vault.

## Common Auth Methods

Vault provides a variety of auth methods so that organizations can use the one that makes the most sense for their use case. Some auth methods tend to be more popular due to the widespread adoption of their related technologies in modern organizations. For example, the growing adoption of public cloud platforms has resulted in the popularity of their corresponding auth methods in Vault deployments.

This section focuses on the following popular auth methods and how to use them. Make sure to check HashiCorp documentation to understand the other auth methods available to use (*https://www.vaultproject.io/docs/auth*).

- AppRole
- Cloud Provider (AWS, Azure, GCP)
- JWT/OIDC
- LDAP

## *AppRole Auth Method*

The AppRole auth method allows non-human entities to authenticate with Vault using custom roles defined within Vault. AppRole is the ideal authentication approach for machine-to-machine authentication due to its open design and ability to handle a large number of apps. A unique role can be created for each application. This role defines the Vault policy, which permits the access needed for the application.

Each role configured in Vault consists of two main components: the RoleID and the SecretID. These components can be viewed as being synonymous with a username and password, but for machines. When applications are authenticating to Vault, both RoleID and SecretID are sent as part of the authentication request. If the request is valid and successful, Vault returns a service token. This service token is what the application uses to make subsequent requests to Vault to read a secret, generate a dynamic credential, or encrypt data. The RoleID and SecretID are merely used as the authentication method with the ultimate goal of obtaining this service token.

When configuring AppRole, each unique application is generally defined as a different role. Each role created in Vault is configured with specific settings as defined by the requirements for the application. These settings include the role name, the time-to-live (TTL) for the SecretID and tokens, the policy to be assigned to tokens, and even IP CIDR blocks that can use the role. Once the role has been created in Vault, the RoleID can be generated. The RoleID is used for all machines or instances that support an application. The RoleID is synonymous with a username and is not necessarily deemed sensitive data. As a result, the RoleID is often provided using a configuration management tool or baked into an application deployment process, such as in a Docker image.

The second component of AppRole is the Secret ID. The SecretID is synonymous with a password and is not intended to be shared or easily accessible. Although the RoleID remains consistent across application instances, the SecretID tends to be more dynamic and is generated at build time, typically by a CI/CD pipeline or configuration management tool. Each SecretID can be customized in terms of the TTL or usage limitations to better align with each application instance and purpose.

Although each organization is different, the workflow to enable and use the AppRole auth method is similar.

1. Enable AppRole in Vault
2. Create the Vault policy as defined by the application requirements (or use an existing policy)
3. Create the role with associated policy, TTL settings, and other configurations
4. Generate the RoleID
5. Add the RoleID to configuration management, build process, or write to application image
6. Generate SecretID during the build process and deliver to the application
7. The application uses RoleID and SecretID to authenticate to Vault
8. The application uses the token provided in the Vault response to make subsequent requests to Vault, such as retrieve a secret or encrypt data

## *Configuring AppRole*

### Enable the Auth Method

Before AppRole can be used, the auth method must be enabled in Vault. AppRole can be enabled in the UI, CLI, or API. To enable using the CLI with the default path, use the command exhibited in Figure 4-1.

```
$ vault auth enable approle
Success! Enabled approle auth method at: approle/
```

Figure 4-1

To customize the path or description for AppRole, use the command in Figure 4.2.

```
$ vault auth enable -path=applications \
   -description="AppRole auth method for applications" \
   approle
Success! Enabled approle auth method at: applications/
```

Figure 4-2

**Create a Vault Policy to Define Access**

Once AppRole is enabled, a policy should be created (if one does not exist) that defines the application's permissions. For example, suppose the application needs to read a secret stored in a K/V version 2 mount and generate a dynamic credential to access a database. In that case, the policy should permit only those actions. The command displayed in Figure 4-3 can be used to create a policy named *web-app1-policy*.

```
$ vault policy write web-app1-policy - << EOF
path "secret/data/apps/web-app1" {
     capabilities = ["read"]
}
path "database/creds/mysql01" {
     capabilities = ["read"]
}
EOF
Success! Uploaded policy: web-app1-policy
```

Figure 4-3

**Define a Role for the Application**

With the prerequisites now met, the role can be created and associated with the policy. Along with associating the policy to the role, additional configuration items can be defined for the role. A role named *web-app1* can be created using the command shown in Figure 4.4.

```
$ vault write auth/approle/role/web-app1 \
    policies=web-app1-policy \
```

```
    token_ttl=30m \
    token_max_ttl=60m \
    secret_id_ttl=120m
Success! Data written to: auth/approle/role/web-app1
```

Figure 4-4

After one or many roles have been written to Vault, the roles configured in Vault can be listed with a simple command. The command in Figure 4-5 results in a list of the current roles and does not display details of the roles.

```
$ vault list auth/approle/role
web-app1
web-app2
jenkins
```

Figure 4-5

To view the details of a specific role in Vault, use the command in Figure 4-6 to read the configuration and display the result.

```
$ vault read auth/approle/role/web-app1
Key                        Value
---                        -----
bind_secret_id             true
local_secret_ids           false
secret_id_bound_cidrs      <nil>
secret_id_num_uses         0
secret_id_ttl              0s
token_bound_cidrs          []
token_explicit_max_ttl     0s
token_max_ttl              1h
token_no_default_policy    false
token_num_uses             0
token_period               0s
token_policies             [web-app1-policy]
```

```
token_ttl                    30m
token_type                   default
```

Figure 4-6

**Generate the RoleID**

Once the role has been written and the configuration has been validated, the RoleID can be generated. As discussed, the RoleID remains static regardless of what entity is authenticating with it. Figure 4-7 displays the command and output that is used to read the RoleID. In this example, the RoleID has a value of *0a953a6b-f2f7-f34f-4f96-cb2e4e26fdfc*, which will be used by applications as part of the authentication process.

```
$ vault read auth/approle/role/web-app1/role-id
Key        Value
---        -----
role_id    0a953a6b-f2f7-f34f-4f96-cb2e4e26fdfc
```

Figure 4-7

**Generate a SecretID**

In addition to the RoleID, a SecretID needs to be generated for the role. Like other secrets in Vault, the Secret ID has an accessor. The accessor references a separate token that can be used to perform limited functions on the service token but provides secret obfuscation. Each request for SecretID results in a unique SecretID and is bound by the role's configuration parameters. For example, the SecretID may only be valid for thirty minutes if the *secret_id_ttl* parameter has been configured.

The command in Figure 4-8 can be used to generate a SecretID for a role.

```
$ vault write -f auth/approle/role/web-app1/secret-id
Key                   Value
---                   -----
secret_id             448696f6-9397-da64-57ae-cf220a9c5cb9
secret_id_accessor    787ad769-6a5c-1b7e-506b-fc4ce50252eb
```

Figure 4-8

In this example, the value of the SecretID is *448696f6-9397-da64-57ae-cf220a9c5cb9*, which will be used by the entity to authenticate to Vault. As discussed, the SecretID also has an accessor that can be used to manage the SecretID.

In the event a Vault operator does not want the SecretID to be displayed to the terminal, the command displayed in Figure 4-9 can be used to generate the SecretID but write it directly to a file instead. Subsequent commands or automated processes can then reference the file.

```
$ vault write -f -format=json \
   auth/approle/role/web-app1/secret-id \
   | jq -r '.data.secret_id' > web-app1-secretID.txt
```

Figure 4-9

In general, many SecretIDs are generated for a single role in an organization. Vault operators may occasionally need to understand how many SecretIDs have been generated and view information about them to help troubleshoot issues with application authentication. Vault provides a way to view the current SecretIDs associated with a particular role. Vault, however, will only display the accessors to prevent unauthorized access to SecretIDs. The command to list all the SecretID accessors for a particular role is depicted in Figure 4-10. In this example, there are five SecretIDs associated with the role of *web-app1*.

```
$ vault list auth/approle/role/web-app1/secret-id
Keys
----
64fe59a6-2c85-b61d-8cdc-dd88a6616a1b
787ad769-6a5c-1b7e-506b-fc4ce50252eb
a2760f0d-02e8-f2ec-e6d3-da9b95af0ea6
b5d66d01-129d-8d16-d8fd-f97fe032d381
ce333848-4724-7991-fe1c-de6e77a0ee3a
```

Figure 4-10

Once the RoleID is captured and a SecretID has been generated, the application can proceed to authenticate to Vault using the combination of the two items.

> **Note:** Although the configuration for AppRole is shown using the command line, AppRole login typically occurs through the API when the request originates from an application. However, authenticating with AppRole can be accomplished using the CLI as well. The login example below is shown in both the CLI and HTTP API.

Figure 4-11 displays how the authentication request is sent to Vault using the API, where the payload shown includes the *role_id* and the *secret_id* parameters.

```
$ cat payload.json
{
  "role_id": "95eb7a50-a264-3256-f98d-2fd8ec5abad1",
  "secret_id": "2406be8f-2a01-bb03-b6e9-beb2e4821735"
}

$ curl \
    --request POST \
    --data @payload.json \
    http://127.0.0.1:8200/v1/auth/approle/login
```

Figure 4-11

The response from Vault includes the client access token needed to make requests to Vault. Figure 4-12 displays an example response, where the client access token is identified as *9ecbd814-1c83-6cbd-00cc-f0266fafc1d2*.

```
{
  "auth": {
    "renewable": false,
    "lease_duration": 3600,
    "metadata": null,
    "token_policies": ["web-app1-policy"],
    "accessor": "b6569103-2ce5-8d7a-7103-ecadb4674000",
    "client_token": "9ecbd814-1c83-6cbd-00cc-f0266fafc1d2"
  },
```

```
  "warnings": null,
  "wrap_info": null,
  "data": null,
  "lease_duration": 0,
  "renewable": false,
  "lease_id": ""
}
```

Figure 4-12

Since the Vault output is formatted in JSON, a simple JSON processor, such as *jq,* can be used to extract the client access token from the Vault response programmatically. An example of this is shown in Figure 4-13.

```
<command> | jq -r '.auth.client_token' > token.txt
```

Figure 4-13

Figure 4-14 displays how to authenticate using AppRole via the CLI.

```
$ vault write auth/approle/login \
   role_id=0a953a6b-f2f7-1127-4f96-cb2e4e26fdfc \
   secret_id=448696f6-9397-da64-57ae-cf220a9c5cb9
Key                     Value
---                     -----
token                   s.UEiJmfQe20R7JcB7uHRjdMWp
token_accessor          tlpS5Tj5Ff8KNk5c20p0Tm6c
token_duration          3600
token_renewable         true
token_policies          ["default" "web-app1-policy"]
identity_policies       []
policies                ["default" "web-app1-policy"]
token_meta_role_name    web-app1
```

Figure 4-14

# Cloud Provider Auth Method

As organizations continue to migrate workloads from private data centers to the public cloud, cloud provider auth methods are quickly becoming a staple in almost every Vault deployment. Vault is a cloud-agnostic product that supports one or more auth methods right out of the box for the most popular cloud providers. Integrations with these cloud providers accommodate various methods for applications to authenticate with Vault without relying on long-lived tokens or AppRole configuration. This strategy increases security while permitting applications to access Vault services.

Cloud provider auth methods permit specific cloud-based entities to access Vault. These cloud-based identity management systems are treated as third-party trusted services by Vault. For example, Vault can be configured to permit an IAM-based entity in AWS, such as an AWS Lambda function's execution role. Alternatively, Vault can also use signed metadata information supplied to each EC2 instance for authentication. As cloud-based workloads are provisioned, they can easily access Vault without additional operating system-based or app-based configurations.

Once the auth method has been enabled in Vault, cloud-based entities can be mapped to a policy using a named role. A role binds the entity to one or more Vault policies, which permits or denies access to Vault along with additional settings. Roles can be created based on the different application requirements, which are generally defined by the access required within Vault.

The basic workflow to enable and configure a cloud provider auth method is simple.

1. Enable the auth method
2. Configure credentials to permit Vault access to the cloud provider
3. Create or identify a Vault policy for the application
4. Create a role to map the cloud-based entity to the Vault policy

## *Configuring a Cloud Provider Auth Method*

**Enable the Auth Method**

The first step to using a cloud provider auth method is to enable it in Vault. Auth methods can be enabled in the UI, CLI, or API. To enable the AWS auth method on the default path using the CLI, use the command in Figure 4-15.

```
$ vault auth enable aws
Success! Enabled aws auth method at: aws/
```

Figure 4-15

Now that the auth method has been enabled, Vault needs permission to make API calls to the cloud provider to validate the identity of a requesting client. If Vault has been provisioned on a cloud platform, integrated features such as IAM roles can be used to permit Vault nodes the access to make API calls without embedding credentials in the auth method configuration. If Vault was provisioned outside of the cloud platform, credentials need to be provided to Vault.

For Vault environments provisioned on AWS, the IAM role needs to permit the *sts:GetCallerIdentity* for AWS resources that could become Vault clients. While the policy in Figure 4-16 is an example of a policy permitting this action, keep in mind that other actions may already be configured for Vault nodes in an existing policy. This permission can be added to that policy.

```
{
  "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": "iam:GetInstanceProfile",
        "Resource": "*"
      }
    ]
}
```

Figure 4-16

**Configure the Auth Method with Credentials**

For Vault environments that are provisioned outside of AWS, the configuration in Figure 4-17 can be used to provide credentials to the AWS auth method.

```
$ vault write auth/aws/config/client \
secret_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY \
access_key=AKIAIOSFODNN7EXAMPLE
Success! Data written to: auth/aws/config/client
```

Figure 4-17

While embedding an AWS access key and secret key may be a security concern, it should be noted that the credentials cannot be read back once submitted to Vault. As an example, after the configuration is executed, the result of reading the configuration can be found in Figure 4-18. Note that the secret key is not displayed on the response.

```
$ vault read auth/aws/config/client
Key                          Value
---                          -----
access_key                   AKIAIOSFODNN7EXAMPLE
endpoint                     n/a
iam_endpoint                 n/a
iam_server_id_header_value   vault.example.com
max_retries                  -1
sts_endpoint                 n/a
sts_region                   n/a
```

Figure 4-18

HashiCorp also recommends setting the header value for the AWS auth method to avoid reply attacks against Vault. This header is recommended, especially for organizations that have multiple Vault clusters for different software development lifecycles. The inclusion of this header prevents a request for a dev server from being re-sent to a production server. Most often, this header is set to the name of the Vault cluster. This parameter can be passed alongside the parameters being set in the command depicted in Figure 4-17, or it can be issued separately, as shown in Figure 4-19.

```
$ vault write auth/aws/config/client \
    iam_server_id_header_value=vault.example.com
Success! Data written to: auth/aws/config/client
```

Figure 4-19

**Create a Vault Policy to Define Access**

Now that the auth method is configured, a Vault policy can be created (if one does not exist) that defines the permissions for the application. For example, if the application needs to read a secret and generate a dynamic credential to access a database, it should permit only those actions. The command found in Figure 4-20 can be used to create a policy named *dev-lambda-read-only*.

```
$ vault policy write dev-lambda-read-only -<<EOF
path "database/creds/mysql01" {
    capabilities = ["read"]
}
EOF
Success! Uploaded policy: dev-lambda-read-only
```

Figure 4-20

**Create a Role for the Application**

Once a policy is created, a role can be created to map the cloud-based entity to the Vault policy. The role can include multiple policies if needed, including specific settings that Vault uses for authentication.

The command in Figure 4-21 displays the command for creating a new role called *dev-aws-lambda.* The role assigns the Vault policy *dev-lambda-read-only,* created in Figure 4-20, to permit access to generate dynamic credentials against a database. Using the AWS secrets engine iam_role method, this configuration permits only the execution role of *arn:aws:iam::123456789012:role/dev-lambda-function* to access Vault. In this example, the TTL is also set to a value that matches the AWS Lambda maximum timeout of 900 seconds.

```
$ vault write auth/aws/role/dev-aws-lambda \
    auth_type=iam \
```

```
    token_ttl=900s \
    max_ttl=900s \
    policies=dev-lambda-read-only \
    resolve_aws_unique_ids=false \
    bound_iam_principal_arn=arn:aws:iam::123456789012:
      role/dev-lambda-function
Success! Data written to: auth/aws/role/dev-aws-lambda
```

Figure 4-21

Now that the AWS auth method is configured, it can be used to authenticate to Vault. In the snippet of Python code seen in Figure 4-22, a Lambda function can connect to Vault using the AWS auth method and request credentials using Boto3 (*https://aws.amazon.com/sdk-for-python*) and the HVAC client for Vault (*https://pypi.org/project/hvac/).*

```
def getDBConnection():

    credentials = session.get_credentials()
    vault = hvac.Client(url=os.environ["vault_url"])

    vault.auth.aws.iam_login(
        credentials.access_key,
        credentials.secret_key,
        credentials.token,
        header_value=os.environ["xVaultAWSIAMServerID"],
        role="dev-aws-lambda")

    vault_res = vault.read("database/creds/dev-aws-lambda")

    dbUser = vault_res["data"]["username"]
    dbPass = vault_res["data"]["password"]
```

Figure 4-22

As another example, the AWS Lambda extensions can also be used to retrieve secrets from Vault as well. (*https://github.com/hashicorp/vault-lambda-extension)*

## OIDC Auth Method

Many organizations are employing a single sign-on solution for authentication across their IT environment. Using Vault's OIDC auth method, organizations can leverage their existing user directory for fine-grained access to Vault services. Configuring Vault to interact with an OIDC provider gives organizations additional benefits over direct integration with AD, such as direct multi-factor integration.

Vault's OIDC auth method works by using the user's web browser. When a user accesses Vault and requests authentication via OIDC, Vault launches a new browser window using the configured discovery URL that prompts the user for credentials. Once the user submits credentials, Vault validates the credentials against the OIDC provider. Assuming the credentials are valid, the OIDC provider responds to Vault in the form of a token. This token contains user information, including OAuth scopes and a signature that is passed back to Vault. The OAuth scope received from the OIDC provider contains information about the user, such as the security group(s) that the user is a member of, which could map to an external group within Vault using a group claim. The preconfigured external group determines which Vault policy or policies the user's token is then attached to.

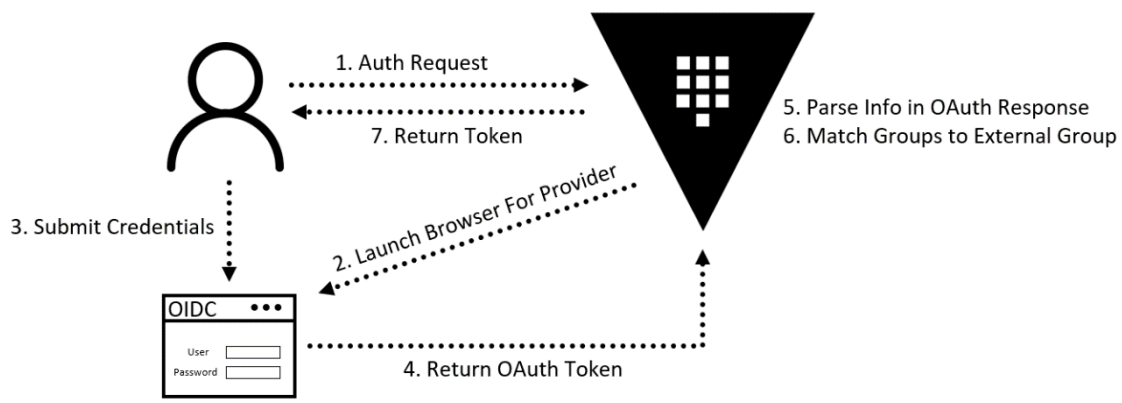



Figure 4-23

The basic workflow for the OIDC auth method mapping to an external group, for example, has multiple integrated steps to complete before a user is granted access to Vault. This workflow is depicted in Figure 4-23.

1. User requests authentication via OIDC auth method
2. Vault launches a browser to the configured discovery URL for the OIDC provider
3. The user submits credentials to the OIDC provider
4. The OIDC provider validates credentials and returns a token to Vault
5. Vault parses the information in the token, specifically the memberOf groups provided by the OAuth scope
6. Vault compares the groups to the preconfigured external Vault groups
7. If there is a match to one or more groups, a token is created and associated with the Vault policy configured for the matching external group(s)
8. If there is no match, a token is created and associated with the Vault policy configured for the default role.

## *Configuring the OIDC Auth Method*

**Enabling the Auth Method**
As with most other auth methods, the OIDC auth methods needs to be enabled. Most auth methods can be enabled and configured in the UI, CLI, or API.

To enable the OIDC auth method using the CLI on the default path, use the command displayed in Figure 4-24.

```
$ vault auth enable oidc
Success! Enabled oidc auth method at: oidc/
```

Figure 4-24

**Configure the Auth Method**
Once the OIDC auth method has been enabled, the auth method itself must be configured. These settings include the *oidc_discovery_url*, *oidc_client_id*, and the *oidc_client_secret* shared by Vault and the OIDC provider for authentication. These settings are critical to configuring the communication between Vault and the OIDC provider. The *default_role* needs to be defined as well.

To configure the OIDC settings, use the command found in Figure 4-25.

```
$ vault write auth/oidc/config
    oidc_discovery_url="https://sso.example.com:443" \
    oidc_client_id="client-id" \
    oidc_client_secret="super-secret-password" \
    oidc_discovery_ca_pem=@ca.pem \
    default_role="default"
```

Figure 4-25

**Create the Default Role**

Now that the OIDC auth method is configured, the default role must be created. The default role and the associated policy are used only when a user successfully authenticates using OIDC but is not a member of a configured group. Commonly, the default role is assigned the Vault *default* policy, limiting the user's capabilities upon authentication. It is also essential to configure the *allow_redirect_uris* to define where users are redirected after authenticating. Note that these redirect URIs must match exactly with the configuration on the IdP configuration. If OIDC is being configured within a namespace, the URL should include the namespace name as well.

```
$ vault write auth/oidc/role/default \
    bound_audiences="client-id" \
    allowed_redirect_uris="https://vault.example.com:8200/ui/vault
      /auth/oidc/oidc/callback" \

    allowed_redirect_uris="http://vault.example.com:8250/oidc/call
    back" \
    user_claim="sub" \
    policies="default" \
    oidc_scopes="groups" \
    groups_claim="groups" \
    verbose_oidc_logging="false"
```

Figure 4-26

**Create an External Group and Policy Mapping**

The last configuration for OIDC is to create the external groups. Each external group is associated with a Vault policy based on the access that should be granted. Once the external group is created, a group alias is created. The group alias essentially maps the

external group to a group in the OIDC auth provider. When Vault matches an external group with an OIDC auth provider group, it adds the entity to the external group as a member. It then creates a token with the associated policy and returns it to the user.

The command in Figure 4-27 can be used to create an external group in Vault for Vault administrators, for example, and save the external group ID to a file for future reference.

```
$ vault write -format=json identity/group name="admins" \
    type="external" \
    policies="vault_admin" | jq -r ".data.id" > group.txt
```

Figure 4-27

**Create the Group Alias**

Once the group has been created, the group alias needs to be defined, mapping the response from the OIDC provider to the external group. The command detailed in Figure 4-28 shows how to add a group alias to the external group found above.

```
$ vault write identity/group-alias name="vault-admins" \
    mount_accessor=<accessor_id> \
    canonical_id=$(cat group.txt)
```

Figure 4-28

Other external groups and group aliases may be created depending on the needs of the organization. Most organizations have many Vault users with similar policy requirements that can be easily grouped using this method. Developers, for example, can be grouped into a single group if they have similar requirements for access in Vault.

## LDAP Auth Method

Most organizations centralize credentials for end-users, applications, and services using an LDAP solution, like Microsoft Active Directory. Regardless of the solution being used, it is common for Vault operators to integrate Vault with the existing LDAP solution so as not to increase the number of accounts a user needs for authentication. The Vault LDAP auth method enables this functionality, allowing Vault operators, end-users, and even applications to authenticate with Vault with existing LDAP credentials. The LDAP auth

method is both secure and flexible, allowing it to meet the needs of a wide variety of organizations.

The LDAP auth method works by communicating directly with an organization's LDAP servers. When authentication is requested via LDAP, the Vault server validates the credentials with the backend LDAP server using a service account. Once the credentials have been validated, Vault generates a token, attaches the corresponding Vault policy, and returns the token to the user.

The LDAP auth method is configured using the primary configuration to establish communication with the LDAP servers and the group to policy mappings. This mapping assigns one or more policies to authenticated members of an LDAP group. If needed, policies can be mapped directly to a specific user as well.

Before the LDAP auth method can be configured, the Vault operator must gather various information about the environment with which Vault is integrated. The parameters required for each environment depend solely on the deployment and architecture of the LDAP solution. Therefore, configurations will vary by organization.

Some of the LDAP information to be collected includes:

- Service Account and password used to make LDAP queries
- Certificates used to secure communications between Vault and LDAP
- LDAP server names or IP addresses
- The distinguished name (DN) of the OU where user search starts
- The distinguished name (DN) of the OU where the group search starts
- User attribute to use – frequently *cn* or *sAMAccountName*
- Group attribute to use – frequently set to *cn*

Vault also must be able to communicate to the LDAP server(s) over TCP and UDP port 389 or TCP. Alternatively, if LDAPS is in use, Vault needs to communicate using TCP and UDP 636.

## *Configuring the LDAP Auth Method*

### Enabling the Auth Method

Before the LDAP auth method can be used, it must be enabled. Like other auth methods, the LDAP auth method can be enabled using the UI, CLI, or API. To enable the LDAP auth method using the CLI with the default path, use the command displayed in Figure 4-29.

```
$ vault auth enable ldap
Success! Enabled ldap auth method at: ldap/
```

Figure 4-29

### Configure the Auth Method

Once the LDAP auth method is enabled, it must be configured to permit Vault to communicate to the backend LDAP server(s). The command found in Figure 4.30 shows how to configure the LDAP auth method using basic LDAP configurations.

```
$ vault write auth/ldap/config \
    url="ldap://dc1.example.com, ldap://dc2.example.com" \
    userdn="ou=Users,ou=Resources,dc=example,dc=com" \
    groupdn="ou=Groups,ou=Resources,dc=example,dc=com" \
    groupattr="cn" \
    userattr=”sAMAccountName” \
    binddn="cn=svcvault,ou=ServiceAcct,dc=example,dc=com" \
    bindpass=”supersecretpassword” \
    certificate=@ca_cert.pem \
    insecure_tls=false \
    starttls=true
    token_ttl=2h
    token_max_ttl=8h
Success! Enabled ldap auth method at: ldap/
```

Figure 4-30

### Assign a Vault Policy to a Group

Now that Vault can successfully communicate with LDAP using the configuration in Figure 4-x, existing security groups or users in LDAP need to be mapped to the appropriate Vault

policies. Creating this mapping allows organizations to manage group membership and Vault privileges within LDAP rather than creating and managing groups within Vault itself.

