HashiCorp
Vault

Modern-Day PKI Management with HashiCorp Vault

WHITEPAPER

Contents

Preface 1

What is PKI 1

Common Uses for PKI 4

Securing Communications 4

Authenticating Users and Systems (SSH) 5

Signing and Encryption 6

Traditional PKI Management 7

People and Process 7

Lifecycle Management Patterns 7

Shortcomings with Traditional PKI 9

People 9

Process 10

HashiCorp Vault Specializes in Internal PKI 11

PKI Design Overview 12

Core Elements of PKI 12

Improving Certificate Lifecycle and Security 13

Example Scenario 14

Designing the Vault PKI Solution 16

Design Considerations 16

One CA, One Secrets Engine 16

Configure CRL/OCSP in Advance 17

Safe Minimums 17

Token Lifetimes and Revocation 18

Patterns: Managing Root Certificates 19

What is a Root Certificate? 19

Vault PKI Root CA Anti-Patterns 19

External CA as a Public Signing Authority 19

The Root CA as an Issuing CA 20

When Vault should Manage the Root CA(s) 20

Lifecycle Management 21

Preparation 22

Root Certificate Generation 22

Certificate Expiration, Renewal, and Revocation 23

Securing the Vault CA using an HSM 24

Securing the Root Private Key Generation 25

Key Length and Algorithm Selection 25

Root Certificate Validity Period 26

Defining Root CA Constraints 27

Monitoring Recommendations 28

Patterns: Managing Intermediate CA 33

Integrating with Existing CAs 35

CA Hierarchy Design Recommendations 38

Intermediate CA Anti-patterns 38

Common Intermediate CA Management Patterns 39

CA Lifecycle Management 49

CA Validity Time Periods 52

CA Succession 55

1. Renew CA certificate TTL only 55

2. Replace CA without changing the private key 58

3. Replace CA and rotate the private/public key pair 60

4. Replace Intermediate CA and Root CAs 63

Certificate Revocation List and OCSP 65

Revoking CA Certificates 67

Vault and SPIFFE/SPIRE 68

Monitoring/Audit Recommendations 70

Operation 70

Example path and data snippets 70

Patterns: Managing Leaf Certificates 73

Anti-Patterns 73

Lifecycle Management 74

Preparation 74

Certificate Issuance 74

Usage and Validation 78

Renewal/Rotation 79

Deployment / Automation 83

Example Vault Agent Configuration 84

Revocation 85

When do I need to revoke leaf certificates? 85

Revoking leaf certificates using the lease ID 86

Revoking leaf certificates via certificate serial number 88

Revocation checking for clients 88

Viewing revoked certificates in the CRL 90

Validating certificates via OCSP 90

Tidying 91

Common Configurations 92

Leaf Certificate Time-To-Live 92

Time-To-Live Hierarchy 93

Maximum TTL for the leaf certificate from highest to lowest. 93

Precedence (highest to lowest) 93

Example Configuration 93

Other Leaf Certificate Considerations 94

Monitoring / Audit Recommendations 95

Generating a certificate 95

Revoking a certificate 96

Secure Introduction 99

Securely Introducing Vault Clients 99

Platform Integration 99

Trusted Orchestrator 100

Automating Secure Introduction 101

Vault Agent Auto-Auth 101

PKI Solution Architecture 103

Deployment Reference Architecture 103

Hardware Considerations 103

Consul Reference Architecture 106

Vault Reference Architecture 108

Multi-Region Deployment Reference Architecture 110

Production Hardening 112

Scale and Performance Considerations 114

Health Monitoring Recommendations for Vault 115

Deployment Anti-Patterns 116

Gathering Service Level Requirements 116

Determining Service Level Objectives 118

Mapping Service Level Indicators 118

PKI Use-Case Design 120

Overview 120

Assumptions 120

Configuration 120

Overview 121

References 121

Steps 121

Param 123

Description 123

Integrations 124

Hashicorp Consul 124

Kubernetes 127

How-To 134

Request a Certificate 134

Revoke Certificates 136

Remove Expired Certificates 137

SSH Login 138

Automate Endpoint Certificate Rotation 144

Contributors: *Dan Mcteer, Patrick Gryzan, Kawsar Kamal, Neil Dahlke, Fraser Pollack and Marc Chua.*

Preface

As technology continues to improve at an increasingly rapid pace, so do the threats to the information being managed by that technology. Because of these growing threats to information security, it is necessary to protect sensitive data through all stages of management.