The command in Figure 4-31, depicts an example where the *vault-admins* group in LDAP is mapped to the *policy-vault-admin* policy within Vault, which would provide sufficient privileges for Vault administrators.

```
$ vault write auth/ldap/groups/vault-admins \
Policies=policy-vault-admin
Success! Enabled ldap auth method at: ldap/
```

Figure 4-31

Once a user who is a member of the *vault-admins* group authenticates to Vault, the user is assigned the *policy-vault-admin policy* and inherits the permissions defined within that policy.

## *Secrets Engines*

Secrets engines are the heart of Vault and provide Vault's core functionality. All other aspects of Vault are built around the use of secrets engines. Without secrets engines, Vault provides little to no value.

Within any organization, static credentials have long been the bane of security. They pose a threat because these credentials are rarely rotated. In addition, they are frequently shared between team members, eliminating accountability, as any member can authenticate with the shared credential. Vault secrets engines are essential in solving this issue because they improve an organization's security posture through its unique approach to managing secrets.

While Vault gives organizations a way to consolidate these long-lived credentials, if needed, Vault's real power is demonstrated with its ability to generate dynamic credentials on-demand for a variety of secrets. Dynamic credentials provide a way for applications to generate unique sets of temporary credentials rather than using a shared service account to access other services. These temporary credentials are then revoked as soon as the associate lease expires, whether due to time or number of uses. This strategy helps organizations minimize exposure of static, privileged credentials.

As an example of the benefits Vault may provide, Figure 4-32 shows an application's workflow that generates dynamic credentials for a database when data needs to be retrieved. In this case, no application-level credentials exist for the database until they are requested. As the credentials are requested, the application authenticates to Vault, generates the secret, and accesses the required data. Once the application no longer needs access to the database, the access can be manually revoked, or it automatically expires once the TTL is reached.

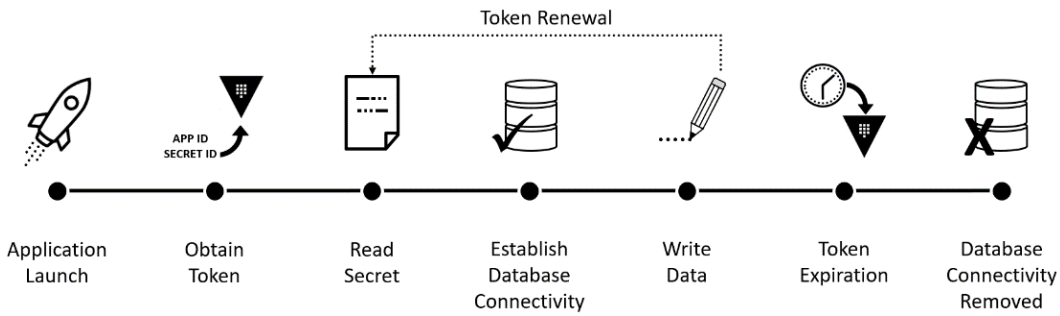



Figure 4-32

Commonly, organizations seek to incorporate secrets engines as part of their automation stack. This strategy allows a Terraform deployment or a CI/CD pipeline, for example, to read or generate secrets during infrastructure or application provisioning. Like the database example, a cloud platform's credentials may be generated when a job is executed. The pipeline can then proceed to provision application workloads and supporting components using the newly created credentials. Once the workload deployments are complete, the pipeline can revoke the credentials as they are no longer needed. Figure 4-33 shows an example of this automation pipeline.

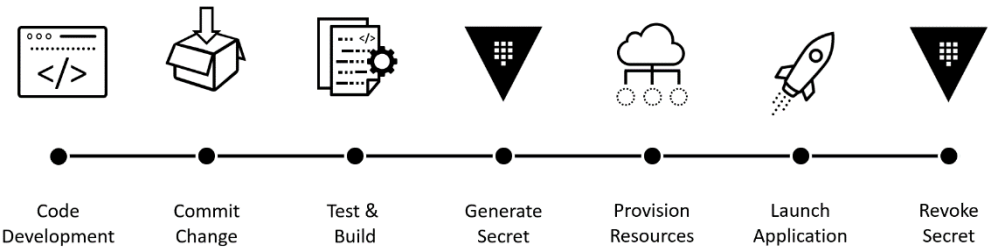



Figure 4-33

Like other configurations in Vault, secrets engines are enabled and mounted at a path. All interactions with the secrets engine are performed through the path. For clients to interact with a secrets engine, the request must be accompanied by a valid token that provides the appropriate permissions. These permissions include access to the requested path and grants for the action being requested. This strategy allows Vault operators to permit actions on the specific paths needed by the application or requesting user, following the principle of least privilege.

## Common Secrets Engines

Secrets engines are the Vault component that can store, generate, or encrypt data for clients. Since secrets engines provide the core functionality of Vault, there is an extensive selection that can be used right out of the box. Each secrets engine's functionality is targeted towards a particular integration or supporting platform. The same secrets engine may be used differently in each situation, depending on the requirements of the Vault use case.

Organizations often utilize a variety of secrets engines within their Vault deployment. Infrastructure and developer teams can consolidate their identity needs in Vault, continually bringing new use cases and requests to the Vault environment. Based on experience, the most popular secrets engines include:

- Key/Value
- Database
- Cloud Provider (AWS, Azure, GCP)
- Transit
- PKI

## Key/Value Secrets Engine

The K/V secrets engine allows organizations to consolidate long-lived, static secrets into a key/value structure that is easy to operate and consume. The K/V secrets engine is the only secrets engine that enables users to store secrets that are written to disk and persists beyond the life of a single user's token. This capability is ideal for organizations that need to store artifacts for many different teams, applications, or use cases and allow this data to be accessed manually or programmatically. The K/V secrets engine is also a common entry point for companies who have recently adopted Vault and are migrating secrets from another solution.
Vault makes two versions of the Key/Value secrets engine available to its users: the simpler K/V version 1 and the more feature-rich K/V version 2. Most organizations opt to enable K/V version 2. However, there are use cases for K/V version 1. K/V version 1 requires less storage and fewer reads/writes against the underlying storage since it is inherently writing less data due to the lack of associated metadata. However, in most customer environments,

the underlying storage can handle additional requests. The additional features provided by K/V version 2 generally offset the additional read/write and storage overhead.

### *K/V Version 1*

K/V version 1 provides Vault users with a Key/Value store in its simplest form. For each entry, Vault only stores the most recently written value for each key. There is no metadata or history associated with the K/V version 1 store entries. If a value is overwritten, K/V version 1 does not support the recovery of the overwritten data. It should be noted that a K/V version 1 secrets engine may be upgraded to version 2 if needed. However, migrating from version 2 to version 1 is not possible.

### *K/V Version 2*

K/V version 2 supports both versioning and recovery of data using the metadata associated with each key version. When data is overwritten, previous versions of the key can be referenced if need. Additionally, if a key is deleted, Vault uses a delete marker rather than destroying the data on disk. This delete marker enables users to recover the data if needed. K/V version 2 also enables additional functionality on each path. These additional features include the ability to define the number of versions to keep, use check-and-set operations to avoid overwriting data, and the inclusion of timestamps for data creation and deletion.

### *Organizing Data*

When writing data to the Key/Value secrets engine, it is common for companies to design the structure of how data is organized under the secrets engine mount path before writing data to Vault. Investing time to design the K/V structure helps simplify Vault policies for Vault operators, as similar data should be grouped beneath the same path, where possible. A well-architected K/V structure also helps reduce the number of Vault policies to secure access to the data stored in the K/V secrets engine.

As an example of a well-architected K/V structure, Figure 4-34 shows a segment of a K/V store that was designed with a specific infrastructure team in mind. Remember that every K/V structure can differ depending on the needs of different teams and applications reading secrets from or writing secrets to Vault.

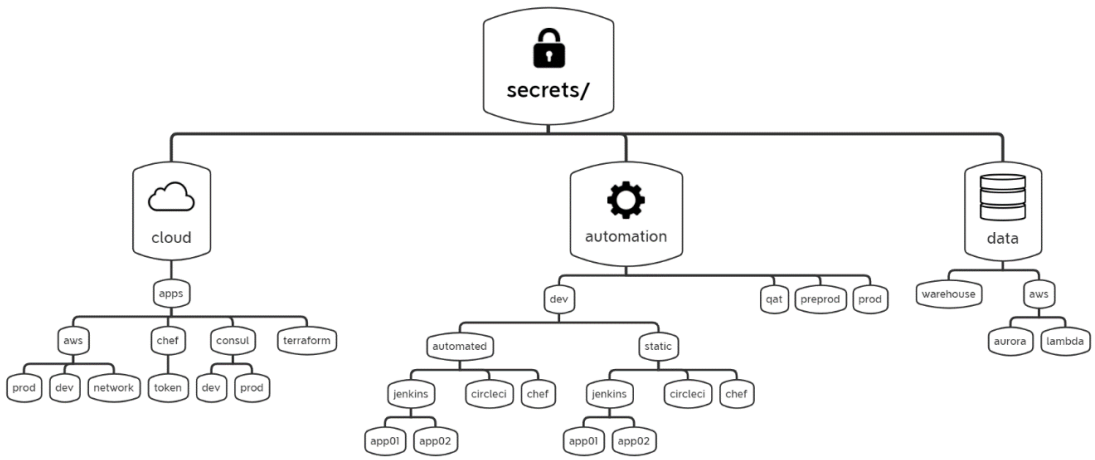



Figure 4-34

**Customer Story:** A major tech company adopted Vault company-wide due to strict performance, availability, and scalability requirements. Two initial objectives needed to be met: moving secrets out of code and migrating secrets from the current secrets management tools. Because Vault was a new tool and there would be an initial learning curve, this company opted to first move all current secrets to the K/V (version 1) secrets engine. The support team planned out the organization of the K/V tree to support a path per business unit and a sub-path per team. Policies were created to align with this path structure, and teams were onboarded. Integrations were built-in applications, configurations management tools, and deployment workflows to not only retrieve secrets but to rotate the value of those secrets every 90 days. Eventually, teams were migrated to K/V version 2, allowing them to use versioning for improved reliability during the secrets rotation process. This process paved the way for teams to adopt dynamic secrets according to their needs and requirements.

## *Configuring the K/V Secrets Engine*

### Enable the K/V Secrets Engine

Before using the K/V store, the secrets engine must be enabled. The K/V secrets engine can be enabled in the UI, the CLI, or the API. To enable the K/V secrets engine using the

CLI on a custom path, use the command found in Figure 4-35. Unless K/V version 2 is specified, the basic command enables K/V version 1.

```
$ vault secrets enable -path=kv_cloud kv
Success! Enabled kv secrets engine at: kv_cloud/
```

Figure 4-35

If the Key/Value version 2 secrets engine should be enabled, the version number can be specified when executing the command, as shown in Figure 4-36.

```
$ vault secrets enable -version=2 -path=kv_infra kv
Success! Enabled kv secrets engine at: kv_infra/
```

Figure 4-36

The Key/Value version 2 secrets engine can also be enabled by referencing the name of the versioned Key/Value secrets engine, as displayed in Figure 4-37.

```
$ vault secrets enable -path=secret kv-v2
Success! Enabled kv secrets engine at: secret/
```

Figure 4-37

Since there are two different versions of the Key/Value secrets engine, it may be necessary for a Vault operator to query Vault to understand what version each secrets engine is running. To see which version each Key/Value secrets engine is running, use the command in Figure 4-38. This figure shows the command and a subset of the response. Notice that for Key/Value version 2 secrets engines, the version is indicated under the *Options* column. If no version is indicated, the secrets engine is running version 1.

```
$ vault secrets list -detailed
Path             Plugin             Options
----             ------             -------
cubbyhole/       cubbyhole          map[]
identity/        identity           map[]
```

```
kv_cloud/         kv_vd2e8d60         map[]
kv_infra/         kv_b7a81a5d         map[version:2]
secret/           kv_ec0090c0         map[version:2]
sys/              system_5f73a609     map[]
```

Figure 4-38

**Upgrade K/V to Support Versioning**

To upgrade a version 1 Key/Value secrets engine to a version 2 secrets engine using the CLI, use the command in Figure 4-39. Note that during the upgrade process, the mount is not accessible to users, so Vault operators should plan accordingly.

```
$ vault kv enable-versioning kv_cloud/
Success! Tuned the secrets engine at: kv_cloud/
```

Figure 4-39

**Note:** Any time a substantial change is made to Vault, it is essential to ensure there are current backups of the data. Replication is not enough to protect from data loss, as changes to the data are immediately reflected across performance replica and disaster recovery clusters. Having a current snapshot of the data helps ensure that it can be restored to a current state if something goes astray.

## *Working with the K/V Secrets Engine*

Vault supports many operations that a Vault operator or end-user may use to interact with the Key/Value secrets engine. These operations are used to add, delete, or read data within the Key/Value secrets engine. While both Key/Value versions support the general operators, only metadata-related operations are supported by version 2.

- Supported by Version 1 and Version 2
    - Write (put)
    - Read (get)
    - Delete

    - List

- Supported by Version 2 only
    - Undelete
    - Destroy
    - Patch
    - Rollback

## *Working with K/V Version 1*

Working with the K/V version 1 secrets engine is relatively straightforward. General operations such as write, read, delete, and list follows the operation in its conventional verbal meaning.

**Write Data**
Write operations to a path automatically create the path structure up to the written key, so there is no requirement to stage the structural design before writing a key and its corresponding value. If a write operation is requested for a key that already exists, it is overwritten without warning. The old value will no longer be available. Keep in mind that a write operation is not a merge. If additional data needs to be added to a path, the new write must include both the old data and the new data.

Writing data to a new path requires *create* permissions. However, writing data to an existing path requires *update* permissions. Figure 4-40 shows how to write a new key pair to a specific path using the *put* command. Additional key pairs can be appended to the command if needed.

```
$ vault kv put secret/vault/book running=production
Success! Data written to: secret/vault/book
```

Figure 4-40

**Read Data**
A read request command can be issued against the path where data is stored to retrieve previously written data. The response includes all corresponding key pairs stored on the path. When a path includes multiple key pairs, a request can be made to read a specific key

using the *field* parameter. If there is no data stored at the requested path, Vault indicates no value is stored at the requested path. Reading data from the K/V store requires *read* permissions on the appropriate path. Figure 4-41 shows how to retrieve data from the K/V store using the *get* command.

```
$ vault kv get secret/vault/book
=== Data ===
Key       Value
---       -----
running   production
```

Figure 4-41

**Delete Data**

Once data is written to the K/V store, it can easily be deleted. Since the K/V version 1 store does not support versioning or metadata, a delete operation permanently destroys the data at the requested path. The only way to retrieve deleted data is to restore the entire storage backend. Deleting data from a K/V store requires *delete* permissions on the appropriate path. Figure 4-42 shows how to delete data from the K/V store using the delete command.

```
$ vault kv delete secret/vault/book
Success! Data deleted (if it existed) at: secret/vault/book
```

Figure 4-42

**List Data**

Occasionally, an end-user may need to list keys in the K/V store to find a path for a key pair. This type of request is almost always generated by a human user as automated systems and applications target a known path. Although a user may request a list operation using the UI, CLI, or API, an end-user will almost always use the CLI or UI, as those methods are more human friendly. To browse using the CLI, the user needs to have *list* permissions on the desired path. If the user expects to browse the K/V store using the UI, the user needs *list* permissions starting at the root of the K/V mount. Figure 4-43 shows how to list data at a specific path using the list command.

```
$ vault kv list secret/vault/book
```

```
Keys
----
running
```

Figure 4-43

## *Working with K/V Version 2*

K/V version 2 introduces metadata for stored secrets, which changes how users may interact with the data stored within the K/V store. Since each key pair in Vault is versioned and has metadata associated with it, the key pairs are prefixed with a *data* path. The metadata is also prefixed with a specific path of *metadata*. Additional prefixes, such as *delete*, *undelete*, and *destroy*, are used to manage versions as well.

Compared to K/V version 1, the addition of the prefixes often confuses Vault operators regarding how Vault handles the paths for both the key pairs and metadata. When interacting with the K/V version 2 store using the API, the prefixes are required for every action requested. In contrast, the CLI operates differently and does not require the prefixes. As an example of the differences, a CLI request does not require the *data/* prefix, while an API does.

Vault policies for permitting access to a K/V version 2 store differ from policies written for a K/V version 1 store. Vault operators must construct policies to include the prefix required for the intended operation(s). Permissions for K/V version 2 need to include access to prefixes such as *data*, *metadata*, *delete*, *undelete*, and *destroy*. For example, if a user needs to read a credential stored in a secrets engine mounted at *cloud_team*, the user requires ***read*** permissions to *cloud_team/data/<path to secret>*. If the user needs to list the secrets beneath a path, the user requires ***list*** permissions to *cloud_team/metadata/<path>/**. More examples can be found in the official K/V version 2 documentation (*https://www.vaultproject.io/docs/secrets/kv/kv-v2*).

**Write Data**

Writing to a K/V version 2 secrets engine using the CLI is identical to writing data to a K/V version 1 secrets engine. The same command formatting applies when making the request; however, Vault's response is slightly different. As K/V version 2 is a versioned store, the response includes the *creation_time*, *deletion_time*, whether the version has

been *destroyed*, and the current version number of the Key/Value pair. Figure 4-44 displays the command and resulting output from Vault.

```
$ vault kv put secret/vault/book running=production
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0566577Z
deletion_time    n/a
destroyed        false
version          1
```

Figure 4-44

When writing data using the API, the *data/* prefix is always required for each request. Figure 4-45 displays an example of the payload and the command to write data to the K/V version 2 store using the API.

```
$ cat payload.json
{
 "data": {
   "running":"production"
 }
}
```

```
$ curl \
    --header "X-Vault-Token:<token>" \
    --request POST \
    --data @payload.json \
    https://127.0.0.1:8200/v1/secret/data/vault/book
```

Figure 4-45

**Read Data**

Read operations for the K/V version 2 secrets engine follow a similar access pattern to a write operation. The *data/* prefix is required when using the API but is not required when using the CLI. The addition of versions, however, offers further options for the request. By

merely reading the secret path, Vault always returns the most recent version of the requested secret. If a previous version of the secret is desired, the version may be indicated within the request. The command in Figure 4-46 displays how to read the most current version of a secret. Although the command is identical to K/V version 1, the Vault response includes both the requested data and the associated metadata. Since the *write* operation displayed in Figure 4-44 was the initial write for this key, Vault has indicated this is the first version of this key.

```
$ vault kv get secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0566577Z
deletion_time    n/a
destroyed        false
version          1

===== Data =====
Key        Value
---        -----
running    production
```

Figure 4-46

When reading data from a K/V version 2 store using the API, the *data/* prefix is always required. Figure 4-47 provides an example of the command used to request the latest version of a key and the response from Vault.

```
$ curl \
    --header "X-Vault-Token:<token>" \
    https://127.0.0.1:8200/v1/secret/data/vault/book

{
  "request_id": "aa06106f-82ca-a276-65eb-5d60138a5c8a",
  "lease_id": "",
  "renewable": false,
```

```
  "lease_duration": 0,
  "data": {
    "data": {
      "running": "production"
    },
    "metadata": {
      "created_time": "2020-04-22T00:39:00.0566577Z",
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  },
  "wrap_info": null,
  "warnings": null,
  "auth": null
}
```

Figure 4-47

If multiple versions of the key exist, a Vault user can explicitly request the version number if needed. This functionality could prove to be handy if a value were mistakenly or prematurely overwritten or in situations where a secret is in the process of being rotated. When requesting a version other than the current version, the *version* parameter is added to the command or API request. An example of this request can be found in Figure 4-48. Notice the version number in the response as well as the new value.

```
$ vault kv get -version=3 secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0673727Z
deletion_time    n/a
destroyed        false
version          3

===== Data =====
Key        Value
```

```
---          -----
running      production_3
```

Figure 4-48

**Delete Data**

Data in the K/V store may need to be deleted at some point in its lifecycle. K/V version 2 offers a way to delete both the current and previous versions of the key if needed. Unlike K/V version 1, a delete operation for a key does not destroy the underlying data but creates a delete marker. Using a delete marker, Vault prevents the data from being returned from reads but enables the restoration of the data if the need arises. When a delete operation is processed against a key version, Vault performs a soft delete, marking the version as deleted and populating the *deletion_time* timestamp. If a read operation is requested against a particular version of the key after being deleted, Vault returns only the metadata. Other versions of the key may be retrieved using the *versions* parameter, or the version may be restored.

As an example, Figure 4-49 shows the result of deleting the current version of a key, followed by a deleted key request. Note the Vault response from the last command does not return the data but includes the timestamp indicating when the key was deleted.

```
$ vault kv get secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0673727Z
deletion_time    n/a
destroyed        false
version          4

===== Data =====
Key        Value
---        -----
running    production_4

$ vault kv delete secret/vault/book
```

```
Success! Data deleted (if it existed) at:
secret/vault/book

$ vault kv get secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0673727Z
deletion_time    2020-04-22T00:39:00.9923375Z
destroyed        false
version          4
```

Figure 4-49

**Destroy Version**

Sometimes K/V data needs to be permanently deleted from Vault without the ability to restore. Vault provides this capability by using the *destroy* path. When a destroy operation is processed, Vault permanently deletes the underlying data and the destroyed version's metadata. Figure 4-50 displays the command to permanently destroy a version and the result of reading the destroyed version. Notice the *destroyed* value is now set to *true*.

```
$ vault kv destroy -versions=2 secret/vault/book
Success! Data written to: secret/destroy/vault/book

$ vault kv get -version=2 secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0673727Z
deletion_time    n/a
destroyed        true
version          2
```

Figure 4-50

**Permanently Delete Key**

As K/V version 2 keys are tracked using metadata, deleting the metadata key causes all metadata and key versions to be permanently deleted from Vault. By deleting the metadata, the data can no longer be retrieved or restored from Vault. The command found in Figure 4-51 shows the deletion of the metadata key, resulting in the data's permanent deletion.

```
$ vault kv metadata delete secret/vault/book
Success! Data deleted (if it existed) at:
secret/metadata/vault/book

$ vault kv get secret/vault/book
No value found at secret/data/vault/book
```

Figure 4-51

**Restore Data**

The *undelete* path can be used to restore the deleted version of a key. Restoring a key version restores the data and removes the *deletion_time* timestamp from the metadata associated with the specified version. Figure 4-52 shows the restoration of a previously deleted key using the *undelete* path.

```
$ vault kv undelete -versions=3 secret/vault/book
Success! Data written to: secret/undelete/vault/book

$ vault kv get -version=3 secret/vault/book
====== Metadata ======
Key              Value
---              -----
created_time     2020-04-22T00:39:00.0673727Z
deletion_time    n/a
destroyed        false
version          3

===== Data =====
Key        Value
---        -----
```

```
running    production_3
```

Figure 4-52

## Database Secrets Engine

The database secrets engine is used to generate dynamic database credentials against a wide variety of supported databases. The most significant benefit of integrating the database secrets engine with business applications is unique and short-lived access to the backend database that store much of their critical data. By adopting this strategy, applications no longer need to hardcode a generic, long-lived service account to access the database. Each application or node can now generate on-demand credentials (username and password) when access is needed, and Vault revokes those credentials at the end of the TTL.
Database support works using Vault's plugin system, which allows Vault to support many popular database platforms. Database configurations identify the plugin to be used to ensure Vault understands how to interact with the database appropriately. Many different database platforms can be configured beneath a single database secrets engine. For example, a single database secrets engine can generate dynamic credentials for MySQL, MongoDB, and Oracle databases.

When integrating applications, Vault needs to understand what database to generate credentials for and what permissions to assign the newly created user. This mapping is done using a role. There can be one or many roles created for each database that Vault is configured to integrate with depending on different applications or users' needs. Each role can be configured to provide vastly different permissions on the same back-end database. This strategy ensures the credentials have the minimum privileges needed on the application's database to function correctly.

Beyond dynamic credential functionality, Vault also supports static roles. Static roles allow a Vault operator to map a single role to a specific database user. Vault maintains the static username on the backend database but can rotate the password as needed. This feature allows organizations to integrate commercial-off-the-shelf (COTS) applications requiring a specific username for the backend database.

Using the database secrets engine starts with Vault client authentication. Once a client is successfully authenticated and is issued a token, it can request credentials from the database secrets engine using the following workflow.

1. A read is issued against database/creds/<role name>
2. Using a pre-configured service account, Vault creates the new user and password on the database with the appropriate permissions.
3. The database responds to the request
4. Vault attaches the configured TTL to the credentials
5. Vault responds to the client with the database credentials
6. The application uses the credentials to establish connectivity to the database directly

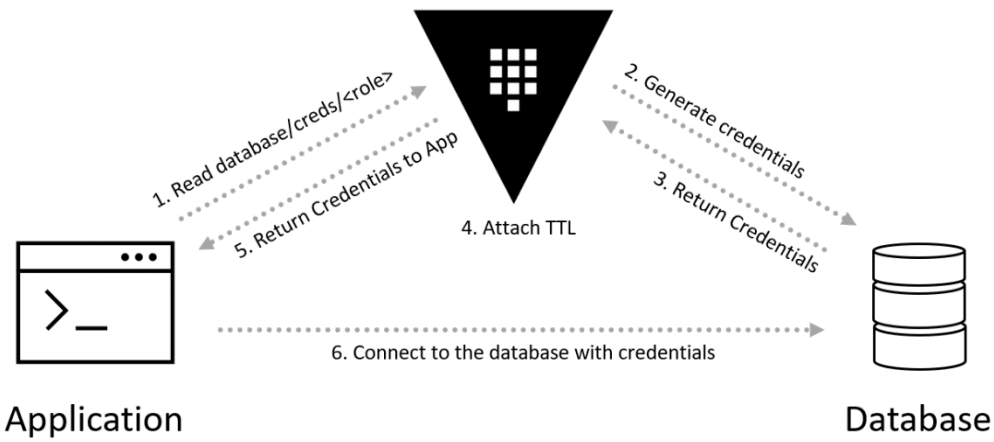



Figure 4-53

## *Configuring the Database Secrets Engine*

### Enable the Secrets Engine

The first step to using the database secrets engine is to enable it. The database secrets engine can be enabled using the UI, CLI, or API. To enable the database secrets engine at the default path using the CLI, use the command found in Figure 4-54.

```
$ vault secrets enable database
Success! Enabled the database secrets engine at:
database/
```

Figure 4-54

**Configure the Secrets Engine**

Before creating a role, Vault needs to understand what database it will generate credentials against, how to connect to it, and what credentials to use to create users. This configuration is required for every separate database instance that Vault integrates with to generate credentials. For example, if an organization has separate database instances for production, integration, and development, each database requires a separate configuration.

```
$ vault write database/config/dev-data \
    plugin_name=mysql-database-plugin \
    connection_url="{{username}}:{{password}}@tcp(dev-
       mysql01.example.com:3306)/" \
    allowed_roles="dev-app01, dev-app02" \
    username="svc_vault" \
    password="a1b2c3d4e5f6g7h8i9"
```

Figure 4-55

Figure 4-55 displays the command to create a configuration for a MySQL database used in a development environment. In this example, Vault connects to a database at *dev-mysql01.example.com:3306* and uses the provided credentials (*svc-vault*) to authenticate and generate the new user. The credentials provided to Vault must have sufficient privileges on the database to create logins, create users, grant permissions, and manage processes. The database platform may have built-in roles that provide these permissions, or they can be granted manually when the initial credentials are created for Vault (*svc-vault*). If Vault cannot connect and validate the connection, the write request for the configuration fails.

The configuration will also only permit two roles to use this configuration: *dev-app01* and *dev-app02*. If additional roles are created for this database, the configuration must be updated, or Vault will deny the request with an error, as shown in Figure 4-56.

```
Error reading database/creds/dev-app01: Error making API
request.

URL: GET
http://vault.example.com:8200/v1/database/creds/dev-app01
Code: 500. Errors:

* 1 error occurred:
        * "dev-app01" is not an allowed role
```

Figure 4-56

Even though the configuration command includes database credentials, the full credentials cannot be read back by reading the configuration for security purposes. Vault displays the username used to establish connectivity on a read request but does not display the password. This approach prevents a Vault operator from gaining unauthorized credentials to the database.

**Create a Role**

After the database configuration has been written, one or more roles need to be defined. Each role specifies the database to use along with the command Vault uses to create the database user. The command includes some generic database platform-specific statements to create the new user and additional grants for the new user. For example, Vault may create a new user and only permit the new user SELECT, EXECUTE, and INSERT on a database named *data*. Figure 4-57 shows an example of creating a role for the database configuration created in the section prior.

```
$ vault write database/roles/dev-app01 \
    db_name=dev-data \
    creation_statements="CREATE USER '{{name}}'@'%'
      IDENTIFIED BY '{{password}}';GRANT SELECT, .
      EXECUTE,INSERT ON data.* TO '{{name}}'@'%';" \
    default_ttl="4h" \
    max_ttl="12h"
```

```
Success! Data written to: database/roles/dev-app01
```

Figure 4-57

**Generate Credentials**

With the prerequisites completed, an application can now generate credentials for the database. The credential generation is performed by issuing a read operation against the *creds/* prefix for the role. For example, the Figure in 4-58 displays this read request for the role created above.

```
$ vault read database/creds/dev-app01
Key                Value
---                -----
lease_id           database/creds/dev-app01/GyZs7rn…
lease_duration     4h
lease_renewable    true
password           A1a-AFzMW1NqKKfcrjoI
username           v-dev--f40BjKiYs
```

Figure 4-58

The response includes the username and password that an application or end-user can use to connect to the database and interact with data. Keep in mind that these credentials are still bound by the permissions configured in the *creation_statements* for the *dev-app01* role.

## Transit Secrets Engine

Most products on the market that Vault competes with have a primary focus on storing and managing secrets. However, Vault has many tricks up its sleeve, and one is the ability to provide encryption as a service. This differentiating feature allows organizations to realize additional value from their investment in Vault. By enabling and using the Transit secrets engine, applications can send clear-text data to Vault to be encrypted. While Vault does not store the encrypted data, it does respond with the AES-256 encrypted data based on the selected encryption key. The application can then store the data at rest in a place of its choosing. When the data needs to be retrieved, the application can send the encrypted data to Vault to be decrypted on the fly.

There are many benefits of using the Transit secrets engine. The obvious benefit is data security. Organizations can now securely store user data with confidence wherever they choose, knowing that the data is secure in the event of a security incident. Using Vault for data encryption eliminates the reliance on a platform or storage-based solution, as these solutions often manage both the encryption key and the data. Separating the encryption process from the storage means there is less chance that the data can be compromised.

Another benefit of entrusting Vault with encryption services is reducing the administrative overhead of managing a dedicated solution for data encryption. Since Vault customers have usually invested in the Vault ecosystem through licensing or training, organizations do not need to invest in additional software. Once Vault operators enable the Transit secrets engine, application developers can quickly integrate applications using the fully-featured API. In most cases, Vault is already classified as a Tier 0 application and requires the highest SLA. Therefore, little to no changes are needed from a Vault deployment perspective to support this new functionality.

At a very high level, the Transit secrets engine works similarly to other data encryption solutions. A named encryption key is created and used to encrypt selected data. The same key is referenced for a decryption operation as well. Keys can easily be rotated, and encrypted data can be rewrapped using the new key. Vault also provides the ability to limit the minimum version of a key for encryption and decryption operations to prevent old copies of ciphertext from being decrypted.

When data is sent to Vault for encryption, it must be base64-encoded, which means that Vault Transit supports both plaintext and binary files, such as a JPG or PDF. When data is decrypted, Vault returns the data as base64-encoded, as well. It is up to the end-user or application to handle the encoding and decoding of the data.
It should be noted that the Transit secrets engine is compute-intensive. Encryption and decryption operations generate a heavy load on the CPU of a Vault node. Care should be taken to provide ample computing power for Vault nodes using the Transit secrets engine. Organizations choosing to provision Vault on a public cloud provider should choose compute-optimized hardware types, such as the C-series in AWS, the F-series in Azure, and C2 machine types in GCP.

## *Configuring the Transit Secrets Engine*

### Enable the Transit Secrets Engine

The Transit secrets engine must be enabled in Vault before it can be used. The secrets engine can be enabled using any of the available interfaces, including the UI, API, and CLI. To enable the Transit secrets engine using the CLI, use the command found in Figure 4-59.

```
$ vault secrets enable transit
Success! Enabled the transit secrets engine at: transit/
```

Figure 4-59

### Create an Encryption Key

Once the Transit secrets engine is enabled, an encryption key must be created before Vault can encrypt data. It is common to create a unique key for each application that interacts with the Transit secret engine. By creating individual keys, Vault operators can permit applications to encrypt and decrypt data using the assigned key. Limiting key usage can also prevent a bad actor from obtaining data from one system and using another system to decrypt it.

Vault provides various options when creating a new encryption key. Vault operators can select which type of key to create, such as AES-GCM, ECDSA, and RSA. While the default selection is *aes256-gcm96*, the proper key type should be selected based upon the application's requirements and the key usage. Keys can also be marked as exportable, as well as support convergent encryption. It is best practice to generate non-exportable keys to help ensure they do not fall into the wrong hands.

The command displayed in Figure 4-60 can create a new encryption key with the default settings using the CLI.

```
$ vault write -f transit/keys/vault-book
Success! Data written to: transit/keys/vault-book
```

Figure 4-60

### View Encryption Key Properties

Once a key has been created, a Vault operator may need to view the properties of the key. These properties provide details such as the number of keys, the type of key, and other supporting information. To read the properties of a key, use the command found in Figure 4-61.

```
$ vault read transit/keys/vault-book

Key                       Value
---                       -----
allow_plaintext_backup    false
deletion_allowed          false
derived                   false
exportable                false
keys                      map[1:1598137750]
latest_version            1
min_available_version     0
min_decryption_version    1
min_encryption_version    0
name                      vault-book
supports_decryption       true
supports_derivation       true
supports_encryption       true
supports_signing          false
type                      aes256-gcm96
```

Figure 4-61

### Delete an Encryption Key

Vault operators may need to delete encryption keys from the Transit secrets engine as part of system maintenance or cleanup. However, this action can prove to be a catastrophic operation if an organization still has data that was encrypted by the deleted key. If the key is deleted, Vault can no longer decrypt that data. Therefore, a Vault operator must explicitly and deliberately update the *deletion_allowed* key parameter from *false* to *true*.

Use the command shown in Figure 4-62 to permit the deletion of a key, and subsequently delete a key.

```
$ vault write transit/keys/vault-book/config \
    deletion_allowed=true
Success! Data written to: transit/keys/vault-book/config

$ vault delete transit/keys/vault-book
Success! Data deleted (if it existed) at:
transit/keys/vault-book
```

Figure 4-62

Since this action can cause significant data loss, it is recommended to restrict this action to Vault operators who manage the Vault deployment. Organizations should also configure a notification when this event is written to the audit log. Data aggregation solutions should be configured to search for "deletion_allowed." An example of an event for setting the *deletion_allowed* parameter to *true* can be found in Figure 4-63.

```
"request": {
  "id": "336a7d4c-4c0d-0317-8ac9-79408f17482e",
  "operation": "update",
  "mount_type": "transit",
  "client_token": "hmac-sha256:31410cc098……",
  "client_token_accessor": "hmac-sha256:8874398063e5……",
  "namespace": {
    "id": "root"
  },
  "path": "transit/keys/vault-book/config",
  "data": {
    "deletion_allowed": "hmac-sha256:f8ebb2a5b……"
  },
  "remote_address": "vault.example.com"
},
"response": {
  "mount_type": "transit"
```

```
   }
}
```

Figure 4-63

**Encrypt Data**

Vault can encrypt data using any of its interfaces. However, modern applications traditionally use the API for integrations. These applications must be properly authenticated to Vault before sending encrypt or decrypt requests to Vault. The token used for these operations must have permission to use the key specified. Otherwise, the request results in an error.

When encrypting data, the request must include the *name* of the encryption key to encrypt against and the *plaintext* string, at a minimum. Requests can also specify the key version to use if the latest version should not be used. The command found in Figure 4-64 displays the command to encrypt base64-encoded data. The output has been truncated.

```
$ base64 <<< "running vault in production"
cnVubmluZyB2YXVsdCBpbiBwcm9kdWN0aW9u

$ vault write transit/encrypt/vault-book \
    plaintext="cnVubmluZyB2YXVsdCBpbiBwcm9kdWN0aW9u"
Key            Value
---            -----
ciphertext     vault:v1:t+W7cp6dfk8+oYCCvgKAasb……
key_version    1
```

Figure 4-64

**Decrypt Data**

Once data has been successfully encrypted with Vault, it is almost certain that an organization will want to decrypt the data for future use. Applications or end-users decrypting data need to provide the *ciphertext* and the *name* for the key for decryption. A decryption request results in a base64-encoded response, and the user or application is responsible for decoding the output to reveal the useable data.

The command depicted in Figure 4-65 shows how to decrypt ciphertext using the transit secrets engine and subsequently decoding the response to reveal the original encrypted value.

```
$ vault write transit/decrypt/vault-book \
   ciphertext=vault:v1:t+W7cp6dfk8+oYCCvgjuky9n6KAasbaHxs4A
   V6A8DqbYOWh7Xi0++Uoni+FCFSRmMfMuy1Wyeg==
Key          Value
---          -----
plaintext    cnVubmluZyB2YXVsdCBpbiBwcm9kdWN0aW9u


$ base64 --decode <<< "cnVubmluZyB2YXVsdCBpbiBwcm9kdWN0aW9u"
running vault in production
```

Figure 4-65

## Cloud Provider Secrets Engine

As organizations continue migrating applications to the public cloud, the opportunity to integrate Vault directly with a public cloud provider is one of the most common requirements. Out of the box, Vault supports major cloud platforms such as AliCloud, AWS, Azure, and Google Cloud. Rather than create a long-lived account on its public cloud of choice, organizations can rely on Vault to create on-demand access to the public cloud provider by way of a secrets engine.

The cloud provider secrets engines have many use cases. However, one of the most prominent use cases is the deployment of cloud infrastructure using HashiCorp Terraform. Terraform only needs valid credentials for the cloud provider during *terraform plan* and *terraform apply* operations, and Vault can quickly provide the credentials when these workflows are executed. Once a Terraform action is complete, credentials are no longer needed and can be revoked. Additional information on the Terraform use case can be found in the chapter on Terraform Integration. Other use cases for a cloud platform secrets engine include integrating a pipeline tool such as Jenkins, an on-premises application reading data from an S3 bucket, or an end-user that needs access to cloud resources.

Configuration and integration with public cloud providers may differ depending on where Vault is deployed and the use case being configured. For example, the AWS secrets engine configuration might look different for on-premises Vault deployments versus Vault deployments on AWS. This difference is primarily dependent on the use of integrations, such as IAM service roles, to permit Vault to communicate with the identity and access management service. Vault deployments running on a cloud platform have the advantage of using these services, while on-premises deployments do not.

The general workflow and configuration for each of the cloud provider secrets engines are similar. The AWS secrets engine, for example, supports three different types of configurations to retrieve credentials: the *iam_user*, *assumed_role*, and *federation_token*. Organizations with a single account may benefit from the simplicity of the iam_user type. In contrast, organizations with multiple accounts can use the *assumed_role* credential type since this configuration method can easily handle cross-account authentication. The configuration and workflow for the *assumed_role* type are more complicated to configure than the *iam_user* type. Figure 4-66 displays the workflow for the AWS secrets engine using the *iam_user* credential type.

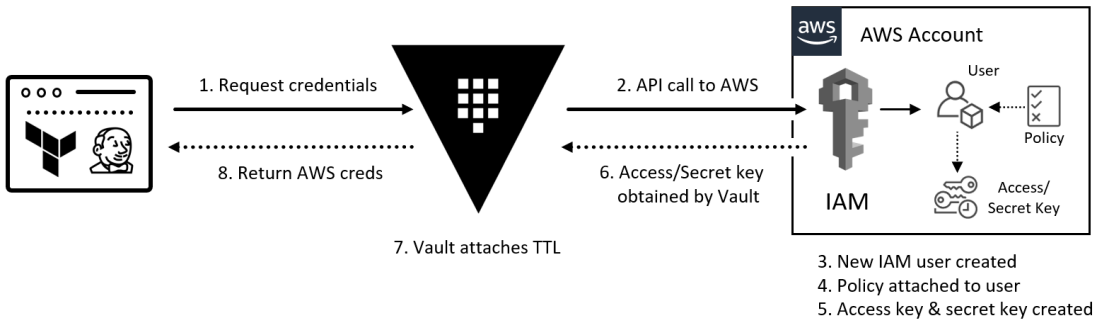



Figure 4-66

## *Configuring the AWS Secrets Engine*

### Enable the AWS Secrets Engine

The first step, once again, to using any secrets engine is to (you guessed it) enable it in Vault. The AWS secrets engine can be enabled using any of the Vault interfaces. To enable

the AWS secrets engine on the default path using the CLI, use the command found in Figure 4-67.

```
$ vault secrets enable aws
Success! Enabled the aws secrets engine at: aws/
```

Figure 4-67

**Configure the AWS Secrets Engine**

Before the secrets engine can be configured, a few prerequisites need to be completed to use the *iam_user* credential type. Vault needs permission to call the AWS APIs to perform certain functions for managing the dynamic IAM users. These permissions include AWS actions such as creating an IAM user, creating and attaching a policy, and a few other AWS actions. These permissions can be granted by way of an access key and secret key generated from an IAM user with the appropriate policy. Alternatively, if Vault is deployed on AWS, an IAM service role can grant these permissions instead.

Figure 4-68 displays the AWS IAM policy that needs to be created and assigned to the user or attached to an IAM EC2 service role. Keep in mind that these permissions are only applicable to Vault operations. The credentials a Vault client receives will be bound by a different set of permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachUserPolicy",
        "iam:CreateAccessKey",
        "iam:CreateUser",
        "iam:DeleteAccessKey",
        "iam:DeleteUser",
        "iam:DeleteUserPolicy",
        "iam:DetachUserPolicy",
        "iam:ListAccessKeys",
```

```
        "iam:ListAttachedUserPolicies",
        "iam:ListGroupsForUser",
        "iam:ListUserPolicies",
        "iam:PutUserPolicy",
        "iam:AddUserToGroup",
        "iam:RemoveUserFromGroup"
      ],
      "Resource":
   ["arn:aws:iam::123456789101:user/vault-*"]
    }
  ]
}
```

Figure 4-68

Once the policy is created, an IAM user needs to be created with the above policy attached. Access credentials need to be generated and are used to configure the secrets engine. Figure 4-69 demonstrates the configuration for the AWS secrets engine. Note that the value for *access_key* and *secret_key* is derived from the IAM user account created in AWS.

```
$ vault write aws/config/root \
    access_key= AKIAIOSFODNN7EXAMPLE \
    secret_key= wlrXUtnFEMI/KNG/bPxRfiCYEXAMPLEKEY \
    region=us-east-1
```

Figure 4-69

Vault includes a command to rotate the IAM access key and secret key programmatically. Organizations will find this beneficial in multiple ways, such as facilitating the frequent rotation of the AWS credentials to meet corporate security policies. It is also helpful to ensure that no human user knows the credentials. The initial credentials can be added manually, and the command can be issued to rotate them immediately. Vault will generate a new access key and secret key on the IAM user, configure the AWS secrets engine to use it, and delete the old security credentials from AWS. This command can be found in Figure 4-70.

```
$ vault write -f aws/config/rotate-root
Key           Value
---           -----
access_key    AKIAIOSFODNN7EXAMPLE
```

Figure 4-70

**Create a Role**

Each unique end-user or application use case requires a different role to be created. This role maps the requested AWS credentials to an AWS IAM policy and should only permit the actions required by the requester. When creating a role, there are two options to attach a policy to the newly created user: have Vault create a new policy or attach an existing managed policy.

There are pros and cons to each of the options, and organizations should evaluate both to determine which strategy works best. The difference between the two options centers around where the IAM policy is authored and maintained. Having Vault create the policy reduces the reliance on the cloud team to create or modify policies and eliminates the need for Vault operators to make changes in AWS. However, the Vault operator needs to be proficient with AWS policies or work with the cloud team to build them. Using an existing managed policy could accelerate the integration with AWS, but any changes would need to be made directly in AWS.

Figure 4-71 illustrates the IAM policy document and how to create a role using the local document. In this case, Vault creates and manages the IAM policy. The command can accept the *policy_document* using a heredoc format or referencing a local file containing the IAM policy, as shown below. When Vault removes the IAM user from AWS, Vault deletes the associated IAM policy since Vault uses an inline policy, not a managed policy.

```
$ cat developer_policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ec2:*",
```

```
      "Resource": "arn:aws:ec2:*:*:instance/*",
      "Condition": {
        "StringEquals": {
          "ec2:ResourceTag/Environment": "dev"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": "ec2:DescribeInstances",
      "Resource": "*"
    }
  ]
}

$ vault write aws/roles/developer \
    credential_type=iam_user \
    policy_document=@developer_policy.json
Success! Data written to: aws/roles/developer
```

Figure 4-71

To create a new role using an existing managed policy, Vault must be given the policy ARN provided by AWS. Figure 4-72 shows how to create a role using the pre-existing policy.

```
$ vault write aws/roles/developer \
    credential_type=iam_user \
    policy_arns=arn:aws:iam::123456789101:role/devRole
```

Figure 4-72

**Generate Credentials**

With all the prerequisites completed, the end-user or application can now generate credentials for the configured AWS account. The request sent to Vault should dictate the name of the role to generate credentials against using the *creds/* prefix. Figure 4-73 exhibits the command used to generate credentials against the *developer* role.

```
$ vault read aws/creds/developer
Key                Value
---                -----
lease_id           aws/creds/developer/iHTRtW1sAZ……
lease_duration     768h
lease_renewable    true
access_key         AKIAIOSFODNN7EXAMPLE
secret_key         Gbt0y7Kkb1UkerbtBGzpwm3v5Q0……
security_token     <nil>
```

Figure 4-73

Once the TTL expires for the generated credentials, Vault proceeds to delete the user on AWS. This workflow allows organizations to use the AWS secrets engine without fear of creating technical or security debt.

**Delete Leases**

Organizations need to manage existing leases and credentials that have already been generated against the cloud provider. If an organization believes that Vault may be compromised, it may choose to delete all AWS credentials and leases to ensure all existing credentials are invalidated. Vault can also revoke individual leases if needed. To delete a single AWS lease and associated credential, use the command found in Figure 4-74.

```
$ vault lease revoke aws/creds/developer/iHTRtW1sAZ……
All revocation operations queued successfully!
```

Figure 4-74

To delete all leases associated with the AWS secrets engine, use the command displayed in Figure 4-75.

```
$ vault lease revoke -prefix aws/creds
All revocation operations queued successfully!
```

Figure 4-75

## PKI Secrets Engine

"Dance like nobody is watching. Encrypt like everyone is." This quote from Werner Vogels of AWS brilliantly sums up how organizations should think about security. Historically, organizations approached security using the castle and moat model when protecting their infrastructure. Having a hardened exterior gave the impression that security could be somewhat relaxed inside since exposure from the inside was believed to be improbable.

However, the migration to the public cloud began altering this mindset, not only for security teams but also for infrastructure teams. Gone are the days of placing a firewall at the edge of the infrastructure and hoping everything behind it remains protected. Organizations are quickly adopting an "encrypt everywhere" mantra in the public cloud and private cloud where this hypothetical moat is no longer a reliable protection method.
While using encryption techniques is not a new strategy for organizations, using technologies to protect internal, trusted traffic has gained momentum. Most experienced IT professionals have witnessed this change, where traffic between the client and front-end application was encrypted using TLS. However, the traffic was often sent in cleartext as it was traversing the trusted internal network. This strategy is generally no longer accepted, especially with workloads running in the public cloud where ownership and control of the network hardware are given up for simplicity and a pay-as-you-go operational model. As a result, organizations deploy and maintain an internal public key infrastructure (PKI) to generate TLS certificates to enable trusted and encrypted communication between any services.

In addition to the need to protect sensitive data in transit and at rest, there is also an element of service reliability and uptime relating to PKI management. Service disruptions often occur due to expired certificates that the support team failed to rotate on time.

All these issues stem from a problem with the traditional PKI provisioning process. The PKI process often requires manual steps to procure certificates, and more often than not, it just takes too long. The traditional PKI certificate request usually requires a service ticket to be submitted to the proper team, putting the requester at the mercy of the security or identity team to complete the request and return the certificate. How can a business keep up with market demands and accelerate the deployment of applications when the procurement of a TLS certificate takes weeks? Enter the Vault PKI secrets engine!

The PKI secrets engine enables the Vault service to generate dynamic X.509 certificates for internal workloads with minimal effort. Through the PKI secrets engine, applications can programmatically obtain a TLS certificate at runtime without the hassle of manually generating private keys or CSRs. By enabling this capability, organizations can now allocate a unique certificate to each instance of an application versus sharing the certificate across instances. This strategy benefits the organization's security posture in multiple ways. It allows certificates to have short TTLs because certificates are now ephemeral. Because of this, certificate revocations are virtually eliminated since new certificates can be quickly generated. It also allows for automated certificate renewal, making outages caused by certificate expiration a thing of the past.

Configuring the PKI secrets engine is relatively simple, and it can be configured as either a root or an Intermediate CA. Most organizations use Vault as an Intermediate CA since a root CA is generally offline for security purposes. Since most organizations already have an existing PKI structure, Vault can be configured to be a trusted intermediate in its certificate chain. Multiple PKI secrets engines can be enabled in Vault, and each can be configured as needed. As with other secrets engines, Vault treats them as entirely separate entities, meaning that Vault can have multiple unrelated PKI secrets engines running in parallel on the same Vault cluster. For example, Figure 4-76 shows a single Vault cluster running two separate intermediate CAs, with the root existing elsewhere in the organization.

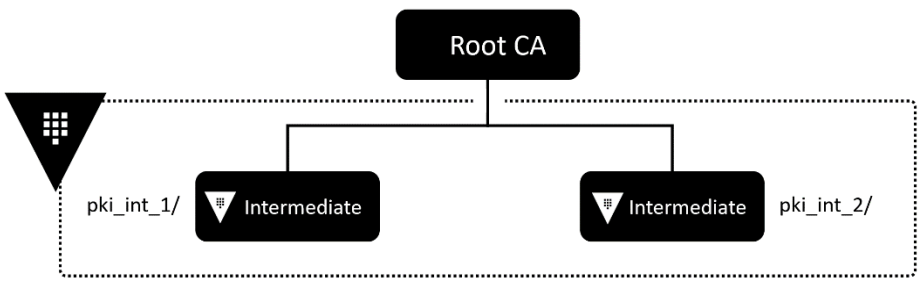



Figure 4-76

## *Configuring the PKI Secrets Engine*

### Enable the PKI Secrets Engine

Before certificates can be generated from Vault, the PKI secrets engine must be enabled. The PKI secrets engine can be enabled using any Vault interface, including the API, CLI, or

UI. Use the command in Figure 4-77 to enable the PKI secrets engine on the custom path, as displayed in the example above.

```
$ vault secrets enable -path=pki_int_1 pki
Success! Enabled the pki secrets engine at: pki_int_1/
```

Figure 4-77

**Configure the PKI Secrets Engine**

Once the PKI secrets engine is enabled, the secrets engine needs to be configured as an intermediate. The resulting CSR needs to be signed by the organization's root certificate. To configure the secrets engine as an intermediate and generate the CSR, use the command found in Figure 4-78.

```
$ vault write pki_int_1/intermediate/generate/internal \
    common_name="example.com Intermediate"
Key    Value
---    -----
csr    -----BEGIN CERTIFICATE REQUEST-----
MIICaDCCAVACAQAwIzEhMB8GA1UEAxMYZXhhbXBsZS5jb20gSW50ZXJtZWRpYXRl
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsc2A+t4BEp/9/r7AMICe
4dlSlOEjWsRUQW75fitvc6rY9kEdLhH5zVIdzdDbhEH6zO30cQdp7HvUqX9a...
-----END CERTIFICATE REQUEST-----
```

Figure 4-78

The command to generate the CSR can also be executed to parse the output using *jq* and save the resulting certificate to a local file. This command is shown in Figure 4-79.

```
$ vault write -format=json \
    pki_int_1/intermediate/generate/internal \
    common_name="example.com Intermediate" \
    | jq -r '.data.csr' > pki.csr
```

Figure 4-79

Now that the intermediate certificate has been created, the next step is to sign the intermediate certificate with the root certificate. This process happens outside of Vault if the root CA is running elsewhere. Due to the sensitive nature of this process, it may take a considerable amount of time to complete. This amount of time depends on where the root CA is located and if access is easily obtained by security personnel.

Once the root CA has signed the intermediate certificate, the certificate needs to be imported back into Vault. Importing the response from the root CA completes the process of configuring Vault as an intermediate CA. To import the certificate into Vault, use the command found in Figure 4-80.

```
$ vault write pki_int_1/intermediate/set-signed
    certificate=@pki.cert.pem
Success! Data written to: pki_int_1/intermediate/set-signed
```

Figure 4-80

**Create a Role**

With all the prep work now completed, the PKI secrets engine can be configured with a role that will be used to generate certificates. The role will also be used in Vault policies to permit users access to generate certificates from only the roles to which they are authorized. Each role has a set of configurable parameters that include:

- ***allowed_domains***: define the domains that certificates can be generated with (such as example.com)
- ***allow_base_domains***: allow the creation certificates matching the actual domain themselves (example.com)
- **allow_subdomains**: permit the creation of certificates that with CNs that are subdomains (vault.example.com)
- **allow_glob_domains:** allow the creation of certificates that include a wildcard (web-app*.example.com

The command found in Figure 4-81 can be used to create a new role.