One of these critical data management stages is the "transit" stage. The "transit" stage involves moving sets of data between infrastructure and services to allow various services to utilize that data in several ways. Protecting data during this stage is just as necessary as protecting data at rest.

But how can secure communication be managed in the era of microservices, where thousands of even tens-of-thousands of microservices need to share data? Manual certificate deployment is no longer capable of meeting the growing demands of scale and speed. Automation must play a key role in certificate management to meet these demands and ensure trust in the exchange of data.

The purpose of this document is to outline a more modern approach to PKI management that solves the growing demand for scale and speed in an automated fashion, eliminating both security and operational compromise that regularly come as a result of human intervention.

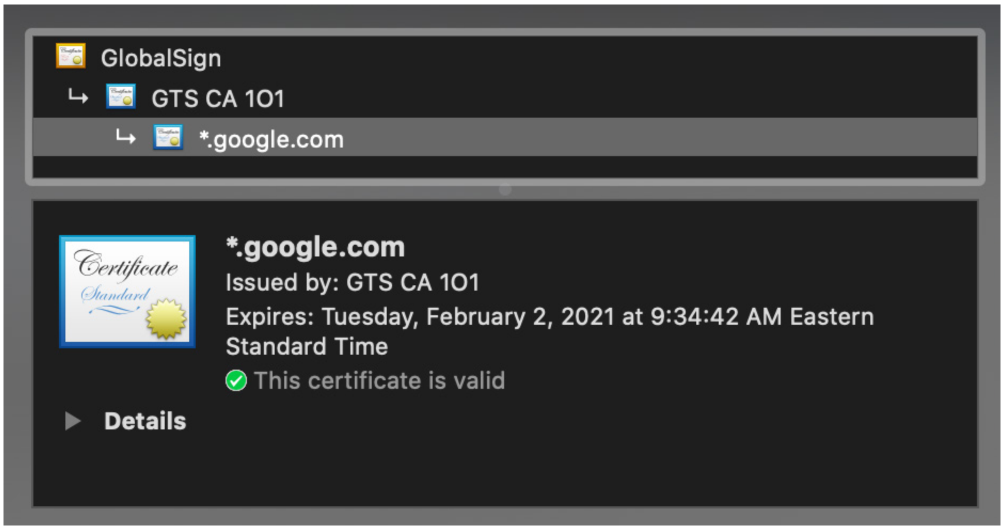
This document is intended to be used by technical staff tasked with deploying PKI management solutions in greenfield environments.

What is PKI

A public key infrastructure (PKI) is a set of roles, policies, hardware, software, and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. The purpose of a PKI is to facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking, and confidential email. - Source - Wikipedia

PKI is generally required and recommended when passwords or other authentication mechanisms are considered insufficient to **verify and validate** the user's identity. The components of a standard PKI infrastructure are;

- **Certificate Authority (CA)**
  - A Certificate Authority (CA) is a company or organization that acts to validate the identities of entities (such as websites, email addresses, companies, or individual persons) and bind them to cryptographic keys through the issuance of electronic documents known as digital certificates. [1]
- **Registration Authority (RA)**
  - A Registration Authority (RA) is an authority in a network that verifies user requests for a digital certificate and tells the CA to issue it.
- **Chain of Trust (Chain)**
  - A Chain of Trust Is made up of a list of certificates that generally starts with a server's certificate, is followed by an intermediate certificate, and finally terminates with the Root certificate. Figure 1-X displays a simple chain of trust.


GlobalSign
↳ GTS CA 1O1
↳ *.google.com

***.google.com**
Issued by: GTS CA 1O1
Expires: Tuesday, February 2, 2021 at 9:34:42 AM Eastern Standard Time
This certificate is valid

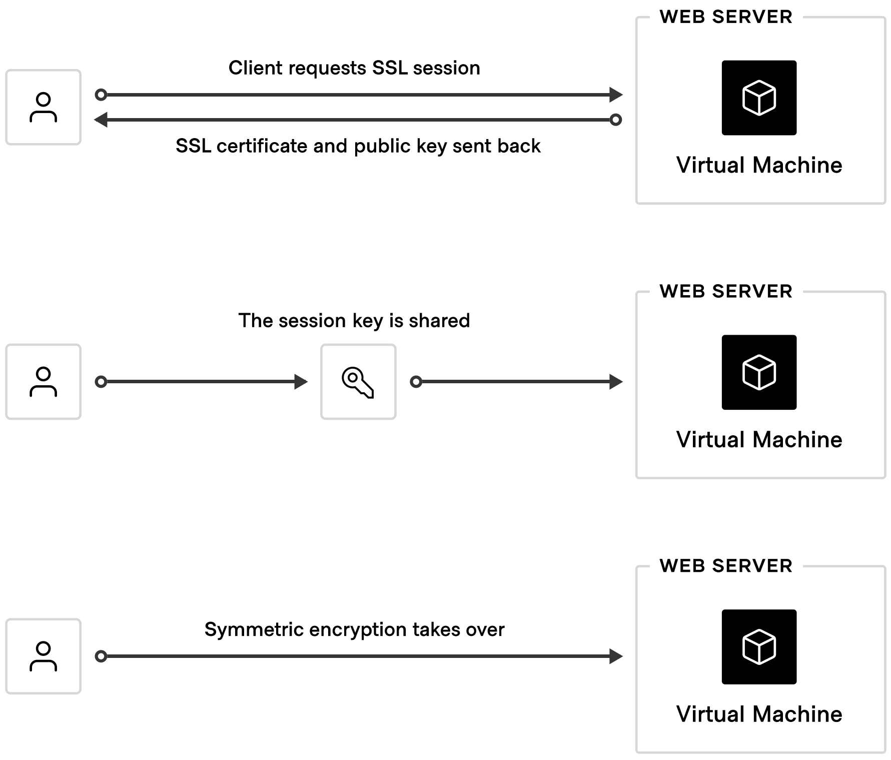
Certificate Chain of Trust

- **Applicant**
    - An Applicant is an entity or user requesting a certificate
- **Certificate Signing Request**
    - An applicant or requester for a certificate generates a certificate signing request. This request is also paired with the public key and presented to the CA for signing. The CSR contains information such as;
        - Common Name
        - Organization Name
        - Organization Unit
        - Etc
- **Signed Certificate**
    - A Signed Certificate is the data file that digitally binds a cryptographic key to organization detail. This data file is the final entity in the chain of objects presented to the end client.