```
$ vault write pki_int_1/roles/vaultbook \
```

```
   allowed_domains="vaultbook.com" \
   allow_subdomains=true \
   max_ttl="24h"
Success! Data written to: pki_int_1/roles/vaultbook
```

Figure 4-81

**Request a Certificate**

The PKI secrets engine is now completed and is ready to accept certificate requests from Vault clients. These certificates can be requested from any of the Vault interfaces (yes, even the UI). To request a new certificate from Vault using the role configured above, execute the command found in Figure 4-82. The value of each certificate or key in the response has been shortened for brevity. Notice how the response includes the ca_chain, the certificate, the issuing_ca certificate, and the private key. The serial number is also included, which can be used to look up information about the certificate or revoke the certificate if needed.

Note that *private* information in the response is only offered on the initial response, and it is up to the requester to parse this information and import the certificate and key to the service where it will be used. The private key cannot be retrieved again.

```
$ vault write pki_int_1/issue/vaultbook \
   common_name="best.vaultbook.com" ttl="24h"
Key                 Value
---                 -----
ca_chain            [-----BEGIN CERTIFICATE-----
MIIDpjCCAo6gAwIBAgIUJJ/yDqCJyav23PAJhV+zZTfUlygwDQYJKoZIhvcNAQEL
BQAwFjEUMBIGA1UEAxMLZXhhbXBsZS5jb20wHhcNMjAxMDEyMDEwMjA1WhcNMjUx
MDExMDEwMjM1WjAtMSswKQYDVQQDEyJleGFtcGxlLmNvbSBJbnRlcm1lZGZ...
-----END CERTIFICATE-----]
certificate         -----BEGIN CERTIFICATE-----
MIIDajCCAlKgAwIBAgIUPoI8lYyzMz+rcEFHmtwykzqtlvMwDQYJKoZIhvcNAQEL
BQAwLTErMCkGA1UEAxMiZXhhbXBsZS5jb20gSW50ZXJtZWRpYXRlIEF1dGhvcml0
eTAeFw0yMDEwMTIwMTIxMzVaFw0yMDEwMTMwMTIyMDVaMB0xGzAZBgNVBAMT...
-----END CERTIFICATE-----
expiration          1602552125
issuing_ca          -----BEGIN CERTIFICATE-----
MIIDpjCCAo6gAwIBAgIUJJ/yDqCJyav23PAJhV+zZTfUlygwDQYJKoZIhvcNAQEL
```

```
BQAwFjEUMBIGA1UEAxMLZXhhbXBsZS5jb20wHhcNMjAxMDEyMDEwMjA1WhcNMjUx
MDExMDEwMjM1WjAtMSswKQYDVQQDEyJleGFtcGxlLmNvbSBJbnRlcm1lZGlh...
-----END CERTIFICATE-----
private_key         -----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAvN5Idir4sBeX+EUxF+gUgmBgUIb+TcXtPSsh7sqzy2+4zQbk
MotgR2jePVc0mK7jlEGN5yUe5iyi5jMWMPQWjYSJhWf1aCwwaYKN1EK2byTp...
-----END RSA PRIVATE KEY-----
private_key_type    rsa
serial_number
3e:82:3c:95:8c:b3:33:3f:ab:70:41:47:9a:dc:32:93:3a:ad:96:f3
```

Figure 4-82

## *Vault Policies*

Modern platforms such as Vault must include a role-based access control (RBAC) system to be considered enterprise-ready. RBAC is essential because it allows a way to grant or deny access to certain features and functions of the platform, depending on the client's role or requirements. In Vault, this functionality is obtained by way of Vault policies. Vault policies provide a way to regulate access to specific paths or actions for any consumers of the Vault service. Once clients are authenticated and have obtained a token, the attached policy dictates what a client can and cannot do.

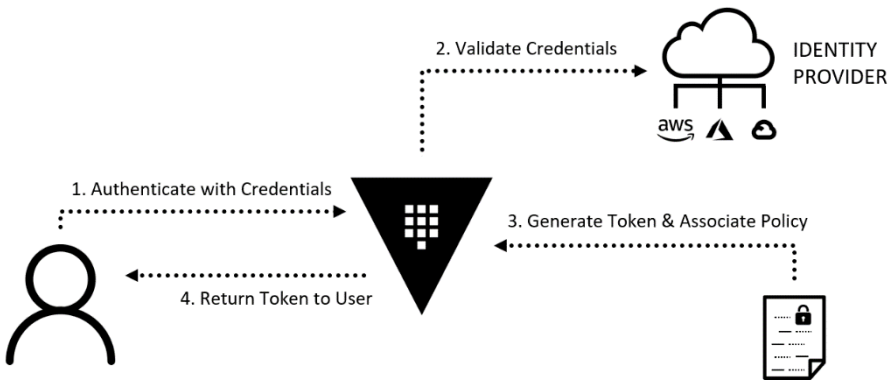



Figure 4-83

Vault policies are written using declarative statements either in HCL or JSON syntax. Policies contain an implicit deny, which means that any actions not explicitly granted to the Vault client are denied by default. Policies are also cumulative, and the capabilities are additive. This design implies that a token with multiple policies has the combined capabilities defined in each policy. Vault policies also support an explicit deny, which always takes precedence over other capabilities.

> **Note:** HashiCorp often refers to Vault policies as ACL policies in its documentation and on the Learn platform. These terms are synonymous, although this book will always refer to them as Vault policies.

Vault policies define both the capability (action) and the path to which that capability applies. As HashiCorp notes in its documentation, these capabilities usually map to the HTTP verb,

not the underlying action taken as a result of a request. Vault supports multiple capabilities, which help provide fine-grained control to a path. The capabilities that Vault supports and the associated HTTP verb include:

- Create (POST/PUT)
- Read (GET)
- Update (POST/PUT)
- Delete (DELETE)
- List (LIST)
- Deny
- Sudo

The basic form of a Vault policy includes a path and the associated capabilities permitted on the path. More complex policies can include variable replacements and parameters to define precise control for tokens. This approach allows organizations to permit definitive actions to a path while following the principle of least privilege.

Out of the box, Vault includes two default policies: the root policy and the default policy. The root policy provides superuser capabilities and cannot be modified nor deleted. A token attached to the root policy has unfettered access to Vault, so it should be used with caution. Conventionally, the root policy is not attached to a token for day-to-day operations and only used to configure Vault initially or in recovery scenarios. As a security best practice, organizations should not use tokens attached to the root policy on a day to day basis. On the other hand, the default token provides a base level of standard permissions and is attached to all tokens. This policy permits each token the ability to look up data about itself and the use of cubbyhole. The default policy can be modified, but it cannot be deleted. Vault can explicitly be configured not to attach the default policy.

Beyond the default policies, organizations are expected to create specific policies based on user and application requirements. Each Vault client and use case may require a separate policy. For example, a Vault operator policy may grant the ability to create and delete secrets engines. In contrast, a policy written for an application may only permit read access to a specific path. Organizations should apply standardizations across their policies, employ a naming standard, and composing policies to be reusable where possible. Policies should include comments to provide documentation and to justify each permission granted. Role-based policies often found in organizations include:

- **Vault Administrator** – full access to Vault
- **Vault Operator** – general Vault operations (auth, secrets, policies)
- **Security Team** – consume audit logs, review and manage policies
- **Developer** – read-only access to specific paths required by their applications
- **Application Owners** – write access to specific paths for their application
- **Database Administrators** – database secrets engine configuration

Beyond these role-based policies for the users interacting with Vault, it is common for organizations to have many more policies specifically for application integrations. Each application may have unique needs, requiring individual policies to be authored and maintained. Multiple policies may also be written for each software development lifecycle for each application, such as development, quality assurance, staging, and production. Policies may also be needed for automation workflows, such as a CI/CD pipeline or other application build tools.

As policies are created, organizations may find policy management is better suited in a code repository solution, such as GitHub or GitLab. By centralizing policy management in a code repo, standardization and change management becomes much easier to manage. Vault operators or application teams can submit pull requests based on recommended changes, and the organization's security team can approve or deny the request based on established corporate policies.

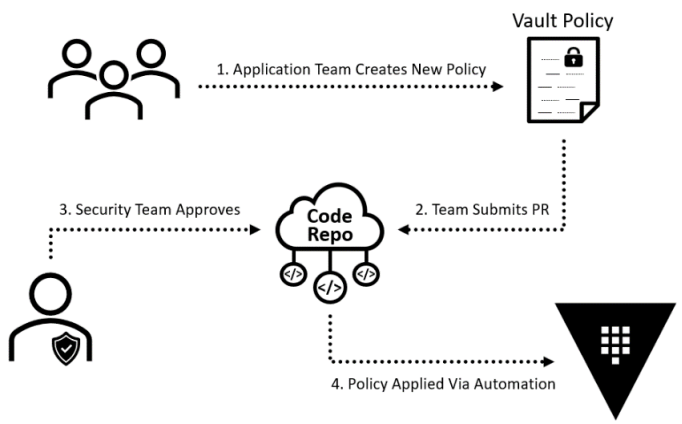



Figure 4-84

## Anatomy of a Vault Policy

A Vault policy is made up of two components, including the path and declared capabilities. Since all Vault functions are path-based, policies can permit access to a secret, obtain a token, or even perform administrative functions. After the desired path is determined, the capabilities are defined regarding what actions the token can perform against the path. Vault evaluates the policy using path-based matching to test the capabilities for each request.

The policy in Figure 4-85 displays a simple policy that permits a token to read from a specific path. Note that the policy only permits a read against this path; the token cannot update or delete the secret. Additionally, the token cannot interact with any other path in Vault.

```
#Permit book author to read secret data

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

Figure 4-85

As more permissions are needed for the token, additional grants can be appended to the policy to permit access to other paths. The policy example in Figure 4-86 permits the token to read a secret from the K/V store, modify a database role for the database secrets engine, and update existing users (such as a password) for the userpass auth method.

```
#Permit book author to read secret data
path "secret/vault/book" {
  capabilities = ["read"]
}

#Permit access to create and modify the database role
path "database/role/webapp-01" {
  capabilities = ["create", "update"]
}
```

```
#Permit administration of local users
path "auth/userpass/users/*" {
  capabilities = ["list", "update"]
}
```

Figure 4-86

## Building Policies

Beyond the simple path and capability options, additional features can be added to make policies reusable and more flexible while still limiting the scope of the policy. These additional features include variable replacements, parameter constraints, a wildcard (*), and the "+" used to indicate any number of characters within a given path.

### *Using the Wildcard (*)*

Out of these more advanced features, the wildcard is the most popular and is heavily used in most Vault environments. When using a wildcard, Vault needs only to match the prefix and not the exact path when evaluating the request. The wildcard allows Vault policies to provide the defined capabilities for all paths beneath the designated path. The policy illustrated in Figure 4-87 shows how a wildcard can permit token access to any path beyond the prefix defined in the policy. The wildcard (*) can *only* be used at the end of a path. In this example, any data written beyond the "*secret/vault/*" prefix can now be read by the token. For instance, data stored at "*secret/vault/authors*" and "*secret/vault/book/cover*" can be accessed with the addition of the wildcard.

```
#Permit book publisher to read all Vault secret data

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

Figure 4-87

### *Parameter Constraints*

The use of parameter constraints enables a policy to restrict a token's capabilities further than just the defined path. Policy owners can use these options to permit or deny specific keys or values for a Vault client's write operations. Based on experience, this capability is frequently used to ensure that standards are being followed or only to permit the inclusion of pre-defined values. Options that can be used for parameter constraints include:

- required_parameters
- allowed_parameters
- denied_parameters

An example of a policy using the *required_parameters* option can be found in Figure 4-88. This policy permits an associated token to create any key/value pair beneath the *secret/vault/authors* path if the key is either *headshot* or *bio*. For example, the user could create a new key/value pair at the path *secret/vault/ authors/cover* and create the key/value pair of *headshot=acquired*. In contrast, the token would not be permitted to create *chapter4=vaultpolicies* on that same path.

```
#Permit book publisher to create Vault authors

path "secret/vault/authors/*" {
  capabilities = ["create"]
  allowed_parameters = {
    "headshot" = []
    "bio"      = []
  }
}
```

Figure 4-88

To signify any value for a key, use the glob (*) to designate a wildcard. To signify a wildcard for the value, use empty brackets [ ], as shown above. If the glob is used for the parameter, the only value that can be used is [ ]. In the example displayed in Figure 4-89, the user can create *secret/vault/chapters* with any parameter if the parameter's value is *draft* or *completed*.

```
#Permit book publisher to create Vault authors

path "secret/cloud_team/*" {
  capabilities = ["create", "read", "list", "update"]
  allowed_parameters = {
    "book" = ["draft", "completed"]
    "*"    = []
  }
}
```

Figure 4-89

### *Using Deny*

Although Vault policies contain an implicit deny, there may be instances where access needs to be denied to specific paths to which a user might have access. In this case, a policy owner may use the *deny* capability to prohibit access to a path explicitly. For example, a user may have access to the broader *secret/cloud_team/** path but be denied from creating a parameter to store a password. This scenario is depicted in Figure 4-90.

```
#Permit access to cloud team K/V

path "secret/cloud_team" {
  capabilities = ["create"]
  denied_parameters = {
    "password" = []
  }
}
```

Figure 4-90

## Managing Policies

Vault provides multiple ways to create, manage, and delete policies. Vault operators can use the UI, CLI, or API to work with Vault policies. Operations for policies include actions such as *list*, *write*, and *delete*. Updates to existing policies follow the same steps as writing a

new policy, except the write should signify an existing policy name. Policy updates take effect immediately and may negatively affect the capabilities of an existing token. Therefore, care should be taken when implementing policy changes.

### *List Vault Policies*

To list all the policies available in Vault using the CLI, use the command found in Figure 4-91. Remember that *default* and *root* are always included since they cannot be deleted.

```
$ vault policy list
default
root
vault_admin
vault_operator
```

Figure 4-91

### *Write a Vault Policy*

Writing policies in Vault requires both the name of the policy and the policy file itself, often referred to as the *policy document*. The policy document can be provided to Vault in several ways when working on the command line. The policy can be written to a local file and referenced, or the policy can be provided in the heredoc format in the command. The heredoc format is typically only used when the policy is relatively short, although it is not required. More complex policies are often written using a source code editor such as Visual Studio Code or Atom. They are commonly stored and versioned using a code repository solution such as GitHub or GitLab.

The command displayed in Figure 4-92 shows how to create a new Vault policy using a reference to a local file that contains the policy. The user needs *create* permissions to write a new policy and needs *update* permissions to update an existing policy.

```
$ vault policy write vault_user /opt/vault/user.hcl
Success! Uploaded policy: vault_user
```

Figure 4-92

To create a policy using the heredoc format, use the command found in Figure 4-93.

```
$ vault policy write vault_user -<<EOF
> #Allow Vault user to read anything in KV store
> path "secret/*" {
>   capabilities = ["read"]
> }
> EOF
Success! Uploaded policy: vault_user
```

Figure 4-93

### *Read a Vault Policy*

After a policy has been written, it might be helpful to read back the policy to troubleshoot or determine modifications to a policy. While the UI is probably the easiest to use for this task, it can be done using the CLI and API. Figure 4-94 shows how to read a policy using the CLI.

```
$ vault policy read vault_user
#Allow Vault user to read anything in KV store
path "secret/*" {
  capabilities = ["read"]
}
```

Figure 4-94

### *Delete a Vault Policy*

A policy can quickly be deleted when it is no longer needed. Note that Vault allows a delete operation regardless of whether the policy is assigned to a token. Deleting a policy affects any existing tokens associated with the policy. The command in Figure 4-95 displays how to delete a policy using the CLI.

```
$ vault policy delete vault_user
Success! Deleted policy: vault_user
```

Figure 4-95

## *Audit Devices*

Audit logs are vital in any security product because they allow administrators to ensure proper usage of the product and investigate further in case of an incident. Vault uses audit devices to create audit logs of every authenticated *request* to Vault and the corresponding *response*. The resulting logs can be easily examined by security or operations teams.

Audit logs are a great place to start when examining unauthorized access to Vault or merely troubleshooting access problems with a Vault client. Audit logs contain data such as the client IP address, the time of the request, the requested action, and the resulting data from Vault. Before data is written to the log file, sensitive information is hashed with a salt to prevent secrets or tokens from being stored in a plain-text format. Proper care of the logs is still essential despite the hashing of secrets. If logs are stored locally, the log file should be encrypted, and care should be taken to prevent unauthorized access to the encrypted logs.

The data generated by Vault audit logs are one of the most powerful tools available to teams using the service. These logs can be funneled into a tool such as Splunk to aggregate and create custom searches. The consumer can use the log data to troubleshoot issues with their application, by the administrator who needs to understand the health of the service, or by the compliance auditor ensuring secrets are being accessed and used securely.

Additionally, features from log aggregation tools can be used to generate alerts for known issues, trigger specific actions that can automatically resolve individual problems, or allow a security operations center (SOC) to correlate the data with other events in a security information and event management (SIEM) service.

As an example of an audit device log entry, Figure 4-96 displays a partial response log generated by a request to retrieve a secret from a K/V store. For this example, the audit device was configured not to hash sensitive information using the *log_raw=false* parameter. This setting should always be set to true for production environments.

```
"request": {
  "id": "7785e1aa-2000-3d16-a80f-4239ba71b446",
  "operation": "read",
  "mount_type": "kv",
  "client_token": "s.jOPCHzvqohzDDqtZlQ6u4un3",
  "client_token_accessor": "8u4fmsGHCpCsKKo3A2pOYvpV",
  "namespace": {
    "id": "root"
  },
  "path": "secret/data/book",
  "remote_address": "10.0.0.142"
},
"response": {
  "mount_type": "kv",
  "data": {
    "data": {
      "vault": "production"
    },
    "metadata": {
      "created_time": "2020-04-22T18:50:20.2643256Z",
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  }
}
```


Figure 4-96

## Types of Audit Logs

Vault has three types of audit devices that customers can take advantage of, including the *file*, *syslog*, and *socket* audit devices. Each audit device has strengths and weaknesses, although the *file* audit device tends to be the most widely used. Like other Vault configurations, multiple audit devices of the same type can be enabled if needed. Audit device configuration can be combined with third-party tools to provide an endless number of solutions to satisfy an organization's needs.

### *File Audit Device*

The file audit device enables Vault to write the audit log directly to a designated file. New logs are appended to the log file, and it is up to the Vault operator to manage log rotation. Third-party tools such as logrotate, FluentD, or DogStatsD can assist with managing rotation during log collection. Log collection tools also help prevent the log file from consuming all available free disk space on the underlying volume.

### *Syslog Audit Device*

The Syslog audit device allows Vault to send audit logs to a local Syslog agent, such as syslogd or rsyslog. Unfortunately, the Syslog audit device currently does not support sending logs directly to a network-attached Syslog server. The Syslog audit device is frequently configured to write to a local file for either log collection or used merely as a second option for audit device configuration.

### *Socket Audit Device*

The socket audit device configures Vault to write the audit log to a specific IP address and port based on the selected protocol. The underlying protocol can be configured for *TCP*, *UDP*, and *UNIX*. Although socket does support TCP, most organizations tend to shy away from using the socket audit device due to the underlying protocols, which are inherently unreliable. Regardless, the socket audit device could prove to be useful to some organizations.

## Using Multiple Audit Devices

If an audit device is enabled, and Vault cannot successfully write to a log, Vault ceases to respond to client requests. This integrated failsafe ensures that all requests and responses are audited correctly. Therefore, it is critical to have multiple audit devices enabled in a production Vault deployment. Enabling multiple audit devices provides the business with multiple copies of the logs as well.

Audit devices must be enabled individually. Within a Vault environment, each of the multiple audit devices should be of different types or have different strategies to write the logs. An example may be writing one copy locally and sending a second copy to a Syslog server. A

negative impact on one of these audit devices should not impact the other from writing the log.

## Consuming Audit Logs

These audit logs are often ingested into an enterprise log aggregation tool such as Splunk or DataDog. Open-source alternatives include Grafana or Kibana. These tools can allow organizations to correlate and visualize the Vault audit logs to understand the environment's overall health. These tools also enable the organization to configure alerting on Vault events, such as changing a policy or the deletion of an encryption key. Once notified, security and operations teams can take appropriate action if necessary.

The log aggregation configuration usually pairs a *file* audit device with a log collector running on each Vault node. The log collector sends the audit logs to a collection server based on the configured interval while managing the log rotation and cleanup.

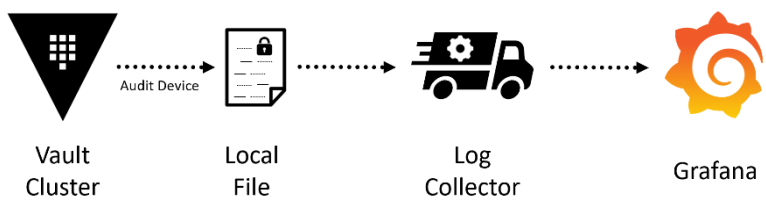



Figure 4-97

Additionally, log aggregation tools can be paired with automation tools such as Phantom or Rundeck to orchestrate specific actions based on certain types of activity. For instance, a user making unauthorized attempts to access Vault data may trigger an action from Phantom to gather and correlate data from other sources regarding other activities in which the user is currently engaged.

# Enabling an Audit Device

## *File Audit Device*

As with other Vault components, there are several options to enable an audit device, including the CLI and API. The UI does not support the configuration of an audit device. To enable a *file* audit device using the CLI, use the command found in Figure 4-98.

```
$ vault audit enable file file_path=/var/log/audit.log
Success! Enabled audit device (if it was enabled) at: file/
```

Figure 4-98

Note that the system user running the Vault service requires permissions to the directory where the log file is written. If the Vault service does not have sufficient privileges, Vault will return an error when enabling the audit device.

## *Syslog Audit Device*

The commands depicted in Figure 4-99 can be used to enable the Syslog audit device. The command enables the audit device to use the default facility of "*AUTH*" and a tag of "*vault*" for logs. These parameters can be set when enabling the Syslog audit device if the default value is not desired.

```
$ vault audit enable syslog
Success! Enabled audit device (if it was enabled) at:
syslog/
```

Figure 4-99

Remember that the Syslog daemon must be configured separately to obtain the Vault logs and forward them to the remote Syslog server. If logs are not populating in the remote Syslog server as expected, check for errors in Vault's logs or the Syslog daemon's log. Do not forget that any firewalls between Vault and the Syslog server must permit the action.

### *Socket Audit Device*

Vault also supports using a socket audit device, which can send Vault audit logs to a TCP, UDP, or UNIX socket. For organizations requiring strong consistency for audit logs, the socket audit device is not recommended since a log entry may be lost due to the underlying protocols used. This warning can be found directly in HashiCorp documentation (*https://www.vaultproject.io/docs/audit/socket*).

The commands found in Figure 4-100 can be used to enable the socket audit device. The configuration can be supplied using key/value pairs. This configuration will instruct Vault where to send the audit logs along with the underlying protocol to use.

```
$ vault audit enable socket address=127.0.0.1:5170 \
 socket_type=tcp
Success! Enabled audit device (if it was enabled) at:
socket/
```

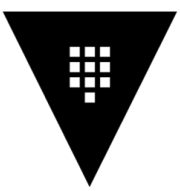
Figure 4-100

# 5 Managing the Vault Service

### *This Section Covers:*

- Vault Service Maintenance
- Scaling and Optimizing Vault
- Replication and Disaster Recovery
- Common Issues and Solutions
- Common Migration and Onboarding Patterns

Now that Vault has been deployed and core functionality has been configured, it is time to prepare for the human element of a Vault deployment. Getting the organization to use Vault correctly while ensuring Vault keeps working as expected is just as crucial as the rest of the planning that has gone into Vault adoption.



# *Vault Service Maintenance*

After the Vault service's successful launch, Vault will require regular maintenance to ensure it remains accessible to those consuming it. Not all Vault maintenance tasks involve Vault itself, though. Some of these tasks pertain to Vault's underlying infrastructure or the tools used to support Vault. This section includes details on the configuration of the Vault service itself, the infrastructure supporting Vault, and general suggestions to ensure Vault performs as expected from day one and beyond.

## *Security Patches*

Throughout the lifecycle of the Vault service, security patches will undoubtedly be released for the underlying operating systems running both Vault and Consul. The Vault operational team should work with the security team to determine the urgency of security patches and apply them accordingly. As always, patches should be tested in lower environments before being introduced in production.

It is worth reiterating that security patches should be addressed through an image-building pipeline rather than performing in-place upgrades. After the new updated image has been validated, the Vault nodes should be re-deployed. Treating the Vault environment as immutable helps ensure the availability, reliability, and consistency of the service.

## *Bug Fixes*

Like any other major software development project, Vault can contain software bugs that can impact the service's security or functionality. It may seem logical to install every bug fix that comes along right away, but this is not always the best idea. If the bug only has a minimal impact on performance, administrators may want to wait to ensure the bug fix is stable instead of just installing it to fix the bug right away.

Additionally, administrators may choose not to install some bug fixes at all. Bug fixes generally only need to be implemented when a bug that negatively affects features or functionality used by their organization. For example, if HashiCorp publishes a bug fix for the Transit secrets engine, but an organization does not use Transit, there is no reason for them to implement that fix.

Bug fixes should be addressed through an image-building pipeline, similar to how operating system patches are managed. An image containing the bug fix should be created, validated, and re-deployed. Validation and testing are critical to ensure the inclusion of a bug fix does not inadvertently cause another bug. Although HashiCorp does a great job of promptly releasing fixes as bugs are discovered, the validation pipelines discussed in the section on *Automating Vault Operations* provide an additional safety layer to ensure bugs do not make it to production.

### *Software Upgrades*

HashiCorp regularly adds new features and improvements to solve some of the most common and pressing issues in secrets management. While it may be enticing to adopt some of these new features as they are released, it is essential to exercise caution with software upgrades. Upgrades should be tested and validated adequately before deployment to production.

Minor releases, sometimes also referred to as *dot* releases, generally include small improvements or bug fixes for the associated major release. The decision of when to upgrade will depend on how valuable the new functionality is to the environment. At a minimum, organizations should perform quarterly upgrades to the minor releases associated with their current Vault version, with appropriate validation and testing.

Upgrades to major releases, on the other hand, should be much less frequent. Major releases should only be implemented if there is a compelling reason to do so. Examples of good reasons to do a major upgrade include a new feature that solves a business problem, a bug fix that is only available in the next release, or deprecation of the current release.

Like other Vault upgrades or changes, a validation pipeline is critical in ensuring that new builds are thoroughly tested before they enter production. More details for validation pipelines can be found in the chapter on *Automating Vault Operations*.

### *Adjusting Node Resources*

If a Vault cluster is not performing as required, it may be necessary to increase resources for the cluster nodes. Fortunately, Vault is very flexible with such adjustments, allowing these to be done "live." Start with the Vault nodes that are neither the active node nor the raft leader in the cluster. To view leadership status on a Vault cluster, use the command

found in Figure 5-1. The address of the leader node will be shown as the value of the *Active Node Address*.