Common Uses for PKI

Securing Communications

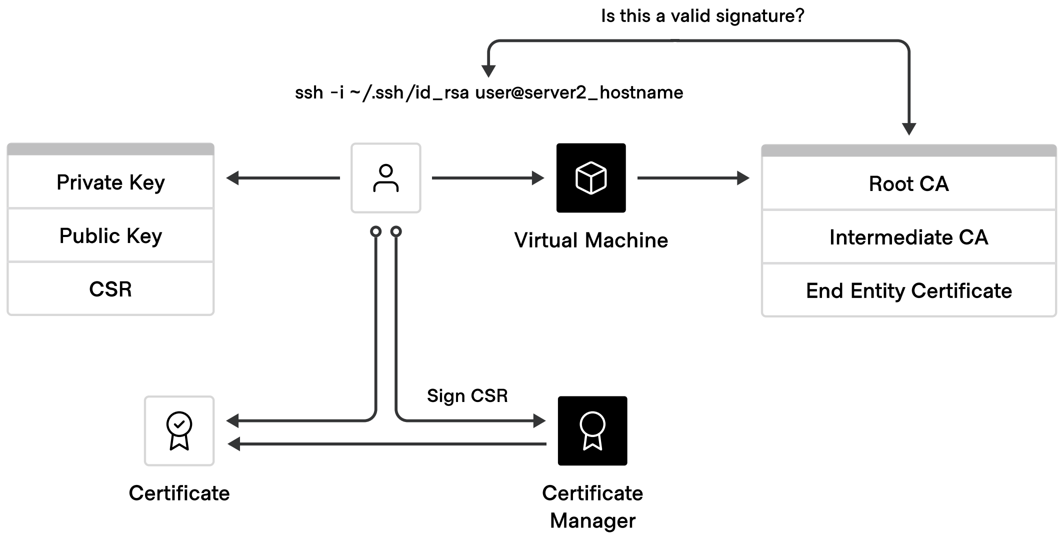
When a certificate is installed on a web service, it allows the use of the HTTPS protocol to secure communications between a web server to a client browser. The presence of HTTPS communications is often indicated by a padlock or green fill in the URL bar. Communications through HTTPS are designed to prevent an attacker from impersonating a valid organization or other entity, as well as protect the exchange of data from being viewed by something or someone other than the intended parties.


WEB SERVER
Client requests SSL session
SSL certificate and public key sent back
Virtual Machine
WEB SERVER
The session key is shared
Virtual Machine
WEB SERVER
Symmetric encryption takes over
Virtual Machine

SSL Client Handshake Process

Authenticating Users and Systems (SSH)

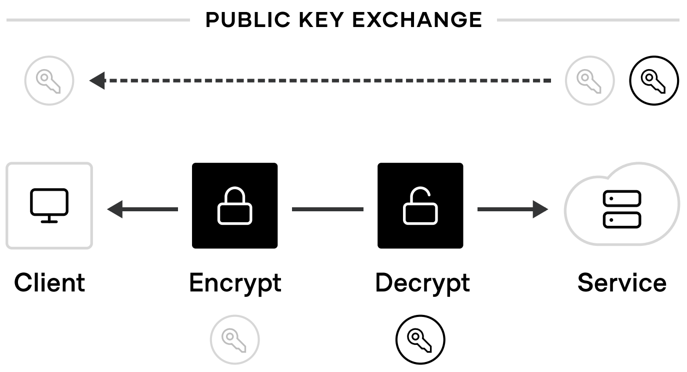
When authenticating to a host, there are typically two forms of identification presented to the host to verify the user; a username and password or an identity file. In the example below, the user passes a **-i** flag, which allows the user to select a file from which the identity (private key) for RSA or DSA authentication is read. This identity file has typically been signed by the CA and is validated by the destination host against its own configured, signed certificate chain. Figure 1-X displays this authentication process.



Certificate Authentication Process

Signing and Encryption

The simplest form of PKI consumption is signing and encryption. A plain text message is **encrypted** using the recipient's public key and produces a form of text known as ciphertext. That ciphertext is then **decrypted** using the recipient's private key producing the original plain text message again. This cryptographic process is outlined in Figure 1-X.



SSL Communications Overview

Traditional PKI Management

Outlined below are some of the everyday experiences and practices organizations have reported to HashiCorp as part of our discovery process when trying to understand how those organizations manage PKI.

People and Process

Typically enterprises employ a team to **manage their PKI certificate infrastructure**, another team to **process certificate requests**, and an operations team to **deploy certificates to the desired entity** or service. In some cases, developers may even be responsible for **configuring their applications with the certificates.** A process such as this provides several different touchpoints where sensitive information is being handled by several people throughout the organization. This process also creates many different instances where a requestor must wait on a certificate to be provisioned and installed

Some organizations have attempted to optimize this experience by providing a front-end web portal for processing certificate requests as long as the required approvals are provided. The requestor can then self-serve a certificate to some extent as needed.