```
$ vault status
Key                     Value
---                     -----
Seal Type               shamir
Initialized             true
Sealed                  false
Total Shares            1
Threshold               1
Version                 1.5.4
Cluster Name            vault-prod-cluster
Cluster ID              c03740ce-65a8-02023401ec528b88
HA Enabled              true
HA Cluster              https://vault.example.com:8201
HA Mode                 standby
Active Node Address     http://node-a.example.com:8200
Raft Committed Index    33
Raft Applied Index      33
```

Figure 5-1

The leader node of the raft cluster can be determined by using the command found in Figure 5-2. The leader will be indicated under the state column.

```
$ vault operator raft list-peers
Node      Address           State       Voter
----      -------           -----       -----
node-a    10.0.8.23:8201    leader      true
node-b    10.0.8.58:2201    follower    true
node-c    10.0.8.15:2201    follower    true
```

Figure 5-2

Once a node has been identified as a standby node, it is safe to stop the Vault service. Maintenance should only be performed on one Vault node at a time to ensure the service can continue to handle requests while during maintenance. After the Vault service is stopped, the host can be powered down. Once the host is down, the resources can be adjusted, and the host can be powered back up. As the Vault service starts, the node will join the cluster again. After each standby node successfully rejoins the cluster, the process can be executed on the active node.

Before upgrading the active node, the node will need to be forced to step down from active duty. To do so gracefully, use the command displayed in Figure 5-3. Once executed, a standby node should obtain a lock on the storage backend and be promoted to the active node. Verify that one of the standby nodes has successfully been promoted into an active node before proceeding with upgrades.

```
$ vault operator step-down
```

Figure 5-3

After all Vault nodes have been adjusted, the same process can be completed for a Consul backend, if applicable. While upgrades should be performed sequentially, be sure to maintain enough nodes to maintain a quorum. Falling below the threshold will negatively impact the Consul service and subsequently take down the Vault service. For example, a Consul cluster of five nodes requires a minimum of three nodes for a majority.

### *Adding Clusters*

Vault can meet organization-wide performance requirements by scaling out. This means that additional clusters should be provisioned when attempting to span Vault over multiple data centers or cloud regions. Requirements that justify the deployment of additional clusters might include a need to accommodate higher transactional throughput, segregation of environments, or to avoid limitations.

These secondary clusters should be provisioned using templates and configuration management tools. Using these tools can ensure that all Vault clusters experience smooth and efficient deployments while reducing human error. Once the new cluster(s) are available, secondary tokens must be generated and applied to each secondary cluster.

More information about Vault replication can be found in the section *Replication and Disaster Recovery*.

Keep in mind that if a new standalone cluster is provisioned, all the necessary Vault configurations and policies will need to be applied to it as well.

### *Unused Tokens*

When tokens are generated for Vault clients, it is common for them to have a longer TTL than required, even though it is an anti-pattern for tokens. Since active tokens have a footprint in memory or disk, it is sometimes necessary to revoke inactive tokens to improve Vault's performance. These revoked tokens can then be garbage collected so that Vault no longer has to manage them.

When removing tokens from Vault, users should proceed with caution: once a token is revoked, it cannot be recovered. Any entity using that token to retrieve secrets will immediately lose Vault access and need a new token to continue operations. Be sure that tokens are no longer needed before revoking them. Audit data from log aggregation tools can assist with determining which tokens to delete. If best practices are followed by using short TTLs, tokens should never need to be cleaned up.

Removing inactive tokens in Vault starts with determining the associated accessors. The accessor is a unique ID created for each token that can be used for limited actions, such as viewing token properties, renewing a token, or revoking a token. A list of accessors for active tokens can be retrieved from the */v1/auth/token/accessors* endpoint using the Vault API. Note that *sudo* permissions are required to request the list of assessors. For example, the command found in Figure 5-4 can be used to issue the request using the API, parse the response to extract only the accessors, and save them to a local file.

```
$ curl \
    --header "X-Vault-Token:<token>" \
    --request LIST \
    http://127.0.0.1:8200/v1/auth/token/accessors \
    | jq -r ".data.keys" > assessors.txt
```

Figure 5-4

Once the list of accessors has been obtained, the token properties can be requested from Vault. This request will obtain the *creation_time* (in epoch time), *creation_ttl* (in seconds), and *expire_time* of the token. The API endpoint to request this information is */v1/auth/token/lookup-accessor*. An example of the request and subsequent response can be found in Figure 5-5. This command will request details for a specific accessor and output the *accessor*, the *creation_time*, and the *expire_time* to a CSV.

```
$ curl \
    --header "X-Vault-Token:<token>" \
    --request POST \
    --data '{"accessor": "<accessor_id>"}' \
    http://127.0.0.1:8200/v1/auth/token/lookup-accessor \
    | jq -r '.data | [.accessor, .creation_time,
       .expire_time] | @csv' > accessor.csv
```

Figure 5-5

For bulk requests, the accessors can be obtained by the first command (Figure 5-4), and the output can be formatted to create a JSON file to be used as the payload for the second command (Figure 5-5). The resulting CSV file would provide a list of every accessor in Vault, the creation time, and the expiration time.

## *Unused Entities*

Entities are a central component of Vault. An entity is generated when a user (including applications, services, and nodes) other than *root* authenticates with Vault for the first time. All service tokens generated by Vault are associated with the entity of the user that requested them. While tools are being added to improve accounting and cleanup for entities, it is helpful to understand how to clean these up manually. This cleanup process will help ensure there is an adequate representation of the current Vault usage.

Entity information can be accessed at the */v1/identity/entity/id* endpoint in the API. Using the *--request LIST* flag provides details on existing entities as found in Figure 5-6. This request is an authenticated action and returns a JSON object that contains information for each entity that has authenticated with Vault. Data includes associated auth method, usernames, and entity IDs but *does not* include information on the entity creation date or the last time the entity was used.

```
$ curl \
    --header "X-Vault-Token:<token>" \
    --request LIST \
    http://127.0.0.1:8200/v1/identity/entity/id
```

Figure 5-6

Knowing how to find entity information is useful for automating cleanup of user data after de-provisioning of access. For instance, when a user leaves a company and their access to systems is revoked, a simple script could locate all associated entity information in Vault and delete those entries.

Proceed with caution when cleaning up entities**.** Deleting an entity still in use will not prevent users from authenticating with Vault after the cleanup because Vault will generate a new entity. However, it will remove access to all tokens associated with the deleted entity.

### *Unused Mounts/Namespaces*

As with any service, some users may stop using Vault at some point. The larger the organization, the greater the chances that those users will not notify the Vault support team that they no longer need their resources in Vault.

Unfortunately, there is no easy way to determine whether a namespace or mount has gone unused without some form of log aggregation tool or audit log tracking. Organizations can, however, automatically track usage of mounts and namespaces based on messages in the audit logs. Tracking this information allows an organization to take a proactive approach to clean up unused resources in Vault. Tracking audit log usage prevents unnecessary scaling of the service and the potential leak of sensitive data due to negligence.

As with most deletion in Vault, mount or namespace deletion should be performed with caution. Once a mount or namespace has been deleted, access to the secrets in the mount or namespace is immediately lost, potentially causing service disruptions. The mount or namespace data cannot be recovered without restoring from a snapshot.

## Performing Maintenance

Maintenance is often seen as exhausting and laborious because it is usually performed manually. However, there a few ways organizations can simplify the maintenance process and reduce efforts across the Vault support team. Adopting these approaches can help simplify basic maintenance tasks.

### *Live Maintenance*

Although maintenance is ordinarily performed after hours or on weekends, Vault's architecture allows much of the work to be performed live without negatively impacting the service. Version upgrades, software patches, or resource modifications are just some of the changes that could be completed during regular business hours. In the event of an issue, other employees and teams are also readily available during business hours, where they might not be after hours.

### *Validation Pipelines*

Validation pipelines can be a Vault operator's best friend during a maintenance window. Ensuring validation pipelines are in place can significantly reduce the amount of time invested in this monotonous task. These pipelines are discussed in greater detail in the section *Automating Vault Operations*. However, they are worth mentioning due to their criticality in ensuring the efficacy and consistency of Vault maintenance.

### *Log Aggregation Tools*

Tracking and cleaning up unused Vault resources can be done by employing a log aggregation tool such as Splunk. This tool could automate reporting regarding what should be cleaned up and when it needs to be completed. For instance, Splunk offers a feature called "summary index" to condense search data down to a smaller, more relevant subset of data and store that in a separate index. This feature can retain certain pieces of data for a longer period with a significantly reduced footprint. A scheduled search could be run daily on the audit logs to determine the last time a namespace or mount was used. The results of the scheduled search could be stored or updated in a summary index.

In addition, another daily search could be run against the summary index to determine which namespaces or mounts have not been accessed in more than 90 days. The result of

this second search could then be used to clean up unused namespaces or mounts. A similar method can be used to track entities or tokens created and used only once.

### *Blue-Green Deployment*

The concept of blue-green deployment allows for a transition to a newer version of an application or service without sacrificing availability. In blue-green deployments, the current version of the service (blue) continues to be available while the new (green) version of the service is introduced.

When Vault is deployed on cloud infrastructure, a new cluster can be provisioned alongside the existing cluster to stage service changes and upgrades. As the new environment approaches readiness and final testing is in progress, connectivity for the Vault service can be migrated to the new cluster. This change may be done on a load balancer or through DNS changes. If problems are discovered during the migration, the service can be easily migrated back to the previous working cluster. If the new environment works as expected, the original cluster can be decommissioned once the updates are deemed stable.

# *Scaling and Optimizing Vault*

As with any critical service, the need to move in step with business demands is vital. With Vault, scaling the service to meet organizational demands is more straightforward than it seems. Knowledge of a few simple rules combined with a strong understanding of Vault's limitations will help operators ensure their Vault deployment is as efficient as possible.

## Principles of Scaling

Throughout the various phases of the Vault service lifecycle, many common issues support teams face can be mitigated or avoided by following these principles regarding scaling.

**What works with ten should work with ten thousand**
Consider a process that only needs a token for a single, five-minute operation, but the TTL is set for 60 days. Vault needs to store and track this token for the full 60 days, even though it is not used. Now consider the same process where the service token is configured to expire after five minutes. Whether running ten or ten thousand of these processes, the leases in Vault are automatically cleaned up as they expire. Be sure to align time-based or use-based resources with the actual needs of the associated process and nothing more.

**Not all secrets engines are created equal**
Each secrets engine in Vault serves a different purpose and uses hardware resources differently. Transit, for example, processes cryptographic workloads and needs more CPU. On the other hand, the K/V secrets engine stores static values and tends to have a larger memory footprint. Design and scale with the most common use cases in mind.

**Vault is designed to scale horizontally**
Since Vault only supports a single active node within a cluster, individual Vault nodes are scaled up for performance (i.e., adding CPU or memory). While a single cluster can process thousands of transactions per second, there are some limitations when it comes to mounts, namespaces, and auth methods. Capacity should be expanded through additional Vault clusters rather than adding Consul or Vault nodes to individual clusters.

**Use the cloud**
In a physical data center, procuring hardware can be a tedious and time-consuming process. Using a cloud provider instead enables flexibility to scale vertically and horizontally without fear of exhausting physical resources. If the support team is waiting for hardware to

arrive because of limited capacity, downtime or outages can impact the Vault service as requirements often change quickly and with little notice.

**Slow your roll**
It is always exciting to see all the new features that the Vault Engineering and Product Management teams release. However, as with any new feature in applications, new features may bring new problems. Ensuring stability is a large part of ensuring scalability. Defer the deployment of those new features until those features have been appropriately vetted and potential bug fixes have been implemented. Make sure to test new features thoroughly before putting them into a production environment.

**Automation is key**
Automation is not just for Vault consumers, but the Vault support team as well. As the Vault service grows, automating the day-to-day operational tasks for Vault will reduce the time it takes to onboard teams. As more support work is automated, the service will get easier to manage. This logic is true regardless of the number of users that Vault is supporting

## How to Scale the Vault Service

Scaling the Vault service involves more than just expanding hardware resources. There are multiple elements at play that ensure a Vault deployment can scale without friction. The following sections cover several aspects of the Vault service and how scaling pertains to them in particular.

### *Building the Base Clusters*

The Vault service is comprised of the Vault application itself and the storage backend. Since these two elements are responsible for handling different workloads, they require different types of resources. The Vault service itself is intended to be the front-end and secrets management portion of the stack. Hosts that are members of a Vault cluster heavily rely on processing and memory resources.

In contrast, Consul is intended to provide the key/value store for storing Vault configuration and data. Hosts that are members of a Consul cluster lean heavily on memory and disk. This distinction is essential to understand since the underlying resources provisioned for Vault will not necessarily match the resources required for Consul.

## *Secrets Engines*

When the Vault infrastructure is being designed, it is essential to design the service and hardware requirements based on the type of secrets engine used to process the majority of the transactions. As the Vault service grows, consider separating clusters into the type of work they are performing. Creating a separate cluster dedicated to the Transit secrets engine, for example, could be an option since Transit is a CPU-heavy operation.

## *Consul Limitations*

Consul is a critical component of many large Vault implementations. The following guidelines should be followed to ensure Consul-based Vault implementations are reliable and scalable.

**Number of Nodes**
It is not recommended to expand a Consul cluster to larger than seven nodes. The performance of the Raft protocol begins to degrade at this point due to the additional replication traffic. If the Consul cluster reaches this size, consider options for separating workloads across other Vault and Consul clusters.

**Writing Data**
During regular operations, there are a few actions that should be happening regularly with Consul data. The data should be frequently written to disk, and periodic snapshots should be taken to avoid significant data loss. Depending on the size of the cluster, these operations can be very intensive for disk resources. While a small, underutilized Consul cluster may get by with magnetic hard disks, the use of SSDs is highly recommended for all Consul clusters.

**Mounts, Namespaces, and Auth Methods**
Consul imposes a few limits that should be considered in the Vault service's design, implementation, and ongoing management. These limitations include the maximum KV value size, the number of mount points, and several others. These limitations are outlined in greater detail in the Vault documentation (*https://www.vaultproject.io/docs/internals/limits).*

## *Segregating Data*

When considering how to divide workloads in Vault, the decision usually comes down to applications versus teams and namespaces versus mounts. Either of these considerations can determine the significance of an operational or security incident. The more granular the segregation of secrets, the smaller the blast radius for incidents.

**Applications vs. Teams**
When considering whether to segregate secrets by applications or teams, it is essential first to consider the limitations of the storage backend. For example, consider a support team that has determined that a new mount or namespace should be created for each application. If thousands of applications need Vault access, the support plan will need to consider horizontal scaling of the service.

**Namespaces vs. Mounts**
Namespaces enable the Vault operations team to provide "Vault as a Service" to the organization. They are essentially a Vault within a Vault. Namespaces are often used to allow the individualized configuration of preferred auth methods, isolate privileged credentials for dynamic secrets, and increase audit visibility. Namespaces are the recommended method of segregation for multi-tenant deployments. However, Consul's limitations allow for fewer namespaces than mounts, requiring a greater need to scale horizontally if namespaces are used.

It is essential to consider the first rule of scale: will this method work with our current workload and continue working with the load we will potentially reach?

## *Scaling Individual Clusters*

For most deployments, the ideal reference architecture would consist of three Vault nodes and five Consul nodes with the recommended amount of resources. However, there may be circumstances where a cluster needs to scale up (vertically) beyond the typical architecture. Scaling up is generally known as either adding hardware resources.

For example, if a Vault host with eight cores and 32GB of RAM has a utilization rate of 70%, it is far more cost-effective to increase the CPU cores and RAM of the instance than it is to deploy a secondary Vault cluster. When scaling the Vault instances, it is essential to keep in mind the limitations of Vault as well. As use cases continue to be added to Vault, it may be

better to migrate workloads to a secondary cluster to avoid Vault's upper limitations, such as the number of mount points and namespace limits.

## *Scaling Out with Clusters*

Provisioning additional cluster(s) is the recommended method for scaling out the Vault service, especially in environments with multiple data centers. Several scenarios may require the addition of a new cluster.

- Deploying a cluster closer to an application to improve the latency of secrets retrieval
- Compliance regulations that require physically separate clusters for certain types of transactions and data
- Differing security requirements in certain "compromised" or "high-risk" zones
- Additional capacity needs due to Consul limitations
- Overlapping network segments where infrastructure and applications do not have appropriate network paths available (physical separation)

Consider the following scenarios where the addition of secondary clusters may be necessary:

**Scenario 1**
A company has decided to use Vault Namespaces as the method to onboard and segregate applications. Each application will have a namespace to limit the scope of security and operational incidents. However, the company has tens of thousands of applications along with thousands of users, all of whom will be interacting with Vault. Such a substantial workload would prove to be an uphill battle with only a single cluster.

There are a few options to solve this issue. The first is to consider segregating by teams rather than applications. This segregation would increase the blast radius for inadvertent or breaking changes but still fit within the confines of Consul limitations. The second option is to separate similar applications on different Vault clusters or replication clusters sets. For instance, there may be a group of microservices that make up a single application. Several of those microservices may also be used across multiple applications, such as a payment processing service. It may make sense to group those shared microservices on a single

Vault cluster that is accessible by all applications but group the other application-specific microservices on a separate cluster.

**Scenario 2**
A company has a single Vault cluster deployed and is using the K/V secrets engine and the Transit secrets engine. It is getting costly for them to scale their instances to provide enough CPU and memory to support both secrets engines as their Vault use increases.

The support team determines that it is best to separate the Transit and K/V workloads to separate instances that better accommodate the required CPU and RAM requirements, respectively. This approach can cut costs by improving efficiency for both use cases.

**Scenario 3**
Certain geographical regions have security or regulatory compliance requirements where it makes sense to isolate data. For example, China does not permit connectivity to other regions for any reason.

If an organization was running a Vault cluster in Singapore and applications in China, the applications would not be able to connect to that Vault cluster in Singapore. It would be necessary to provision a new Vault cluster in China to provide Vault services for the applications running in China.

## *Automated Processes*

Like it or not, we live in a DevOps world. Luckily, Vault is built for the challenges that world brings. While Vault can accommodate current or legacy use cases, it outshines the competition when consumed by automated workflows.

As an example, consider the following common scenario. For a business unit to onboard to Vault, the team must submit a service ticket. The team must wait for that ticket to reach the front of a queue and wait for a Vault operator to create a namespace and associated security policies. Such a process can take anywhere from a couple of days to a few weeks, depending on the size of the organization and current demand. Much of the delay, however, is generally attributed to the human factor.

However, what if an automation tool (e.g., Jenkins, CircleCI) was used to validate the items that Vault operators would typically validate during a request? Employing such a tool could

now automate many of these processes that are typically stalled by humans. Not only is this faster, but it is far more accurate and consistent. Automated processes, in most cases, can reduce traditional wait times to days or even hours. The new process can also free up the support team to work on more compelling items.

### *Training*

Although this does not involve specific changes that can be made to the Vault service stack, few things are more critical to scalability than user training. If done correctly, a good training program can build community, drive adoption, and reduce ongoing support overhead.

Key elements of a user training program include:

- A regular training schedule, such as every six months
- In-person classes for those in need of a focused learning environment
- Hands-on lab resources to apply skills during the training
- Informative recordings to post to a learning management system (LMS), allowing internal users to consume the information at their own pace

For organizations looking for training, HashiCorp will partner with folks looking to implement internal training classes. They can also provide the necessary materials for achieving success. Other options include purchasing training from HashiCorp, which will be delivered by HashiCorp's training partner. Community-based content and resources are also available.

### *Community*

Building an internal community around Vault is valuable because it essentially crowdsources training and support. A well-functioning community is marked by an environment where users of the service are answering technical questions on behalf of the support team, significantly reducing support overhead.

As the Vault service expands, so will the demand for end-user support. The use of a tool such as Slack provides somewhat of a live version of StackOverflow, where users can find live help and search through a long history of frequently asked questions.

It may also help to conduct regular meetings and technical demonstrations where users can showcase their Vault integrations for other users. These demonstrations help to further knowledge, innovation, and adoption of the Vault service. As with training, the value of building an internal community cannot be overstated.

## Optimizing the Vault Service

To improve support for Vault deployments, HashiCorp has introduced documentation on optimizing the various components of the Vault stack. These Learn guides contain lots of great information for improving the performance of all layers of the Vault service, including the operating system, the Consul storage backend, and the Vault service itself.
This information can be accessed on the HashiCorp Learn platform (*https://learn.hashicorp.com/tutorials/vault/performance-tuning).*

## *Replication and Disaster Recovery*

As organizations grow more reliant on Vault, performance and recovery become critical to business operations. Applications are often deployed in active/active or active/passive configuration to ensure that they stay available to their customers. Vault's underlying infrastructure also needs to be extended to all data centers or cloud regions where these applications are actively running so that Vault is accessible to them locally. This strategy can be used to deploy applications closer to customers, or it can be used for disaster recovery purposes.

Extending Vault to additional clusters via replication is a Vault Enterprise feature that requires Enterprise licensing from HashiCorp. There are two distinct options for Vault replication: performance replication and disaster recovery replication. Each of these two options is uniquely licensed, as each option targets a unique use case. It is not uncommon for larger organizations to run multiple performance and disaster recovery clusters throughout their infrastructure.

The selection of the appropriate Vault replication option hinges on whether this new cluster will need to service client requests or not. If the cluster needs to provide Vault services to local applications and users, performance replication is required. If the replicated cluster is deployed solely for disaster recovery purposes, disaster recovery replication can be used. The selection between the two options should be determined in the initial Vault architecture design. Application architecture, strategy, and placement should be the primary factors in selecting how many clusters to deploy and where they should be deployed.

Vault replication follows a leader/follower model, where there is always a single *primary cluster* and one or more *secondary clusters* to which the primary cluster replicates data. Data replicated from a primary cluster includes the Vault configuration, auth methods, secrets engines, audit devices, and any static data such as the K/V backend. Any writes sent to the primary cluster are replicated to the secondary clusters. If a secondary cluster receives a write request, it automatically forwards the request to the primary cluster to handle the operation.

The other key distinction between the two Vault replication options revolves around replicating tokens and associated leases. Service tokens generated on the primary cluster are replicated only to disaster recovery clusters. If a disaster recovery cluster is promoted to a primary, both applications and end-users can utilize the newly promoted cluster without

re-authenticating. This strategy simplifies recovery in the event the primary Vault cluster fails. Since tokens are not replicated to a performance replica cluster, clients accessing a performance replica cluster are expected to authenticate with the performance replicated cluster as well. This cluster generates a token to be used for subsequent requests.

## Performance Replication

Organizations looking to extend the Vault service to other data centers, cloud providers, or regions should use performance replication. Deploying performance replicated clusters not only extends the Vault service, but it also reduces the workload on the primary Vault cluster. Performance replicated clusters can service read requests from clients. The premise is that local clients use the local Vault cluster for accessing secrets or other services, which reduces both latency and network traffic for the request. Using performance replicated clusters, organizations with significant write requests can scale out to handle the additional requests.

A primary Vault cluster can replicate to many performance replicated clusters simultaneously. This scenario is common in larger Vault deployments. Compared to the primary cluster, performance replicated clusters are often found in secondary data centers, in another cloud provider, or another cloud region where applications are deployed. Figure 5-7 shows a Vault architecture that includes a primary in AWS and multiple secondary clusters in Azure and an on-premises data center.

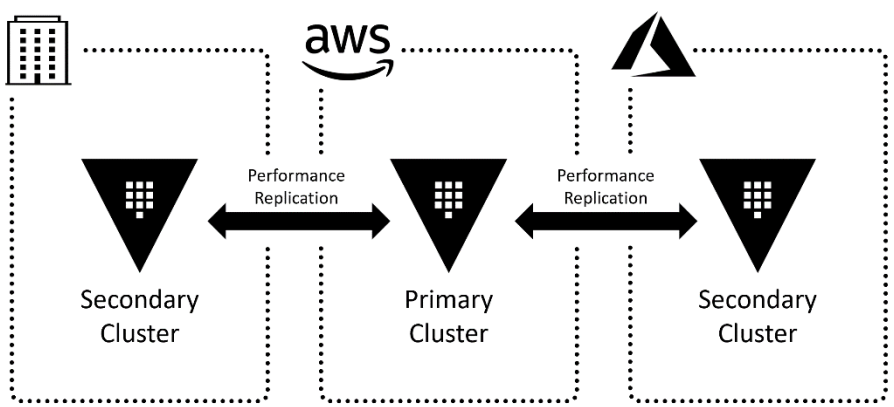



Figure 5-7

## Disaster Recovery Replication

The second type of replication supported by Vault Enterprise is disaster recovery replication. Disaster recovery replication's purpose is self-evident; disaster recovery replication is primarily for disaster recovery scenarios. During normal operations, a disaster recovery cluster does not respond to read or write requests from clients or forward the requests to the primary cluster. Its sole purpose is to mirror the primary cluster by receiving data, configuration, and tokens from the primary cluster. If the primary cluster becomes unavailable, a disaster recovery cluster can be promoted to a primary cluster to begin servicing clients.

Disaster recovery clusters are frequently found in data centers or cloud regions alongside their primary cluster counterparts to support local recovery efforts. These clusters are also commonly deployed in secondary data centers or cloud regions to support complete data center or cloud region failures. This strategy also assumes that applications are failed over to the secondary location as well. Many businesses that run active/passive architectures may opt to run a disaster recovery cluster in the second location rather than run a performance replication cluster. Due to the current licensing model, using disaster recovery clusters tends to be more cost-effective.

Figure 5-8 displays a primary cluster with a local disaster recovery cluster and a disaster recovery cluster configured in a second AWS region.

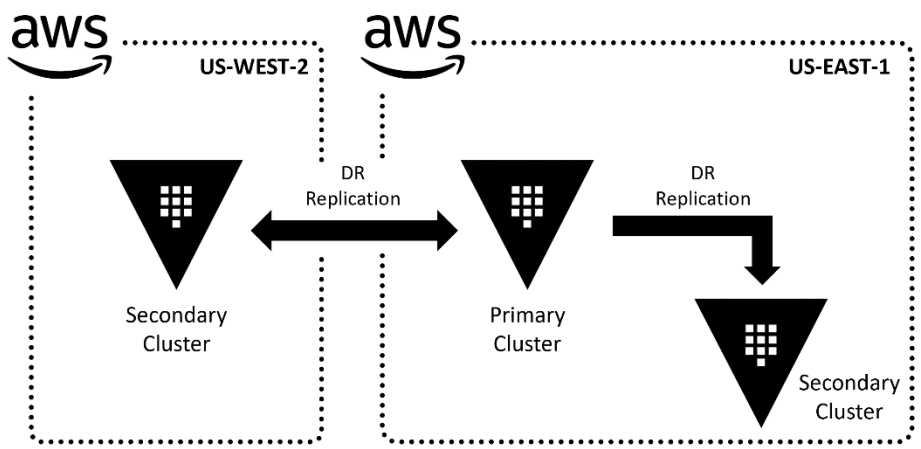



Figure 5-8

## Replication Configuration

Configuring Vault replication is a multi-step process that involves the configuration of both the primary cluster and the secondary cluster. Once replication is enabled on the primary cluster, a *secondary token* must be generated. This secondary token is a wrapped token that contains critical information to establish communication between the primary cluster and secondary cluster. Once the secondary token is generated, it is provided to the secondary cluster to initiate Vault replication. Each secondary token can only be used one time.

When Vault replication is enabled on the primary cluster, an internal certificate authority is enabled. A self-signed root certificate is generated, and a client certificate is created for the secondary cluster. These mutual TLS certificates allow Vault to secure and validate communications between the clusters. The root and client certificate are both passed to the secondary cluster using the secondary token. Remember that these certificates used for replication are separate from the TLS certificates provided to Vault to secure the Vault UI and API interfaces.

Once the token is provided to the secondary cluster, any configuration or data on the secondary cluster is permanently removed and replaced with data and configuration from the primary cluster. This data includes the configuration and any configured secrets engines, auth methods, audit devices, or Vault data. It also replaces the local unseal or recovery keys with the keys replicated from the primary cluster. The initial replication process happens relatively fast, usually in just a few seconds.

After replication has been completed, the primary cluster continues sending updates to the secondary using write-ahead logs (WALs). This log shipping method allows the primary to package up changes and quickly ship them to the secondary cluster. When there are no changes to the underlying configuration or storage, the primary cluster will not send logs to the secondary cluster.

## Promoting a Secondary Cluster

If a primary cluster becomes unavailable, a secondary cluster can be promoted to a primary cluster. This process allows the new primary cluster to handle client requests as part of a business continuity plan or workflow. It is important to note that Vault does not automatically promote a secondary cluster if a primary cluster fails. It is up to the Vault operator, or

automated process, to promote the secondary cluster and make all the necessary infrastructure updates to enable communication to the new primary. This process helps prevent a split-brain scenario where there might be multiple primary clusters within the environment. Additional changes to consider during the promotion of a secondary include DNS record changes, firewall updates, and modification of other Vault clusters if they are expecting write-ahead logs from the original primary cluster.

## Path Filters

While designing and configuring Vault replication, organizations may decide that specific data should not be replicated to a secondary cluster for various reasons. These reasons may be to follow regulatory/compliance standards that the business may be required to follow, such as the General Data Protection Regulation (GDPR). Other reasons may include reducing network traffic or that the specific data may not be needed in the secondary location. Whatever the reason, Vault path filters enable Vault operators to choose which paths to replicate to secondary clusters.

Using path filters in both the root or child namespaces, the Vault operator can fine-tune replication using an allow or deny list. An allow list should be used when Vault operators need to replicate only specific paths. Any paths that are not included in the allow list are not replicated. In contrast, a deny list should be added when most of the paths should be replicated with a few exceptions. When using a deny list, all paths are replicated except the paths included in the deny list.

As an example, Figure 5-9 displays a Vault deployment with multiple performance replication clusters. The primary cluster is replicating the *cloud/* and *apps/* paths to the US-East data center and the *apps/* path to the US-West data center. It does not replicate the *eu/* path to either data center since it contains sensitive data that cannot be replicated outside of EMEA.

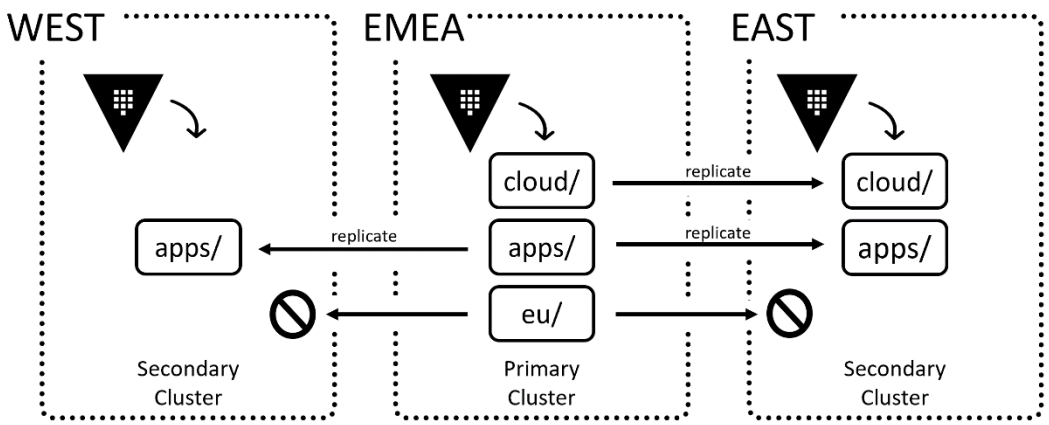



Figure 5-9

## Local Mounts

When using replication, any mount that is created is automatically flagged for replication. However, using the *-local* flag when the mount is created forces the mount to exist only on the local cluster. Local mounts are also a handy feature when data isolation is required.

When using local mounts, there are a few things to keep in mind. Once a mount is created as a local mount, there is no way to reconfigure for replication. If replication is eventually needed, the mount will need to be destroyed and recreated. Since the mount only exists on the local cluster, there is an even greater requirement to ensure the data is protected by using a disaster recovery cluster or by frequent snapshots of the storage backend.

## Disaster Recovery

Even with all the strategies and options available to improve reliability, it is inevitable that unpredictable events will occur with the Vault service. Such events may require simple fixes, such as restarting the service or removing a node from service. However, more severe events may require the support team to failover to another cluster or restore data. Fortunately, there are a few options available that the support team can use to ensure the continuity of the Vault service. The option to use depends on the type of failure that Vault has experienced and its impact on the Vault clients.

### *Failover Using a Performance Replication Cluster*

The simplest form of failure happens when a primary Vault cluster is no longer responding to requests. Replication to a secondary cluster enables multiple benefits, as discussed above. Clients local to the secondary cluster can interact with it while the secondary participates in the replication set. A replication set is defined as “a group of performance replicated clusters that replicate configuration and secret data from a single, shared primary cluster.”

Even when the primary cluster fails, access to secrets can still be gained through another cluster in the replication set. Thus, the use of service discovery through Consul or failover logic in the application can allow the application to use another cluster in the replication set. Depending on how these failover mechanisms are designed, the failover to a secondary cluster can occur in near real-time. However, this scenario requires users and applications to re-authenticate as service tokens are not copied to performance replicated clusters.

When the primary becomes unavailable, not all types of requests will work on a performance standby cluster. If the requests *read* operations, functionality will remain mostly unaffected. However, *write* operations are subject to failure in this scenario since performance standby clusters will forward write requests to the primary. The performance standby must be promoted to a primary cluster to reestablish write functionality (Figure 5-10).

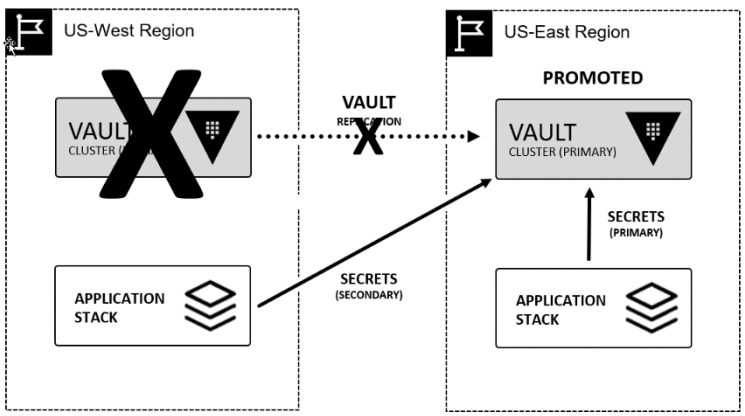



Figure 5-10

Promoting a performance cluster to a primary is a relatively simple task. A Vault operator with sufficient privileges can execute a single command (Figure 5-11) to promote the cluster from a performance replicated cluster to a primary. Such a change should only be performed if the primary has failed and the Vault service needs to be recovered immediately. There should never be two primary clusters within a single replication set, as such a scenario can lead to data loss or corruption.