Lifecycle Management Patterns

Various patterns are exhibited in organizations when managing the lifecycle of PKI certificates. The three patterns outlined below are the most common:

- Generating wildcard certificates that are extremely broad in their application (e.g., *.example.com) and using that wildcard universally or as much as possible at the organization.
    - This wildcard certificate is generated early on in the company's domain footprint and is kept in some object repository as a simple solution for those needing to secure traffic to a service. Access to this certificate is controlled by a request portal, and almost every application, website, and user consumes it in some way.
    - When the wildcard certificate is about to expire, a centralized operations team or certificate management team regenerates it for an extended period of time and goes about a company-

wide effort to contact all the stakeholders consuming it.

- Generating exceptionally long-lived certificates in an effort to reduce renewals and outages due to expiration.
    - A quick web search dictates that the most secure recommendation is that PKI certificates be valid for no longer than one year.
    - Although not publicly available, several reference customers anecdotally report many internal certificates eclipsing this period by double or triple that duration.
- Using tools and dashboards to report on certificate validity but then manually managing their lifecycle and rotation
    - There is a litany of tools that offer certificate endpoint monitoring and workflow management for organizations.
    - The issue arises when organizations use the monitoring and reporting tools to transition to manual management workflows. An alert is triggered that a certificate is expired, which is then manually reviewed by a certificate management team. The certificate management team then engages the stakeholders to go through a process of reviewing the entire ecosystem for the consumption of that certificate and its associated dependencies, finally updating the certificate upon the completion of necessary due diligence.

Shortcomings with Traditional PKI

Based on our discovery above, HashiCorp has identified several principle issues with traditional enterprise PKI management practices. The value of reviewing these practices and making improvements can be summarized in three points:

- **Cost:** Gartner reports that the average cost of application and network downtime is now $5,600 per minute or over $300k p/hour - *Source* - The Cost of Downtime
- **Risk:** Digital Guardian reports that the average risk associated with a data breach or a lapse in security (certificate expiration, compromise, or otherwise invalid) cost on average $8.19 million in 2018. - *Source* - What is the Cost of Data Breach?
- **Speed:** The Standish Group International, an independent research firm renowned for their analysis of IT processes, found that organizations adopting Agile "people and process techniques" are six times more successful with one-fourth of the cost. - *Source* - Agile vs. Waterfall

With the return on investment of evaluating a PKI solution identified, it's helpful to map out concerns along with validated improvements that will **positively** impact these shortcomings.

People and Process

In keeping with the typical approach of analyzing information technology systems using People, Process, and Technology dimensions, it is evident in the case of PKI that the technology is stable and a widely-adopted standard. In short, the core technology is not going to change. People and Process are the areas of influence that can be used to close risk gaps while enabling efficiencies and opening doors to new possibilities for PKI management.

People

- Every person that is involved with the process to procure a certificate is handling sensitive material
- Orphaned key generation material is everywhere (local desktops, jump boxes, random virtual machines used to run **OpenSSL**)

- Process management is slow with too many people involved; I request a certificate, it goes to the next person, and so on until a ticket is passed between multiple teams.
- Separate teams to manage the certificate lifecycle, write policies, dictate regulations, and validate certificates (security, ops, dev, etc.)

Process

- Companies report on expiring certificates but use a manual process to mediate.
- Manual processes typically involve maintenance windows for minor changes (put a new certificate in place, restart the associated service(s)).
- Certificate consumption sprawl (one cert used in MANY MANY different places)
- Stakeholder fatigue leads to certificate expiration; in some cases, applications continue to work in an "insecure" fashion.
- Long-lived certificates give additional time to a bad actor for penetration and infiltration of sensitive data and communications.
- Long-lived certificates are given to more people within the organization, creating a larger attack surface (the number of people or applications consuming the same certificate).
- Managing certificates is tedious work (endpoint management, consumption, auditing processes).
- Audits are challenging with manual certificate management as finding the trail of information sometimes requires the auditor to look in many different places.