```
$ vault write sys/replication/dr/secondary/promote
```

Figure 5-11

### *Failover Using a DR Replication Cluster*

Another scenario for the Vault infrastructure may come in the form of a primary or performance replicated cluster failure. As discussed above, a DR replicated cluster can provide warm-standby capabilities with slightly different functionality than performance replication. Unlike performance replication, this replicated data includes tokens and associated leases. This type of replication allows for full failover without the need for users or applications to re-authenticate to continue using Vault in a failover event.
In situations where a primary or performance replicated cluster has failed, disaster recovery clusters can help recover the service.

With almost any failover operation, there will be an interruption to the service as the DR cluster is promoted, and clients are redirected to the new cluster. The service interruption can be as low as 60 seconds, depending on whether the failover is automated or not. The duration of the service disruption in relation to stated SLAs should always be considered when using this option.

Disaster recovery clusters are often placed in separate regions to ensure service continuity in a worst-case scenario. However, they can also be placed in the same region to provide localized redundancy for a Vault environment. The decision on placement should be decided upon during the design phase of new Vault implementation projects.

Like performance replication clusters, the process for failing to a disaster recovery cluster is designed to be manual to ensure the operator is making a conscious decision to failover the Vault service. It is possible to automate the failover process; however, the automation

should be triggered manually. Automated failovers can sometimes result in a split-brain scenario, which can potentially cause data corruption or loss.

There are two scenarios to consider when failing over to a DR cluster: a planned failover or an unplanned failover. A Vault operator must follow the DR operation token generation process during a planned failover, which methodically promotes a cluster using the unseal keys. Ultimately, this workflow results in a root token that the Vault operator can use to promote the cluster to a primary. For an unplanned failover, the promotion can be accelerated by maintaining a batch token generated from the primary cluster. This process allows the Vault operator to skip the unseal key workflow and immediately promote the DR cluster using the batch token. Applying such a process enables the organization to promote the cluster without requiring the involvement of multiple key holders.

For detailed information on either scenario discussed, the workflow to promote a disaster recovery cluster to a primary can be found on the HashiCorp Learn site (*https://learn.hashicorp.com/tutorials/vault/disaster-recovery).*

## *Storage Snapshots*

Replication for both performance replicas and disaster recovery replicas occurs in near real-time. As modifications or updates are made on a primary cluster, changes are immediately replicated to all attached replicas. While this type of replication is ideal for normal operations, it will also replicate accidental or inadvertent changes.

Another failure scenario may be related to data loss, whether through natural disaster or loss of the storage backend. Ensuring the recovery of this data requires frequent snapshots of the storage backend. These snapshots should be the last line of defense in disaster scenarios. Snapshots should be scheduled to meet the recovery time objectives of the organization. Snapshots should also be copied to redundant, offsite storage for an added layer of protection, such as Amazon S3 or Azure Blog Storage.

By restoring a snapshot, organizations can recover entire Vault clusters. Despite this targeted use case, individual KV data could also be restored by restoring the snapshot to a temporary cluster. Although this is a much more tedious process, it could allow the recovery of targeted data within Vault.
To take a snapshot of the data on the Consul storage backend, use the command in Figure 5-12 on any Consul node in a Vault cluster:

```
$ consul snapshot save <file-name>
```

Figure 5-12

To take a snapshot of the data on the integrated storage backend, use the command shown in Figure 5-13 on any Vault node in a cluster:

```
$ vault operator raft snapshot save <file-name>
```

Figure 5-13

# *Common Issues and Solutions*

As with any application or shared service, Vault operations should expect to address operational issues. From experience, many of the common problems with Vault are a result of miscommunication, lack of process, or an absence of defined policies related to Vault operations. Although the specific experiences of every organization will vary, there are common issues that can be avoided with sufficient planning and understanding of the Vault service.

## Vault Seal

One of the most common issues is Vault entering a sealed state. While a Vault node is in a sealed state, it does not know the master key. The master key is used to decrypt the encryption key used to decrypt Vault data on the storage backend. Until Vault is unsealed, it will reject any client requests, which is likely not the desired state for a production deployment of Vault.

### *The Cause*

Most commonly, a Vault node is sealed due to the Vault service or underlying node being restarted. When Vault is configured to Shamir unseal keys, it requires manual intervention to unseal Vault. Until Vault operators provide these unseal keys, Vault will remain sealed. Vault operators can manually seal a Vault node as well.

### *The Solution*

Before Vault 1.0, open-source customers would need to manually enter the unseal keys or build automation that would automatically unseal Vault when Vault entered into a sealed state. Fortunately, HashiCorp open-sourced *auto unseal* in Vault 1.0, and now both Enterprise and open-source customers alike can take advantage of this feature. Auto-unseal enables Vault to use an external service, such as a cloud key management service, to decrypt the master key. Using an external service enables the Vault nodes to unseal automatically, rather than rely on unseal keys. It is highly recommended to use this feature as it can increase the availability of the Vault service.

## Loss of Keys

Regardless of what unseal mechanism is being used, organizations need to manage the unseal or recovery keys properly. Without the unseal keys, Vault cannot reconstruct the master key to decrypt the encryption key and access the Vault data. Since all data stored in Vault is encrypted with this key, losing the unseal keys could result in the loss of all data stored in Vault.

On the other hand, recovery keys are used for highly privileged operations, such as the generation of a new root token. These keys are also important to maintain since they can be used when all other authentication methods are not working. If the recovery keys are lost, and there are no other means for authentication, it may be difficult or impossible to recover the Vault service.

### *The Cause*

Loss of keys generally occurs when an operator initializes Vault and places the keys where they can be easily lost, such as a scratchpad.

### *The Solution*

Although Vault has a *rekey* option, it can only be used if the existing keys are available. If Vault is in an unsealed state, it is recommended to copy data to a secure location using a script or other automated method. For example, a new Vault cluster could be provisioned, initialized, and the data could be migrated to the new cluster. As an absolute last resort, it may be possible to extract the encryption key from memory and create new unseal keys, assuming root access to the Vault host.

If Vault is in a sealed state, the data is likely unrecoverable.

The unseal/recovery keys should be stored in a trusted location to avoid these scenarios. For example, the keys can be stored in a separate password vault or printed out and stored in a physical safe. Remember to store them in an encrypted format, if possible.

## Loss of the Storage Backend

Vault is wholly dependent on the storage backend to operate since its data and configuration are stored on the backend. Historically, Vault has leveraged HashiCorp Consul as the primary storage backend for enterprise-level deployments. If the Consul service becomes unavailable or Vault loses connectivity to Consul, the Vault service will cease to function.

### *The Cause*

Loss of the storage backend can occur through loss of connectivity to the Consul cluster, software or environmental problems, configuration changes, or operator mistakes.

### *The Solution*

Vault recently introduced a feature known as *integrated storage*, which provides similar storage functionality as Consul. Integrated storage uses the same raft protocol found in Consul to provide replicated storage, but it runs on the Vault nodes instead. Using Integrated Storage enables a supported backend without the requirement to operate an additional service like Consul.

Beyond migrating to integrated storage, the best solution to manage a storage backend loss is to take frequent snapshots of the Vault data and save them in a secure location. In the event of a complete loss of the storage backend, the data can be restored to a new backend using a recent snapshot. Each storage backend will have a unique strategy to snapshot the data. For example, the Consul snapshot agent can schedule snapshots and save them to cloud-based storage.

## Memory Exhaustion

If Vault or its configured storage backend does not have sufficient memory, the Vault service will be subject to out-of-memory errors. If this happens, Vault will refuse client requests and connectivity, preventing further authentication and retrieval of secrets. Insufficient memory may also result in a complete service failure. The Consul service is especially prone to out-of-memory conditions since it relies heavily on memory for data and replication operations.

### *The Cause*

The Vault service itself is not a heavy consumer of memory, but sufficient memory still needs to be allocated to the service. Often, memory issues result from under-provisioned hardware resources or poor service hygiene. Common offenders may include unused secrets or tokens with unnecessarily long TTLs.

### *The Solution*

Mutable infrastructure can play a vital role in maintaining proper resource allocations. The ability to quickly modify resources as the adoption of Vault increases is critical to ensuring the Vault service operates efficiently. Over-allocating resources initially will result in a more stable Vault deployment. Monitoring and trending can be used to right-size resources long term, which may be especially important in a public cloud environment.

Suggested sizing for Vault instances can be found in HashiCorp's documentation. Vault tends to be heavier on CPU than memory, but different use cases and architectures may affect the requirements. If Consul is being deployed as the storage backend, ensure the Consul hosts are allocated sufficient memory.

## Expired TLS Certificates

One of the most common interruptions for secure services results from expired TLS certificates. Most services require these certificates to secure communication between the service and the client. Certificates generated for these services should have a short expiration to ensure proper rotation and protection of the underlying service. For Vault and Consul, the use of TLS certificates should be non-negotiable.

### *The Cause*

The SSL certificates associated with Vault or Consul are not replaced before the expiration date. Expired certificates will disrupt connectivity to Vault and prevent further transactions. It may even appear as though connections to the Vault service are hanging.

### *The Solution*

Ensure an automated process is in place to automatically generate and rotate SSL certificates for Vault and the Consul storage backend. This certificate replacement will

require a sequential restart of both the Vault and Consul nodes, assuming Consul is used as the storage backend. Do not forget about any certificates that are used on load balancers, either.

## Audit Logs Fail to Write

If audit devices are enabled, Vault requires that at least one audit device can successfully record an audit log associated with a client request and response. If an audit device cannot write the log, Vault will cease to respond to client requests. This process is a security mechanism to prevent the use of Vault without attestation.

### *The Cause*

Different audit devices fail to write audit logs for different reasons. Vault currently only supports three options for audit devices, so the solution is easy to find.

**File**
A file audit device writes audit data directly to a local file. Reasons this audit device may fail include a missing directory, incorrect permissions, or lack of disk space.

**Syslog**
This audit device makes use of the *nix Syslog mechanism to write audit data. If this audit device is enabled, audit data is sent to a local Syslog agent. If the Syslog agent fails, so will the Syslog audit device.

**Socket**
The socket audit device writes to a predefined TCP, UDP, or Unix socket. Issues with this audit device are usually related to network connectivity.

### *The Solution*

Since the issues are directly related to which type of audit device is being used, the solutions will vary based on the type of audit device.

**File**
The system account running the Vault service must have access to write to the specified file. If the account does not have access, Vault will not permit the initial configuration of the

file audit device. Care should be taken to ensure this permission is not removed so it will not impact audit log operations. The audit log file will also need to be rotated frequently to ensure the log file does not fill up the underlying disk. Commonly, a data collection tool like Fluentd is used to forward logs to a log aggregation tool, which can manage the rotation of logs. If no such tool is in use, an open-source solution such as *logrotate* can assist with rotation.

**Syslog**
Since the local Syslog agent is primarily responsible for this audit device, resolving problems with this audit device will likely focus on the Syslog agent itself. Ensuring the Syslog agent is always running is should be the primary focus of a solution. Mechanisms can be put in place, such as *monit*, to monitor and restart the service if it stops. In addition to service management, Vault operators should be notified so they can investigate the issue. Notifications should be sent even if the problem was automatically resolved.

**Socket**
This audit device is prone to network-related issues. These connectivity problems can even occur when connecting to the localhost (*127.0.0.1*). The simplest way to troubleshoot this audit device is with standard networking tools such as *ping*, *MTR*, *netcat*, or *traceroute.* These tools allow Vault operators to identify where the network path is broken. This issue is often the result of a service dependency that is out of the control of the Vault service operator.

## Denial of Service

A denial of service (DoS) event can prevent the use of a shared service by flooding it with too many requests or network traffic. These denial of service events can be intentional, orchestrated by an attacker, or unintentional, such as an internal user inadvertently sending too many requests. Regardless of the cause, the result commonly prevents the legitimate use of that service. Like any other shared service, Vault is susceptible to denial of service attack as well.

### *The Cause*

Although many assume that DoS events are explicitly related to network resources, a DoS event can occur in many forms. There are two common offenders regarding Vault that will trigger a DoS event: an overabundance of network connections or hardware resource limits.

For example, when a user authenticates to Vault, a token is generated for retrieval of secrets. The token is stored until the max TTL is reached, and the token expires. Imagine a scenario where a user requested a new token every five minutes with an extended TTL. This process amounts to almost 10,000 tokens being stored in Vault on behalf of that user. Now, imagine if 1,000 users were doing the same thing. Soon enough, the memory available to Vault would be exhausted, and the service would refuse to accommodate further requests.

Additionally, consider how connections to the Linux operating system work. When a connection is made to a Linux host, a file descriptor is created to track the connection's status. If Vault runs on a host that limits the number of files, Vault may refuse to accept any further requests due to the operating system's limitations. The connections may be from an increase in the number of use cases or a runaway application that inadvertently sends too many requests.

### *The Solution*

In Vault 1.5, a new feature called *Resource Quotas* was added to Vault to prevent such scenarios. This feature allows Vault operators to set limits on the number of requests per second that can be made to the Vault service or individual mounts. Additionally, Enterprise customers can limit the number of leases that can exist in Vault at any given time. Using a resource quota means that a Vault administrator could limit the number of tokens in a Vault cluster, preventing memory exhaustion on the operating system.

Customers using versions below Vault 1.5 can take other measures to protect the service from DoS events. Placing an L7 network device in the path of the Vault service, such as a proxy or a load balancer, could help manage the number of transactions per second to the Vault service. Depending on the solution, it could even manage the number of requests for specific paths within the Vault service.

Additionally, the operating system's resource limits can be modified to increase the number of available file descriptors. For cloud-based or virtualized deployments, CPU, memory, and disk resources can also be quickly increased to accommodate growing demand, if necessary.

Regardless of what solution is selected, it helps to monitor connections to the host operating system to ensure applications are not leaving open connections to the Vault service. Using a tool such as *netstat* provides an easy way for Vault operators to identify connections in a *CLOSE_WAIT* status. This strategy can help determine whether Vault clients are closing out their connections gracefully.

## Replication Issues/Failures (Enterprise)

One of the most popular features of Vault Enterprise is the ability to replicate Vault data to one or more replicas. Many large enterprises commonly use this feature to distribute critical secrets to various regions and data centers worldwide. If problems arise, replication might be interrupted to only specific replicas or all the entire replication set. Ongoing replication operations that are out of sync can create discrepancies in the data across the clusters.

### *The Cause*

There are many elements throughout an organization's infrastructure that can negatively impact cluster replication. Hardware issues, networking connectivity, or an increase of Vault usage could impact the write-ahead logs from being shipped to secondary clusters. If issues impact the primary cluster, replication may be severely impacted throughout the Vault infrastructure. Other issues may be isolated to specific secondary clusters.

**Hardware Resources**
Secondary clusters should always be provisioned with enough resources to host production data and support the volume of requests from Vault clients. If the cluster has been provisioned with fewer resources (such as storage) than the primary cluster, the secondary cluster will presumably have difficulty keeping up with replication. The cluster may also have difficulty handling Vault clients if it were to be promoted to the primary.

Additionally, the connectivity between the primary Vault cluster and secondary clusters is of utmost importance for replication. As such, the underlying networking hardware used for replication should be able to handle the required traffic. Hardware may include the network interface card attached to the Vault instances and the infrastructure between the nodes. The network infrastructure must be capable of keeping pace with the amount of data being replicated, particularly in the case of the primary cluster. When a primary Vault cluster is attached to many secondary clusters, it can put undue stress on the network infrastructure.

**Secrets Volume**
As changes are made to Vault, logs must be shipped to the secondary clusters to ensure they are in sync. All this data is replicated using TLS over tcp/8201. If there is an influx of changes, the network and underlying resources must support the transport of the data. This increase in data could result in write-ahead logs to become out of sync. For example, if an automated process inserts millions of secrets into a replicated path, the data must also be replicated to all secondary nodes. As Vault attempts to move this data out to other clusters, bottlenecks can occur.

## *The Solution*

Monitoring is key here, and good visibility into the health of replication will ensure the Vault service is in an optimal state for the users. HashiCorp makes guides available to assist with configuring monitoring for the health of replication across clusters.

**Hardware Resources**
Ensure Vault instances are provisioned according to the amount of load that will be placed on them. Starting with the recommended Vault node size is a good practice to avoid issues that might arise from resource exhaustion. The Vault instances should be adjusted up or down depending on the need as the service matures. Third-party monitoring services can be used to determine trends for each Vault cluster.

Finally, verify that all Vault nodes are configured with a network interface that aligns with the anticipated load. For example, if a primary cluster replicates data to ten or more secondary clusters, a 10Gb network interface will likely be required. For physical hosts, ensure the available network paths are configured with redundancy.

**Secrets Volume**
As new use cases are added to Vault, the Vault operations team should work with new end-users to determine how they plan to leverage Vault. This strategy will help organizations ensure that Vault is being used appropriately while ensuring Vault operators understand how to support their end-users. Vault operators should determine the volume generated by each new use case and calculate any changes required for the Vault infrastructure.

If needed, mount filters can be created to reduce the amount of data replicated to secondary clusters. Organizations should assume that replication will not be configured for a particular mount or namespace unless requested by the application teams.

# *Common Migration and Onboarding Patterns*

Once Vault is up and running, Vault operators must understand how to onboard users to the Vault service. While the operations and security teams may be excited about Vault's security and features, the application team and other end users may not share the same sentiment. Care should be taken to ensure onboarding includes education and internal marketing to demonstrate Vault's capabilities to drive the service's adoption throughout the organization.

## Service Adoption

Before jumping into onboarding and migration patterns, it is important to note the importance of promoting adoption. Driving adoption of any new service is frequently overlooked by support teams, although support is one of the most critical parts of offering a service to the organization. If the potential users are not aware of Vault or do not understand its value, it is difficult for the service to flourish and provide a return on investment.

Many issues can impede the adoption of a shared service, including internal politics, lack of urgency, and operational cost. Regardless of the problem, a few essential methods can help drive adoption in an organization.

### *Flexible Standards*

Security and compliance projects are often deferred in favor of other projects. Some high-performing teams do not make time for security until it becomes a priority due to a breach or security incident. In most cases, this deferment is primarily due to either budgetary constraints, unrealistic requirements, or a lack of understanding of the security services.

In most organizations, security and compliance teams define the security objective and explain how to achieve it. As security teams build the solution, operations teams tend to rebuke the service before working towards the defined objective. This strategy often results in failed projects or single-threaded solutions.

Fortunately, Vault can integrate with a variety of different technologies in various ways, allowing it to meet the needs of multiple teams throughout the organization. There are several commonly used tools that practitioners want to integrate with Vault to automate

secrets management in their designated workflows. As described in Armon's foreword, Vault was built by the practitioner, for the practitioner. Vault allows organizations to create a single programmatic function that they can use for secrets retrieval across various clouds, platforms, and tools.

Successful security is a partnership between the development team, the operations team, the compliance team, and the security team. While security and compliance teams should continue defining the outcomes, defining the path to get there should be a collaborative effort. Allowing teams to integrate with Vault in the manner that fits their workflows best will make it popular in any organization. This will also result in quicker integration without compromising security.

### *Relationships of Trust*

Application teams are responsible for the experience of their customers. Failure can negatively impact the credibility of the work being completed. Application teams face the decision to either run all the components of their technology stack themselves or entrust them to other teams. When there is no relationship between the application and operational teams, application teams tend to decide to go it alone and run those critical components themselves. This usually leads to additional technical debt and sprawl.

Vault cannot be successful and operate at the highest standards without the support of a credible team. Thus, it is essential to ensure the support team selected to operate the Vault service is fit to do so. Specific guidance on how to build a strong team is included in the *Choosing the Support Team* section of this book since this is such an important topic.

As employees build trust across various teams, it is essential to build a community within the organization to promote collaboration and knowledge transfer. Not only will this elevate the credibility of the Vault team and the service, but it can also encourage teamwork and a sense of accomplishment for those involved. Additional information on building a community is available in the *Scaling and Optimizing Vault* section of this book.

## Service Onboarding

Vault is a tool designed with a DevOps culture in mind. It was built around concepts such as automation, configuration as code, and continuous integration. These concepts apply to all phases of the Vault service, whether deploying clusters, applications, or onboarding new

users. Vault will feel natural to any users who are already familiar with these concepts. For users new to a DevOps culture, there may be more of a learning curve. It is essential to design an onboarding process that simultaneously meets the demands of the DevOps practices instilled in the organization. Onboarding should be written to satisfy those comfortable with DevOps concepts but also introduces DevOps concepts to those who are not as familiar with them.

DevOps practices and users, alike, demand simplicity. When it comes to onboarding new services, users want a comfortable and pain-free experience. Vault onboarding is no different. However, full self-service can be difficult. Like other services, Vault has limitations that could cause service failures if exceeded. Finding the correct balance in the onboarding process can be difficult, but it is possible.

## *Onboarding Tools*

In addition to onboarding documentation previously discussed, there are a couple of other support tools that may help users in the journey to adopt and integrate with the Vault service.

**Migration Scripts**
While it should come as no surprise, migration scripts are meant to move secrets from one place to another. The chances are high that an organization deploying Vault already had another tool for secrets management. These existing tools may have also used a pathing system similar to Vault for storing secrets. By creating a script, organizations can programmatically and securely migrate secrets from the incumbent tool to Vault. Using such a process eliminates adoption barriers for Vault users and ensures secrets do not end up in an insecure element, like a spreadsheet on someone's laptop.

Examples of migration scripts are readily available on GitHub, thanks to the HashiCorp Vault community. Specific scripts are not included here due to the wide variety of technologies employed by organizations worldwide.

**Security Policy Templates**
Part of the onboarding process permits users to manage their security policy as a form of self-service. Therefore, it is essential to provide guidance on authoring and managing Vault security policies. Generally speaking, a few examples in a Git repository with accompanying links in the Onboarding Guide will suffice.

Additional ideas for support tools can be found in this presentation by James Atwill from Hootsuite (*https://www.hashicorp.com/resources/growing-vault-at-hootsuite*).

## *Onboarding Process*

In most cases, developing an onboarding process depends on the preferences and technical tools of the organization. However, here are some general suggestions for building the onboarding process to ensure that it is scalable:

- The process should be simple and straightforward
- The process should be automated as much as possible
- The process should avoid complete self-service

While the first two items are self-explanatory, the last item may raise an eyebrow. The only reason for limiting complete, full self-service is due to the limitations that exist in Vault (as discussed in the previous chapter) and the desire to avoid hitting them. Although the limitations are generous, reaching a limitation may happen faster if teams have complete access to onboard applications. If full self-service is supported, it is even more critical to ensure those limits are closely monitored.

There are two general ways organizations choose to onboard entities to Vault, per team or application. In Vault Enterprise, a namespace would be created per team or application based on the organization's decision. For Vault OSS, the same would apply, but with mounts rather than namespaces. Either of these methods can work well for most organizations. When onboarding applications, the effects of an operational or security incident are significantly reduced since changes and exposure are limited to specific applications rather than teams who are likely managing multiple applications. However, where there is more granular segregation, there is also a greater likelihood of hitting the Vault service and storage backend limitations.

An example of an onboarding process that organizations can use could be as follows:

1. **A user submits an onboarding request:** The request form should capture all necessary information for the support team to onboard the user or application. New

iterations of the request form can be made for continuous improvements. Some organizations opt to use feature-rich enterprise tools for creating forms include ServiceNow or Cherwell. In contrast, simpler tools, such as Google Forms or Microsoft Forms, may be used in smaller environments.

2. **The onboarding meeting is scheduled:** The initial session is only scheduled with first-time Vault users. The meeting is an opportunity to discuss the plans for integrating with Vault and ensure the users understand how to interact with the service before the initial access is provisioned.
3. **A Git repository is provisioned:** A repository is created for the initial security policy for that user or application. It is from this repo that the security policy is managed long-term. The requestor has access to make pull requests to the repository. Pull requests are peer-reviewed, approved, and deployed via a pipeline.
4. **A namespace or mount is created:** The Vault resource is provisioned, and the policy is attached to the user or application's auth method to permit access to the new resource.
5. **Confirmation is sent:** Pertinent information is sent to the requester to tell them that the request is complete. The requester should confirm that the requested access is working.
6. **Follow up:** Following up every week during the first month ensures that the requestor continues to use Vault and can ask questions or request assistance if needed.

### *Onboarding Phases*

Onboarding should not be available to the entire organization from the start. Users familiar with DevOps tools, APIs, and automation should be able to adopt Vault with few issues. However, some may struggle to understand how to integrate applications and processes. This knowledge gap can generate extra work for the support team as Vault is launched.

Onboarding should take a structured approach that aligns closely with the technology adoption lifecycle. The technology adoption lifecycle, depicted in Figure 5-14, begins with a few innovators initially, expands into a larger number of early adopters, followed by the early and late majorities.

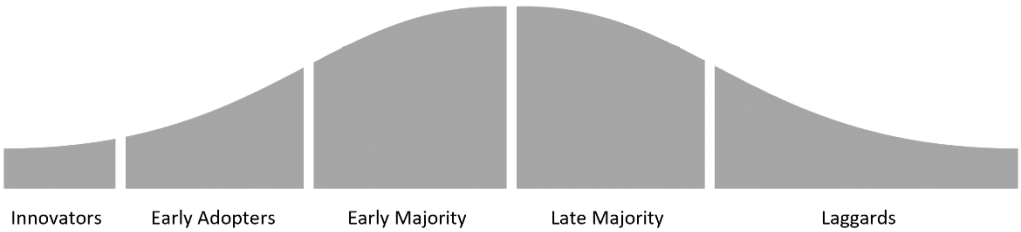



Figure 5-14

**Phase 1**
The first phase of onboarding should begin shortly before the production launch of the service. This phase targets the "innovators" (about 3% of the users) in the organization. This group is most likely aware that the Vault service is being deployed and has already inquired about gaining access. This initial group should be given access to a sandbox environment through the prescribed onboarding process and provided with links to all support documentation. The support team should keep in close contact with this group over the few months leading up to the production launch. The group of innovators should be limited to only several teams so as not to overwhelm the support team as they work towards the Vault production launch. Some of the most crucial feedback comes during this phase, particularly regarding the onboarding process.

**Phase 2**
The second phase begins at the production launch and generally lasts between twelve and eighteen months. During this phase, the early adopter and early majority groups are onboarded. These adopters account for about half of the userbase. For the most part, users in these groups are like the innovators in that they will proactively reach out to the support team to be given access to Vault. At this point, it is unnecessary to track which teams still have not adopted the service.

**Phase 3**
The third phase usually begins about twelve and eighteen months after the service has been running in production. In this phase, the late majority (about 34% of the users) will usually adopt Vault.  The late majority is the group of people that are concerned with reliability. They want to see that Vault has been managed successfully in production for a considerable amount of time. At this point, the support team will likely need to generate a list of teams that have still not adopted Vault and reach out to them to start the onboarding process.

**Phase 4**
The fourth and final phase is an ongoing phase. This phase usually begins about 24 months after launch and continues for the lifetime of the service. At this point, around 90% of the user base has adopted Vault and integrated their applications. Those who have not yet adopted Vault likely have no intention to adopt the service. If this user base is already using an approved tool for secrets management, there is no need to reach out to them. When this is not the case, work with security and management to create an adoption path for the remaining teams.

## *Onboarding Applications*

Once human users have been onboarded to Vault, those users can begin the process of onboarding applications. If namespaces and the onboarding process discussed earlier are used, users will have complete control over application onboarding. If Vault OSS is being used, the Vault operations team may need to assist with application onboarding.

Onboarding an application involves enabling the application to securely authenticate to Vault to retrieve secrets to access backend resources. While there are a few options available for applications to authenticate with Vault, the most common option is AppRole. Taking this into consideration, the onboarding process for applications should follow these steps:

1. **The AppRole for the application is created:** A user would configure the AppRole secrets engine in the team/application namespace, create an AppRole for the application with the designated access policy, and generate the RoleID for the new AppRole. If namespaces are not being used, AppRole can be enabled at the root.

2. **The RoleID is placed in an application configuration file:** Since the RoleID is similar to a username in purpose, it does not necessarily need to be protected in the same fashion as the SecretID. The RoleID can be placed in a Git file with other configuration parameters for the application and called at runtime.'

3. **The SecretID is generated, and a trusted source configures the app:** The SecretID should be provided to the application by a trusted source. This source can be a configuration management tool like SaltStack or a container orchestration platform such as Kubernetes. The trusted source would authenticate with Vault,

generate a SecretID for the specified AppRole, and securely communicate that SecretID to the target nodes or containers. The SecretID could then be stored as an environment variable in memory.

4. **The application uses the configured RoleID and SecretID to authenticate with Vault:** Upon successful authentication, a token is received and can be used to retrieve secrets for the backend resources that the application needs.

Like any process relating to Vault, application onboarding can be fully automated from start to finish. This is particularly true when namespaces are used, and application teams are fully responsible for managing their namespaces.

## Migrations Patterns

When organizations begin to adopt Vault, there are typically two patterns of migration for those adopting Vault. Organizations typically move from another secrets management tool or adopt Vault as the first secrets management tool (usually moving secrets from code). Unfortunately, there are too many tools to cover specific guidance on each one, so the information in this section discusses these migration patterns in a general sense.

### *Migrating from Existing Tools*

Unfortunately, HashiCorp does not provide any tools that assist in migrating data from other secrets management tools to Vault. To do so would take a considerable amount of effort for ongoing maintenance and support. However, Vault is a flexible platform with a fully-featured API, so creating tools to accommodate an organization's needs should be relatively straightforward.

Multiple strategies can be used to simplify migration from other tools. Support documentation and tools should be in place to assist users with this process. More specifically, a script should be created to assist users with migrating secrets from the existing tool to Vault. The support documentation should also outline the specific processes for migrating secrets using the provided script.

Tracking which teams have migrated to Vault may be useful for the Vault operations team to understand who has yet to migrate. Documentation should be created to track which teams are using Vault and their associated use cases. For those needing simple management for

a few static secrets shared between a few human users, migrating to Vault may not be a priority. The teams looking to automate secrets management for processes and applications should prioritize the Vault migration since Vault is an industry-leading solution for these use cases.

Ultimately, onboarding should be as stress-free as possible. The organization might benefit from running a series of brown-bag seminars for internal teams. The format of these seminars involves introducing Vault to the attendees, discussing the migration between tools, and demonstrating the migration script. Participants should be offered an opportunity to ask questions regarding the Vault service and migration. These meetings should last no more than an hour and be offered multiple times to assist users with the adoption and migration to Vault.

### *Migrating from Code*

Unfortunately, secrets stored in plaintext code is still a widespread issue. Many companies are adopting Vault as their first real secrets management tool to solve this very problem. In these cases, the processes and functions created to retrieve secrets for applications and services will be entirely new to the organization.

The migration from code typically begins with an assessment to determine where secrets exist in code. After the initial assessment, teams should map out which applications are dependent on these secrets and subsequently design the new secret retrieval process for each application.

After the application assessments and mappings are complete, code can be developed to retrieve secrets systematically. The new code should be tested with the plaintext secrets still in the code. However, the new code should be the first method used for retrieving the required secret from Vault. If that function fails, the application or service can fall back to the secret in the code. Once confidence is gained with the application's ability to retrieve secrets from Vault, the secret can finally be removed from the code.

# 6 Extending and Integrating Vault

***This Section Covers:***

- Integrating with Kubernetes
- Consul for Service Discovery
- Terraform Integration
- Coding for Reliability

After working with Vault for some time, organizations may look for additional ways to mature the service offering. This chapter discusses how to integrate with existing technologies in order to simplify and automate security throughout the underlying infrastructure.

# *Integrating with Kubernetes*

It is no secret that Kubernetes is taking the container world by storm. With the vast flexibility and scalability Kubernetes offers, there is no denying the benefits of using this declarative system to orchestrate containers throughout the organization. Fortunately, Kubernetes is now a first-class citizen when it comes to Vault. Organizations looking for a seamless workflow for containerized workloads will be pleasantly surprised at the Kubernetes integrations directly supported by Vault.

Over the last several years, HashiCorp has continually added more and more native integrations to Vault. Not only can Vault (and Consul!) be easily deployed with Kubernetes through the use of a Helm chart, but these seamless integrations permit pods, and ultimately the containers within them to access the Vault service to obtain secrets stored or generated by Vault. Containers can easily authenticate with Vault using the Kubernetes auth method while also using the Vault Agent Injector method to render secrets to a shared volume within a pod. This approach allows the containers running workloads to obtain secrets without being Vault aware.

Since HashiCorp has invested a considerable effort to document the Kubernetes integrations for Vault, this section summarizes some of the integrations and why they are useful. Documentation (*https://www.vaultproject.io/docs/platform/k8s*) and step by step tutorials (*https://learn.hashicorp.com/collections/vault/kubernetes*) for Kubernetes integration can be found on HashiCorp's website.

## *Deploying Vault with Kubernetes*

Deploying workloads with Kubernetes can be tricky with the ever-growing landscape of open-source tools and integrations. On top of that, writing a manifest to provision pods, services, and deployments can be a daunting task. Because of these obstacles, organizations are adopting Helm to define, install, and upgrade Kubernetes-based workloads. Helm simplifies the adoption of Kubernetes and the administration of its workloads by enabling functionality like a package manager. Through this new functionality, HashiCorp manages an official Helm 3 chart for Vault to simplify the deployment and adoption of the Vault service (*https://github.com/hashicorp/vault-helm*).

The Vault Helm chart can be used to deploy Vault to internal Kubernetes clusters and managed Kubernetes services, such as Amazon EKS or Google GKE. Regardless of where

the Kubernetes cluster resides, standard best practices should be considered when deploying Vault. Because Vault contains sensitive data and the cleartext encryption key is stored in memory, it is recommended that pods running Vault and Consul do not share a node with other unrelated pods or workloads. HashiCorp recommends running a dedicated Kubernetes cluster specifically for the Vault service.

Vault and Consul cluster nodes should also be evenly spread across fault zones, where available. For example, a cloud-based Vault deployment should make use of multiple availability zones. Pods for an on-premises Vault environment should be spread across physical hosts. By spreading pods out, organizations can provide a secure and highly available service to the organization. When scheduling Vault and Consul pods, each service should be scheduled to run on separate nodes to ensure a performant service. Inter-pod anti-affinity rules should also be used to ensure isolation for the Vault and Consul servers. This approach requires proper labels on the Vault and Consul container definition.

In addition to using Helm to deploy production environments, Helm is an excellent choice for deploying a proof-of-concept or test environment. These environments can be provisioned in seconds so that Vault use cases can be tested with minimal effort. Once testing has been completed, the environments are easily torn down and leave behind no technical debt. For added flexibility, developers or operators can use Vault and Consul Helm charts with Minikube (*https://minikube.sigs.k8s.io/docs)* on their local machines to create a Vault cluster on-demand. This approach is a good alternative to the traditional Vault dev server for fast and efficient testing.

To run Vault and Consul locally with Minikube and the official Helm charts, make sure to check out the tutorial found on the HashiCorp Learn site (*https://learn.hashicorp.com/tutorials/vault/kubernetes-minikube*).

## *Kubernetes Auth Method*

As organizations continue to containerize their applications, Kubernetes has quickly gained popularity and established itself as the container management tool of choice. HashiCorp has made tremendous strides to support Kubernetes as a first-class citizen across all its products, and Vault is no different. Organizations will find that it is essential to integrate these ephemeral workloads with Vault to enable a natively secure and automated way to retrieve secrets.

The Kubernetes auth method allows containers to authenticate to Vault using a Kubernetes Service Account Token. When a container makes an authentication request, Vault verifies the Service Account Token using the Kubernetes TokenReview API to ensure that it is still valid. Assuming the token is valid, the workload is permitted access to Vault based upon the Vault token policy. This approach allows Vault operators to introduce a Vault token into a Kubernetes Pod and grant containers homogeneous access to Vault services.

Like other Vault auth methods, the Kubernetes auth method consists of the primary auth method configuration settings and one or more roles. The role defines parameters such as the Kubernetes service account to be permitted, the Vault policy to be applied, the permitted namespace, and the token TTL. Multiple roles can be configured based upon the needs of the workloads running on the containers.

The general authentication workflow for the Kubernetes auth method follows multiple steps before a container can gain access to secrets stored in Vault. An example of this workflow can be found in Figure 6-1.

1. The container passes the Kubernetes Service Account JWT to Vault
2. Vault verifies the JWT is valid using the Kubernetes TokenReview API
3. If successful, the API returns the permitted names/namespaces, and Vault generates a token with the associated policy
4. Vault returns the token to the container

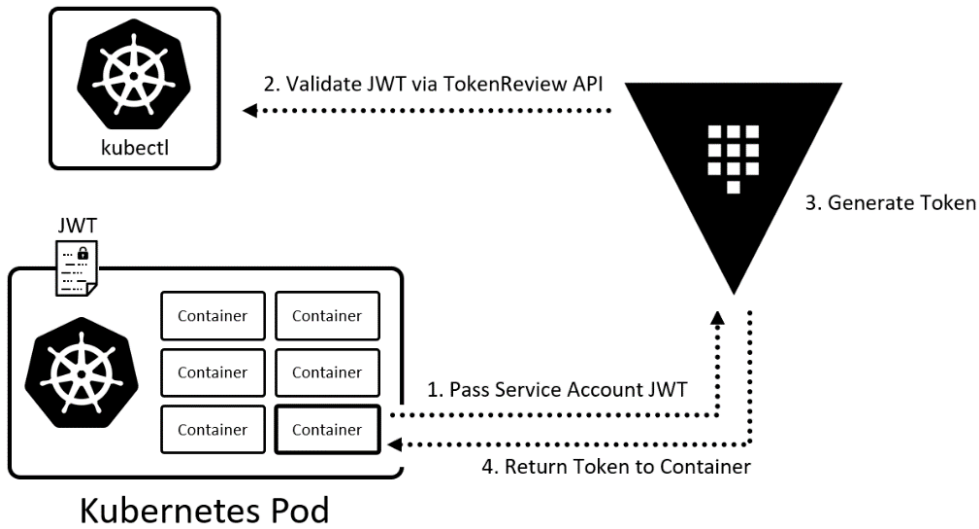



Figure 6-1

## *Configuring the Kubernetes Auth Method*

### Enable the Auth Method

Before the Kubernetes auth method can be used, it must be enabled in Vault using the UI, CLI, or API. To enable using the CLI with the default path, use the command in Figure 6-2.

```
$ vault auth enable kubernetes
Success! Enabled kubernetes auth method at: kubernetes/
```

Figure 6-2

### Configure the Auth Method

Once the auth method has been enabled, the Kubernetes auth method must be configured to establish communication with the Kubernetes cluster. At a minimum, the auth method must be configured to identify the Kubernetes host that will be used to validate tokens. Additional configuration items include the certificate used for secure communications and the service account JWT that Vault uses to access the Kubernetes TokenReview API. Note that a Kubernetes auth method must be enabled for each Kubernetes cluster that will integrate with Vault. Multiple Kubernetes auth methods can be enabled at different paths to enable this functionality.

The command in Figure 6-3 shows how to configure the Kubernetes auth method with these parameters. Note that the value for the *token_review_jwt* parameter should reference the service account created within Kubernetes. This service account should permit access to the TokenReview API.

```
$ vault write auth/kubernetes/config \
    token_reviewer_jwt="vault-auth" \
    kubernetes_host=https://k8s.example.com:8443 \
    kubernetes_ca_cert=@k8s.crt
Success! Data written to: auth/kubernetes/config
```

Figure 6-3

**Create a Role for the Application**

Now that the Kubernetes auth method is enabled and configured, it is time to create a role. A role maps the Kubernetes service account to the desired Vault policy, the TTL configuration, and other parameters needed to satisfy the application requirements or security standards. Remember that multiple roles may be required depending on the different application requirements to access Vault services and data.

The role configuration in Figure 6-4 shows the creation of a role named *app01*, binding the Kubernetes service account to the Vault policy named *policy-k8s-app01* along with setting a TTL to 2 hours.

```
$ vault write auth/kubernetes/role/app01 \
    bound_service_account_names="vault-auth" \
    bound_service_account_namespaces="*" \
    policies=policy-k8s-app01 \
    ttl=4h
Success! Data written to: auth/kubernetes/role/app01
```

Figure 6-4

### *Enabling Secret Retrieval for Containers*

Most organizations running workloads in Kubernetes want containers to access secrets stored in Vault to enable a more dynamic and stateless infrastructure. Although containers can access Vault like any other workload, Vault offers native capabilities to simplify access to secrets from containers, such as the ability to inject secrets into a Kubernetes pod using the Vault Agent.

The ability to inject secrets into a pod can be a compelling approach for organizations that want to integrate container-based workloads with Vault without modifying the application itself. Containers can be configured to use a Vault Agent sidecar, in which the sidecar uses the Kubernetes auth method to retrieve secrets and write them to a path. Since containers running in the same pod share an underlying mount, the workload container can easily read the file containing the secret(s). This approach is enabled using Kubernetes annotations.

Figure 6-5 displays a high-level diagram of how the Vault Agent injector works.

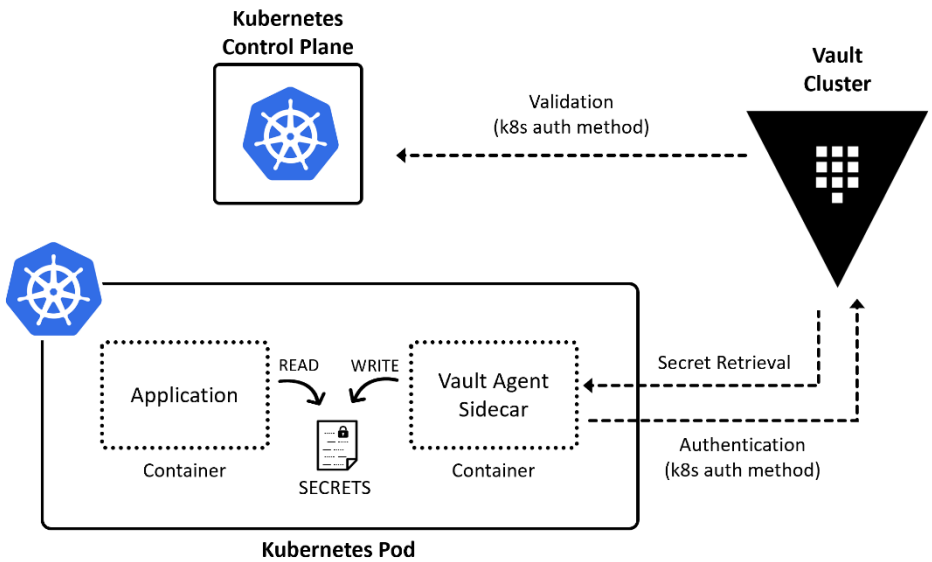



Figure 6-5

More details on using the Vault Agent sidecar, including the specific Kubernetes configuration, can be found on HashiCorp's Learn site (*https://learn.hashicorp.com/tutorials/vault/kubernetes-sidecar*).

## *Consul for Service Discovery*

In a fast-moving business where applications are running on dynamic infrastructure, determining where services are running and how they can be accessed may become a challenge to manage. Fortunately, HashiCorp provides a solution to this dilemma with its Consul product. By instructing applications and services to register with Consul, a service registry can be populated, and applications can query the registry to discover and connect to critical services throughout the organization.

Since Consul is a solution provided by HashiCorp, it should come as no surprise that HashiCorp Vault includes tight integration with Consul to simplify access to Vault. Users and applications can query Consul using either DNS or the API when communication to Vault is required. Consul responds to the query with the information needed to establish communication with the active Vault node. If the Vault environment changes, Consul is immediately aware of those changes and begins directing subsequent requests to the now-active Vault environment. This strategy can help organizations reduce or eliminate the reliance on a front-end load balancer that may not reflect changes fast enough when using a health check configured with timeouts and thresholds.

In addition to maintaining a connection to the active node, Consul can also simplify the connectivity to standby nodes within the cluster. This connectivity may be especially convenient when running Vault Enterprise with performance standby nodes. As previously discussed, these performance standby nodes can service Vault read operations and help Vault operators scale those read operations within a cluster. By using Consul to communicate with the performance standby nodes, clients are always directed to the current nodes based on the Vault cluster's latest changes.

## Registering the Vault Service with Consul

When a Vault cluster is deployed using a Consul storage backend, the Vault service is automatically registered with the Consul service registry by default. However, HashiCorp recommends that the Consul cluster deployed as the storage backend never be used for additional Consul functionality, such as service discovery. This recommendation ensures the Consul cluster can dedicate all available resources to servicing the storage backend rather than other Consul functions. If Consul service discovery is desired, a second Consul cluster should be deployed to manage this functionality.

The *service_registration* stanza needs to be added to the Vault configuration file to register the Vault service with a Consul cluster. This configuration includes the *address* and port of the Consul cluster, the Consul ACL *token* to permit Consul access (if Consul ACLs are enabled), and other configurable parameters such as *service_tags*. Keep in mind that the *token* is a Consul ACL token, not a Vault token. The ACL token needs to be associated with a Consul ACL policy with appropriate permissions, such as reading from the key/value path and creating the Consul service. Figure 6-6 provides an example of the stanza to be added to the Vault configuration file.

```
service_registration "consul" {
   address          = "consul.example.com:8500"
   token            = "a1b2-c3d4e5f-6a7b-8c9d"
   service_tags     = "vault, production, us-east-1"
}
```

Figure 6-6

## Connecting to Vault using Consul

When Vault is registered with Consul, all Vault cluster nodes are registered as the Vault service. In support of the Vault service, Consul also supports multiple tags by default, such as *active*, *standby*, and *performance-standby*. These tags can be used to discover the IP address of a specific node type to establish connectivity. Note that the response for queries only includes Vault nodes of the requested type that are unsealed.

To use Consul to discover the active node in a cluster, a client can query Consul with the DNS name found in Figure 6-7. This DNS name assumes the default Consul domain and service name.