The remainder of this document explores how an organization can improve the People and Process aspect of PKI management and how HashiCorp Vault can assist with that optimization.

HashiCorp Vault Specializes in Internal PKI

HashiCorp focuses on internal PKI workflows because we can own the end-to-end lifecycle, objects (root to leaf certificate), and policies. By managing the entire PKI lifecycle, we can optimize the experience and provide a workflow regardless of the end technology. Read more about HashiCorp's design philosophy and tao here.

PKI Design Overview

Core Elements of PKI

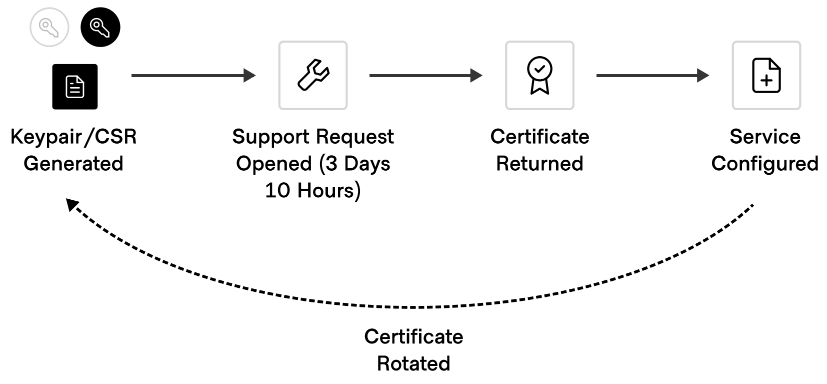
Several core components exist in every traditional PKI deployment and are critical to ensuring the service operates in a secure and reliable manner. These components are outlined below:

- **Private Key:** Used as part of the client/service communication process.
- **Public Key:** Used in both the certificate request process and the client/service communication process.
- **Certificate Signing Request (CSR):** Created on behalf of the service for the purpose of requesting a certificate from the certificate authority.
- **Certificate Authority (CA):** Responsible for creating/signing certificates, validating certificate authenticity requests, and managing certificate revocation lists.
- **Certificate Revocation List (CRL):** A list of certificates that have been issued by the certificate authority but were revoked prior to their configured expiration date.
- **Root Certificate Authority:** This is the top-level certificate authority for all other certificates in the tree.
- **Intermediate Certificate Authority:** This is a certificate authority with authority to generate and sign certificates on behalf of the root certificate authority. Intermediate certificate authorities are generally used to protect the root certificate authority by handling the day-to-day certificate operations.
- **Leaf Certificate:** This is the generated and signed certificate that is used by a host or service entity to establish trusted communications between a client and that entity.

As described in the previous section, the various portions of the certificate lifecycle involving the above components are generally distributed across multiple teams, which usually causes significant delays in the creation and renewal of certificates. Due to these delays, operational fatigue usually triggers the decision to extend certificate expiration to unsafe levels. Additionally, this shift to longer expiry has an added consequence of generating enormous CRLs because certificates with a longer lifetime generally have a greater chance of needing to be revoked.

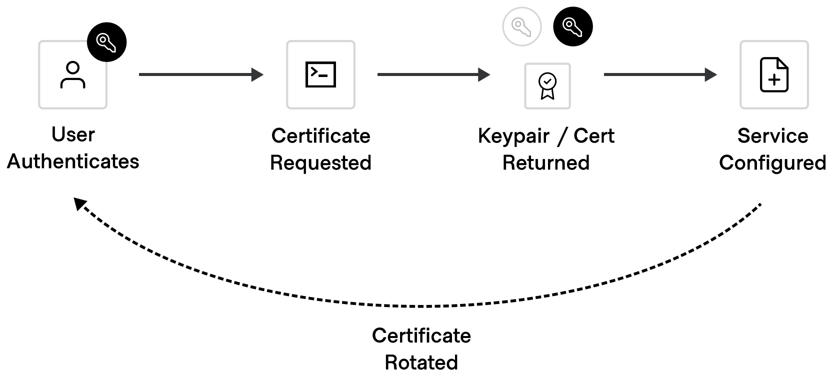
Improving Certificate Lifecycle and Security

With a traditional certificate request process, there are manual steps. A requestor will create a keypair and a certificate signing request (CSR), and will generally submit the CSR to a separate team to create the certificate. Based on industry data, that request takes approximately three days and ten hours on average to complete.



Traditional Certificate Request Process

With Vault, all actions in the process are instantaneous and can be handled through an API. This means that the process can be shortened significantly as well as automated from beginning to end, removing the need for human intervention and even preventing outages that occur due to expiring certificates.



Vault Certificate Request Process

Vault PKI Secrets Engine

The pki secrets engine generates dynamic X.509 certificates. With this secrets engine, services can request certificates without going through the usual manual process of generating a private key and CSR, submitting to a CA, and waiting for a verification and signing process to complete. Instead, Vault's built-in authentication and authorization mechanisms provide the necessary verification functionality. Additionally, by allowing for relatively short TTLs, revocations are less likely to be needed, keeping CRLs short and helping the secrets engine scale to large workloads. This combined functionality allows each instance of a running application to have a unique certificate, eliminating sharing and the accompanying pain of revocation and rollover.

Example Scenario

You need to provide self-signed certificates to multiple applications. There is a requirement for each application development team to be able to manage their own SSL certificates. Each development team has been assigned a DNS subdomain in your organization's internal DNS domain.

In this recommended pattern, the example organizational layout would look something like this:

- **root-admins**
  - Responsible for the central management of Vault, including management of the root PKI engine and the setup of namespaces
  - This group exists either as an identity group in Vault or a mapped LDAP group.
- **example.com** - The root-level domain for the installation.
- **Application-A**
  - An application that serves the **login.example.com** sub-domain
- **Application-B**
  - An application that serves the **subscribe.example.com** sub-domain
- **Application-C**
  - An application that serves the **news.example.com** sub-domain

- **App-Dev Team A**
    - Responsible for the development and integration of application-A
    - Namespace-admins are those who have elevated namespace admin rights
- **App-Dev Team B**
    - Responsible for the development and integration of application-B
    - Namespace-admins are those who have elevated namespace admin rights
- **App-Dev Team C**
    - Responsible for the development and integration of application-C
    - Namespace-admins are those who have elevated namespace admin rights

Designing the Vault PKI Solution

Planning the PKI deployment is the most critical piece of building the full solution. Proper and careful planning can help to ensure the solution is scalable and resilient for years to come.

Design Considerations

Before beginning the planning phase of the PKI solution, there are several considerations that should be made. These considerations are outlined in detail below.

Root CA

Vault storage is secure but not as secure as a piece of paper in a bank vault. It is, after all, networked software. If your root CA is hosted outside of Vault, don't put it in Vault as well; instead, issue a shorter-lived intermediate CA certificate and put this into Vault. This aligns with industry best practices.

Since 0.4, the secrets engine supports generating self-signed root CAs and creating and signing CSRs for intermediate CAs. In each instance, for security reasons, the private key can only be exported at generation time, and the ability to do so is part of the command path (so it can be put into ACL policies).

If you plan on using intermediate CAs with Vault, it is suggested that you let Vault create CSRs and do not export the private key, then sign those with your root CA (which may be a second mount of the pki secrets engine).

One CA, One Secrets Engine

In order to vastly simplify both the configuration and codebase of the pki secrets engine, only one CA certificate is allowed per secrets engine. If you want to issue certificates from multiple CAs, mount the pki secrets engine at multiple mount points with separate CA certificates in each.

This also provides a convenient method of switching to a new CA certificate while keeping CRLs valid from the old CA certificate; simply mount a new secrets engine and issue from there.

A typical pattern is to have one mount act as your root CA and to use this CA only to sign intermediate CA CSRs from other pki secrets engines.