```
active.vault.service.consul
```

Figure 6-7

To discover the current standby node(s) in a Vault cluster, the client can query Consul for the DNS name found in Figure 6-8. If there are multiple standby nodes, Consul responds to the query with all the standby nodes.

```
standby.vault.service.consul
```

Figure 6-8

When running Vault Enterprise, a client or application may need to discover the performance standby node(s) in a Vault cluster for read operations. To do so, the client can query Consul for the DNS name found in Figure 6-9. If there are multiple performance standby nodes, Consul responds to the query with all the standby nodes.

```
performance-standby.vault.service.consul
```

Figure 6-9

# *Terraform Integration*

Vault is one of HashiCorp's most popular products, but its adoption is dwarfed by the popularity of another HashiCorp product, Terraform. Terraform allows customers to define infrastructure components as code to create a predictable and repeatable process for standing up applications and supporting infrastructure. Terraform supports integration with Vault in several ways, including the ability to deploy Vault infrastructure, manage Vault infrastructure, configure Vault components, and retrieve Vault secrets.

## Deploying Vault Infrastructure with Terraform

Organizations deploying Vault to automate the security of applications and infrastructure are often Terraform adopters as well. If so, it is highly likely that these organizations maintain custom Terraform modules for their data center or their selected public cloud platform. Using these modules or modules from the Terraform public registry, customers can simplify a Vault cluster's deployment and ongoing management.

As discussed in the *Designing the Vault Architecture* section, many components must be considered when architecting a Vault solution. Since Terraform supports many private and public cloud resources, organizations can easily define these components in Terraform for deployment. Components such as the Consul and Vault nodes, load balancers, security groups, DNS records, and network connectivity can easily be declared in a Terraform configuration file. Any changes to the environment can be applied using Terraform for simplified management as well.

One of the most significant benefits of deploying Vault with Terraform is repeatability. As Vault adoption increases, so does the need for performance or disaster recovery replicated clusters to ensure high availability and recovery of the service. By developing Terraform configurations for Vault, an organization can readily deploy additional clusters when needed. This strategy also ensures that each cluster is provisioned consistently regardless of where the new cluster is needed.

HashiCorp makes several Terraform configurations available to the public for Vault deployments. These configurations allow organizations to fork the Terraform code and modify the modules based on requirements and configurations of the Vault environment.

Terraform code for Vault deployment for the three biggest cloud providers can be found on GitHub:

- Amazon Web Services (*https://github.com/hashicorp/terraform-aws-vault*)
- Microsoft Azure (*https://github.com/hashicorp/terraform-azurerm-vault*)
- Google Cloud (*https://github.com/terraform-google-modules/terraform-google-vault*)

## Configuring Vault Using the Terraform Provider

Once Vault is up and running, Terraform can configure Vault components using its Vault provider. Terraform's Vault provider facilitates programmatic configuration of a Vault environment as an alternative to manual configuration of Vault components. This approach allows the Vault configuration to be defined as code and simplifies the repeatability for configurations, such as managing namespaces or even multiple clusters. When combined with Terraform code for the Vault infrastructure deployment, the Vault provider can enable the deployment of a turnkey Vault environment.

### *Configure Vault Provider and Credentials*

Getting started with the Terraform Vault provider is like using any other Terraform provider: the provider and related arguments should be declared within the Terraform configuration file. Any sensitive information should be provided using environment variables. This technique ensures that information is not stored in a cleartext Terraform file. For example, if Terraform authenticates with Vault using a token, the Vault address and token should be provided as an environment variable. In this scenario, two environment variables need to be set: *VAULT_ADDR*, which sets the IP address or hostname of the targeted Vault cluster, and *VAULT_TOKEN*, which is used to authenticate to Vault.

Other options for authentication with Vault include the use of other auth methods, such as approle or userpass. The credentials for these auth methods can be passed to Terraform using a variable, as well. This strategy, however, results in potentially sensitive information being stored in a Terraform file if using Terraform open-source. Terraform Enterprise customers can store these as sensitive variables in the workspace.

Figure 6-10 displays an example of the environment variable configuration, while Figure 6-11 shows the provider declaration defined within a *.tf* file. The combination of the

environment variable and the provider declaration enables the connectivity between Terraform and Vault.

```
export VAULT_ADDR = "https://vault.example.com:8200"
export VAULT_TOKEN = "s.g13orv6V96COtRfsENTOjDTw"
```

Figure 6-10

```
provider "vault" {
  alias = "prod"
}
```

Figure 6-11

Alternatively, Figure 6-12 depicts the configuration for the Vault provider when using AppRole for authentication. The values of the *approle_role_id* and *approle_secret_id* variables should be declared in a *.tfvars* file.

```
provider "vault" {
  auth_login {
    path = "auth/approle/login"

    parameters = {
      role_id   = var.approle_role_id
      secret_id = var.approle_secret_id
    }
  }
}
```

Figure 6-12

## *Configure Vault Components Using Terraform*

Now that Terraform has the proper configuration and credentials to communicate with Vault, it can create and manage Vault configurations. A few examples of the configurations that can be managed include secrets engines, auth methods, and audit devices. By writing these

configurations in a modular fashion, a Vault operator can quickly build a Vault environment or demo by consuming the modules needed for the environment.

Earlier in the book, the *Audit Devices* section discussed how to create and manage Vault's audit devices using a manual process. This process can be automated by using Terraform, as displayed in Figure 6-13.

```
resource "vault_audit" "vault_audit_log" {
  type = "file"

  options = {
    file_path = "/var/log/vault_audit.log"
  }
}
```

Figure 6-13

Enabling and configuring an auth method is another example of managing Vault components with Terraform. The *Auth Methods* section discussed how to configure the LDAP auth method and map policies to an Active Directory security group. This configuration process can be simplified through Terraform automation. Figure 6-14 offers an example of this configuration.

```
resource "vault_ldap_auth_backend" "ldap" {
   path     = "ldap"
   url      = "ldap://dc1.example.com"
   userdn   = "ou=User,dc=example,dc=com"
   userattr = "sAMAccountName"
   groupdn  = "ou=Groups,dc=example,dc=com"
   binddn   = "cn=svcvault,ou=User,dc=example,dc=com"
   bindpass = var.ldap_bind_password
}

resource "vault_ldap_auth_backend_group" "adm-group" {
   groupname = "vault-admins"
```

```
    policies  = ["vault-admins"]
    backend   = vault_ldap_auth_backend.ldap.path
}
```

Figure 6-14

### *Consuming Secrets with Terraform*

When organizations use Terraform for resource deployment, credentials are needed to connect to the underlying infrastructure or public cloud provider. Rather than provide Terraform with static credentials, Terraform can integrate with Vault to retrieve or generate credentials before applying the configuration. This strategy eliminates the need for managing long-lived credentials on the infrastructure platform. For example, Terraform can retrieve temporary AWS credentials from Vault to deploy EC2 instances. After the lease has expired, Vault automatically revokes the AWS credentials.

Retrieving secrets from Vault uses the Terraform Vault provider. The provider supports several data sources used to retrieve credentials, including Azure, AWS, and K/V. By using these data sources, Terraform can use these values throughout the configuration. For example, the configuration for the AWS provider can use the value of credentials retrieved from Vault. Figure 6-15 displays the Terraform code used to retrieve an AWS credential from Vault. The credential is then used to pass the value for *access_key* and *secret_key* in the AWS provider, so Terraform can provision resources in AWS using these credentials.

```
data "vault_aws_access_credentials" "creds" {
  backend = vault_aws_secret_backend.aws.path
  role    = vault_aws_secret_backend_role.role.name
}

provider "aws" {
  access_key = data.vault_aws_access_credentials.creds.access_key
  secret_key = data.vault_aws_access_credentials.creds.secret_key
}
```

Figure 6-15

A second example of using Terraform to retrieve Vault can be found in Figure 6-16. This example shows static data retrieval from a K/V path, which includes both a username and a

password. After retrieval, the credential can then be used to authenticate to another provider.

```
data "vault_generic_secret" "f5_lb" {
  path = "secret/platforms/prod/f5"
}

provider "bigip" {
  address  = var.f5_url
  username = data.vault_generic_secret.f5_lb.data["user"]
  password = data.vault_generic_secret.f5_lb.data["pass"]
}
```

Figure 6-16

It is important to note that the data (secrets) read from Vault will be written to the state file when using the Vault provider. Care should be taken to protect access to the state file where possible. This restriction can be done by merely restricting user access to the location, such as an S3 bucket, or configuring Workspace permissions if using TFE.

For more information about the integration between Vault and Terraform, make sure to check out the tutorial on the HashiCorp Learn platform (*https://learn.hashicorp.com/tutorials/terraform/secrets-vault*).

# *Coding for Reliability*

A well-designed Vault solution can provide an exceptional end-user experience under pressure in a variety of different circumstances. However, additional processes can be implemented to provide increased reliability for integrated applications. These opportunities for improvement can often be found outside the Vault solution itself. For example, automated processes can ensure applications can access the vault service, regardless of where the active node is or underlying changes to the environment.

Consider the following scenario at a large tech company, where a team is responsible for managing a shared, multi-tenant Vault service for hundreds of internal teams. Their Vault environment was a large, multi-cluster Vault deployment that spanned the globe, processing hundreds of millions of requests every month. As applications were initially onboarded, each was configured to attach to a single, local cluster. Problems would arise, however, when the local cluster would fail or when maintenance was performed. Despite all the steps taken to build an excellent Vault service, it was clear that more needed to be done to improve availability.

The team took the problem to heart and built solutions to increase reliability and application connectivity across the Vault infrastructure. This section dives into solutions to assist organizations with these types of challenges.

## The Vault Process

As previously discussed, Vault uses a token-based approach when it comes to accessing secrets. It should also be known that these tokens are not replicated to performance replicated clusters within the Vault environment. Applications and users alike would need to reauthenticate to another cluster if the local cluster were to become unavailable.

## The Problem

Providing uninterrupted access to Vault during failover means that the applications and entities must use programmatic functions to authenticate and access Vault through the Vault API. For example, consider the following Vault environment. A primary Vault cluster is deployed in a public cloud region (US-West) with a performance replicated cluster in a second region (US-East) to provide geographical redundancy for an organization. Since replication is enabled, the Vault configuration and its secrets are replicated across the

organization's transit network, making both clusters available to all clients, regardless of their location. During normal operations, applications in each respective region consume Vault-provided services from the local cluster. However, applications need continuous access to Vault in the event of a local Vault service outage. An example of this architecture is depicted in Figure 6-17.

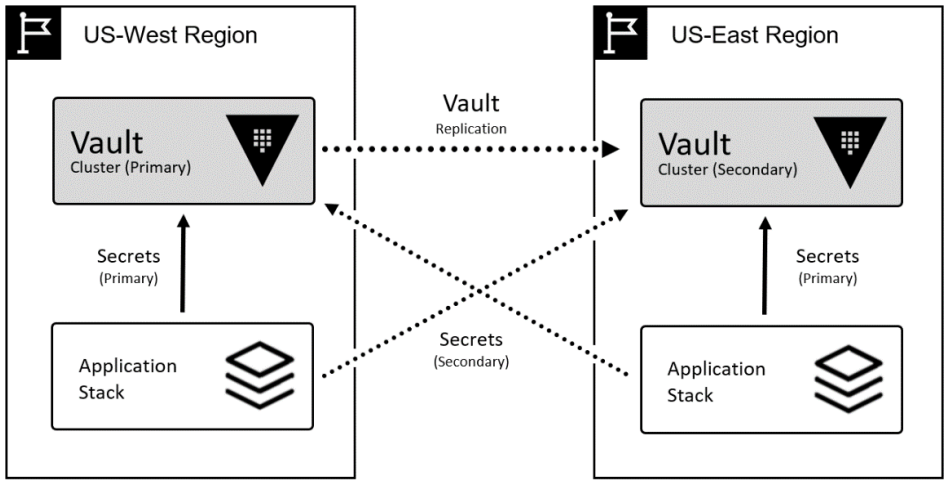



Figure 6-17

A few options exist to support a local cluster failure. The first option is to use native disaster recovery options built into Vault. However, they require downtime during recovery. Since applications rely on Vault for day-to-day operations, they would not function correctly during the recovery process. Application downtime could also violate strict availability requirements set forth by the organization.

A second option to provide continuity during failures would be to employ a load balancer in front of the Vault clusters. However, applications would still need to reauthenticate if the request has been routed to a different cluster since its existing token would not be valid for a secondary cluster. This scenario also requires the application to depend on the load balancer to determine when a cluster is no longer available and route requests to a healthy cluster.

Ideally, the application would have native capabilities to communicate with the secondary Vault cluster to retrieve secrets automatically. This capability would give software and

system engineers more flexibility in determining when a cluster is unavailable and when the application should route requests to the secondary cluster instead.

## The Solution

Historically, operations teams have been responsible for providing a reliable and transparent experience regarding service availability. Cultivating such an experience should be the definitive goal wherever possible. However, there are instances where this cannot be achieved without adding additional complexity or overhead to the environment. The following snippet from the AWS DevOps Cultural Philosophy does a great job of outlining the evolving role of operational teams and how these teams play a fundamental part in the customer experience:

"*[Development and operations teams] take full ownership for their services, often beyond where their stated roles or titles have traditionally been scoped, by thinking about the end customer's needs and how they can contribute to solving those needs.*"

Based on these evolving roles, each team should work in harmony to produce solutions that benefit the organization itself, rather than just focusing on their own team. The operations team can offer infrastructure and Vault expertise, while the development team can provide application competencies. Joining the teams allows for collaboration and the creation of technical solutions that might not have otherwise been possible.

Revisiting the above scenario, the application team needs to interact with Vault in the most efficient way while considering high availability across multiple clusters. Knowing that tokens are not replicated across clusters, the application team needs a solution to authenticate to any Vault cluster to retrieve secrets.

The operations team can start by using the AppRole auth method since both RoleIDs and SecretIDs are replicated across all clusters. Combining this authentication mechanism capability with automation or pre-defined script can ensure applications can always access the Vault service, regardless of the status of the Vault environment.

### *Preparation*

Before application teams can use AppRole for their applications, the auth method must be enabled and configured. A role must be configured for each application and use case, while

unique SecretIDs should be created for each instance of the application. Once the RoleID and SecretID are created, Vault replication will ensure these configurations are replicated to both Performance and DR replicated clusters. This strategy ensures that applications can use the same credentials to retrieve a token on the primary cluster and any performance replicated clusters.

## *Code Overview*

The two example scripts included in this section illustrate a real-world, customer derived solution. The first script includes functionality for retrieving secrets from a single Vault cluster. This method is considered the minimum requirement for applications and services retrieving secrets from Vault and is only provided for the sake of comparison. The second script also includes functions that enhance the availability of secrets retrieval since the script is aware of multiple Vault clusters.

The following points should be considered when using the previously discussed scripts:

- AppRole is used for application authentication. Environment variables are used to configure the RoleID and SecretID in which the application will use to authenticate and retrieve a token. A configuration management tool could be leveraged to securely introduce and manage these environment variables across the application hosts in a real environment.
- The policy associated with this AppRole permits the token to read the value of a single secret. It has no other permissions.
- In this example, the code is not meant to be secure, production-ready, comply with Python best practices, or even run well. It is meant solely to illustrate how development efforts can enhance the capabilities of Vault architecture.

## *Code Process*

### The Vault_Call Function

The *vault_call* function illustrated in Figure 6-18 is a base function responsible for executing *curl* commands to interact with Vault. Since many of the same Python calls are used to authenticate and retrieve a token, they are consolidated into a single function. This function can also be found on GitHub (*https://github.com/btkrausen/hashicorp/blob/master/vault/scripts/vault_call.py)*

```
def vault_call(vserv, vact):

    crl = pycurl.Curl()
    data = BytesIO()
    base_url = "https://" + vserv + ":8200/v1/"

    if vact == "login":
        vault_url = base_url + "auth/approle/login"
        role_id = os.environ.get("ROLE_ID")
        sec_id = os.environ.get("SEC_ID")
        login_data = json.dumps({"role_id": role_id, "secret_id":
sec_id})
        crl.setopt(crl.POSTFIELDS, login_data)
    elif vact == "get":
        vault_url = base_url + "secrets/data/myapp"
        vault_header = ["X-Vault-Token: " + cluster_token[vserv]]
        crl.setopt(crl.HTTPHEADER, vault_header)
    else:
        print("Invalid Option")
        exit()

    crl.setopt(crl.URL, vault_url)
    crl.setopt(crl.WRITEFUNCTION, data.write)

    try:
        crl.perform()
        resp_data = json.loads(data.getvalue())
        resp_code = crl.getinfo(pycurl.RESPONSE_CODE)
    except pycurl.error:
        crl.close()
        return "Connection Failed"

    crl.close()

    if resp_code != 200:
        return resp_code
    elif vact == "login":
        return resp_data["auth"]["client_token"]
    elif vact == "get":
        return resp_data["data"]["data"]
    else:
        return resp_data
```

Figure 6-18

In the first half of the function, the *curl* command is instantiated and assembled based on the Vault cluster (vserv) and action (login or get) selected. The *ROLE_ID* and *SEC_ID* environment variables are called if the "login" action is selected. These environment

variables are references to the RoleID and SecretID associated with the *myapp* role and are required to authenticate with Vault and retrieve a service token. If the *get* action is selected, this service token is passed in the HTTP header to identify the entity to retrieve the desired secret.

In the last half of the function, which begins with the "try" statement, the assembled curl command is executed. If the *curl* command is executed successfully, both the return code and the response data are collected and stored as variables. If the *curl* command fails for any reason, a "Connection Failed" error is returned. A resulting error usually means that either endpoint did not exist or did not answer. This error could also be caused by a bad or misconfigured DNS name.

Once the curl command is executed and the responses are stored in variables, the code closes the Vault endpoint connection and determines what response it received from Vault. If the response was successful (200), the code will either determine the value of the service token ("login") or determine the value of the secret ("get") based on which action was issued to the function. It will then return that data to the entity.

If a response code other than 200 were to be received, that value would be returned instead. Possible response codes include 403 (permission denied), meaning the token does not have permission to the requested path, or the token has expired. A 404 (invalid path) means that the secret could not be found on the specified path due to an incorrect path or missing secret.

**The Get_Token Function**

The get_token function will enable retrieval and ongoing management of service tokens. This function is shown in Figure 6-19 and can be found on GitHub (*https://github.com/ btkrausen/ hashicorp/blob/master/vault/scripts/get_token.py)*

```
def get_token(vserv):

    vtoken = vault_call(vserv, "login")

    if not re.match(r"(^s.{25}$)", vtoken):
        return False
    else:
        cluster_token.update({vserv: vtoken})
```

```
    with open("tokens_mem", "w+") as fh_:
        json.dump(cluster_token, fh_)

    return True
```

Figure 6-19

The role of this function is simple. It will attempt to authenticate with Vault using the *vault_call* function above and retrieve a service token. This function leverages regex to verify that the response contains a valid service token. If successful, the token is stored in a service token object in memory along with the name of its associated Vault cluster. If a valid service token is not received, the function responds with a *False* to the calling function.

The overall goal is to maintain an object where the name of a Vault cluster is stored with the associated service token retrieved from that cluster. This approach allows the code to leverage these cached service tokens to avoid retrieving a new service token every time an action is required.

### *Non-HA Script Overview*

The script found in Figure 6-20 leverages the two functions above to complete the necessary steps to interact with the Vault service. If there is no token, it will retrieve one from Vault and subsequently retrieve a secret in Vault. This script can be found on GitHub (*https://github.com/btkrausen/hashicorp/blob/master/vault/scripts/non-ha-script.py*)

```
cluster_token = {}

def get_secret():

    global cluster_token

    with open("tokens_mem", "r") as fh_:
        cluster_token = json.load(fh_)

    if "vault-east" not in cluster_token:
        if not get_token("vault-east"):
            return "Unable to Retrieve Token"

    secret = vault_call("vault-east", "get")
```

```
    if secret == "Connection Failed":
        return "Unable to Retrieve Secret"
    elif secret == 403:
        if get_token("vault-east"):
            secret = vault_call("vault-east", "get")
        else:
            return "Unable to Retrieve Secret"
    elif secret == 404:
        return "Empty Secret"

    return secret
```

Figure 6-20

This function begins by instantiating the *cluster_token* global variable. This variable is the object that stores the service token with the relative Vault cluster name, as mentioned above. The next two lines of code load the known token from "memory" into the *cluster_token* variable. For the sake of simplicity, a text file is used to mimic this behavior.

The next lines of code are used to determine whether a service token already exists for the selected Vault cluster. If a service token already exists for the specified cluster, it moves on to the next part of the function. If a token does not already exist in the *cluster_token* object, then the *get_token* function above is invoked. If the function cannot retrieve a valid service token, it will exit with an "Unable to Retrieve Token" error.

The remainder of the function attempts to retrieve a secret. The *vault_call* function outlined above is called and stored in the secret variable. The function then attempts to determine whether it received a valid response from Vault. If this function receives a "Connection Failed" error from the *vault_call* function, the script will exit with an "Unable to Retrieve Secret" error. If the *vault_call* function receives a valid response from the selected Vault cluster, but the response is an error code, the script will exit with the appropriate error message. Otherwise, if all checks pass, the value of the specified secret will be returned.

## *HA Script Overview*

The *Non-HA* script above used a static value for the Vault cluster (*vault-east*) that it was interacting with to retrieve secrets. This approach is not resilient since the script is unaware of other Vault clusters. If the script cannot communicate with the hard-coded cluster, it will be unable to retrieve secrets. By including a reference to the performance replicated cluster

(*vault-west*), the *HA* script can leverage the highly available Vault architecture. If the primary cluster is unavailable, the script can alternatively retrieve secrets from the secondary cluster. The script can be found on GitHub (*https://github.com/btkrausen/hashicorp/blob/master/vault/scripts/ha-script.py*).

Note that the Vault clusters are stored in a Python list object in the order of preference. Python honors order for list objects. Therefore, the script will always attempt to interact with the *vault-east* Vault cluster before communicating with the *vault-west* Vault cluster.

```
cluster_tokens = {}
vault_clusters = ["vault-east", "vault-west"]

def get_secret():

    global cluster_tokens

    with open("tokens_mem", "r") as fh_:
        cluster_tokens = json.load(fh_)

    for vserv in vault_clusters:
        if vserv not in cluster_tokens:
            if not get_token(vserv):
                if vserv != vault_clusters[-1]:
                    continue
                else:
                    return "Unable to Retrieve Token"
            else:
                break
        else:
            break

    for vserv in vault_clusters:
        if vserv in cluster_tokens:
            secret = vault_call(vserv, "get")
        else:
            continue

        if secret == "Connection Failed":
            if vserv != vault_clusters[-1]:
                continue
            else:
```

```
                return "Unable to Retrieve Secret"
        elif secret == 403:
            if get_token(vserv):
                secret = vault_call(vserv, "get")
                break
            else:
                continue
        elif secret == 404:
            continue
        else:
            break

    return secret
```

Figure 6-21

Like the previous function, this *HA* function begins by instantiating the *cluster_tokens* global variable and loading the known tokens from "memory" into the *cluster_tokens* variable. Again, a local text file is used to mimic this behavior.

The *get_token* function is the first part of this script to take advantage of the highly available Vault architecture. After checking for an existing service token for the selected cluster, the script continues to the next part of the function. If a token does not already exist in the *cluster_tokens* object, then the *get_token* function is invoked. If the function cannot retrieve a valid service token from the selected cluster, it will deviate to the next cluster in the list and attempt the service token retrieval process again. Failure to retrieve a token from all Vault clusters will return an "Unable to Retrieve Token" error.

If the function successfully retrieves a service token, the *vault_call* function is called and stored in the secret variable. This section of the function will only use a Vault cluster where an available service token has been stored in the *cluster_tokens* variable. The function then attempts to determine whether it received a valid response from Vault. If a 403 (permission denied) response code is received, the function will attempt to fetch a new service token using the *get_token* function. If any other error is received, the function will attempt to retrieve a secret from the next Vault cluster in the list. It will continue to do so until it has attempted to retrieve a token from all Vault clusters. If the function has exhausted the Vault clusters without receiving a response, it will return an "Unable to Retrieve Secret" error. Otherwise, if all checks pass, the value of the specified secret will be returned.

### *Benefits of the HA Script*

With only nineteen lines of additional code, the Vault environment portrayed above benefited from significantly improved resiliency in the secret retrieval process. These are real-world scenarios where small changes have yielded an improvement from 99.9% availability up to 99.99% availability for most Vault operations. The caveat is that the service will be unable to perform any write operations in a situation where the primary cluster is temporarily unavailable. If there are relatively few write operations being initiated (which will be true in most cases), this is not a substantial concern.

## Additional Considerations

In addition to the guidance above, additional considerations can improve the resiliency of applications and processes when interacting with Vault.

- Retry timers for secret retrieval should be built into the code to accommodate for situations where there are no available Vault clusters. Using this method will help prevent the application from overwhelming the Vault service.
- Caching secrets in memory can alleviate the need to make frequent requests to Vault secret retrieval.
- When using dynamic secrets, managing TTLs inside the application allows the application to request a secret only when it expects the secret to change.
- When working with static secrets, it might be helpful to subscribe to an event system that can notify the application when a secret value has changed. This design allows the application to reach out to Vault to refresh the secret as needed. This approach can also be accomplished with Consul Template (*https://github.com/hashicorp/consul-template*)
- A log aggregation tool can be used to ingest audit logs and report when the practices illustrated in the script are not being followed. Vault operators can then work with the application owners to improve best practices when it comes to retrieving secrets from Vault.

# *What's Next?*

Thanks for reading our book. We hope you enjoyed it and are ready to deploy and manage Vault in your own production environment. ***If you loved the book, please reach out and let us know on social media. We'd love to hear from you.***

### HashiCorp Certification

If you have not yet earned the *HashiCorp Certified: Vault Associate* certification, give it a shot. Although this book was not intended to be a study guide, it does cover many of the topics tested on the exam. Check out the exam blueprint (*https://www.hashicorp.com/certification/vault-associate)* and the study guide (*https://learn.hashicorp.com/tutorials/vault/certification-vault-associate*) to get familiar with the objectives and topics covered.

### Supplemental Training

If you are looking for online training on Vault, including introductory or Advanced topics, check out all the resources Bryan has available at GitHub (*https://github.com/btkrausen/hashicorp*). He has taught over 6000+ students skills on HashiCorp products, such as Vault, Terraform, and Consul. Links to online training courses, as well as the latest discounts codes, are always frequently updated. Courses include the highly-rated *Getting Started with HashiCorp Vault* course, the *HashiCorp Vault: The Advanced Course*, and a *Vault Associate practice exam*. There are also resources for additional HashiCorp products here as well, such as Terraform and Consul.

### HashiCorp User Group (HUG)

If you are passionate about one or more HashiCorp products, chances are you have something to share to help others. Make sure to check out the HashiCorp User Group in your local city. We believe community is a huge part of a successful and thriving IT career, so get out there, join your local HUG, and share your story with others.