Short Certificate Lifetimes

This secrets engine aligns with Vault's philosophy of short-lived secrets. As such, it is not expected that CRLs will grow large; the only place a private key is ever returned is to the requesting client (this secrets engine does not store generated private keys, except for CA certificates). In most cases, if the key is lost, the certificate can simply be ignored, as it will expire shortly.

If a certificate must truly be revoked, the normal Vault revocation function can be used; alternately, a root token can be used to revoke the certificate using the certificate's serial number. Any revocation action causes the CRL to be regenerated. When the CRL is regenerated, any expired certificates are removed from the CRL (and any revoked, expired certificates are removed from secrets engine storage).

This secrets engine does not support multiple CRL endpoints with sliding date windows; often, such mechanisms have the transition point a few days apart, but this gets into the expected realm of the actual certificate validity periods issued from this secrets engine. A good rule of thumb for this secrets engine would be to simply not issue certificates with a validity period greater than your maximum comfortable CRL lifetime. Alternately, you can control CRL caching behavior on the client to ensure that checks happen more often.

Often multiple endpoints are used to avoid an outage situation in the event where a single CRL endpoint is down so that clients aren't left without a response. Run Vault in HA mode, and the CRL endpoint should be available even if a particular node is down.

Configure CRL/OCSP in Advance

The `pki` secrets engine serves CRLs from a predictable location, but it is not possible for the secrets engine to know where it is running. Therefore, you must configure desired URLs for the issuing certificate, CRL distribution points, and OCSP servers manually using the config/URLs endpoint. It is supported to have more than one of each of these by passing in the multiple URLs as a comma-separated string parameter.

Safe Minimums

Since its inception, the pki secrets engine has enforced SHA256 for signature hashes rather than SHA1. As of 0.5.1, a minimum of 2048 bits for RSA keys is also enforced. Software that can handle SHA256 signatures should also be able to handle 2048-bit keys, and 1024-bit keys are considered unsafe and are disallowed in the Internet PKI.

Token Lifetimes and Revocation

When a token expires, it revokes all leases associated with it. This means that long-lived CA certs need correspondingly long-lived tokens, something that is easy to forget. Starting with 0.6, root and intermediate CA certs no longer have associated leases to prevent unintended revocation when not using a token with a long enough lifetime. To revoke these certificates, use the pki/revoke endpoint.

Patterns: Managing Root Certificates

What is a Root Certificate?

As the backbone of a PKI infrastructure, a root certificate authority (CA) sits on the apex of the certificates' trust hierarchy and is used to sign other certificates. A root certificate is a self-signed certificate generated by the root CA that a) follows the X.509 standards in defining the public key certificate formats and b) cryptographically signs intermediate and leaf certificates. This allows the capability to provide a multi-level hierarchy in a chain of trust in an authentication model between clients/applications to validate a trusted source for machine identity.

Vault PKI Root CA Anti-Patterns

External CA as a Public Signing Authority

Vault's PKI Engine aims to provide internal PKI certificate issuance only and is not meant for public web applications in securing TLS traffic for applications that are externally accessible. Hashicorp recommends leveraging publicly-trusted CAs (e.g., Geotrust, Verisign, Digitrust) to sign the application CSR to generate external-facing certificates. Vault's pluggable framework allows integration to these external CAs using a built in plugin like the Venafi Secrets Engine or writing your own using the custom plugin framework.

The benefit of having Vault broker certificates from external CAs is simplifying the DevOps process with a single workflow to:

1. Allow a variety of authentication through different methods
2. Enable ACL policy templates to authorize on permissible certificates to be generate
3. Utilize a uniform set of APIs for for automation
4. Conform to a unified logging and auditing source of the certificate lifecycle trail for troubleshooting and compliance

The Root CA as an Issuing CA

While the Vault root CA (`/pki/root/generate:type`) can also generate leaf certificates without an intermediate CA by setting the `allowed_domains=true` to its assigned role, this is not a recommended deployment as there's no layered protection on the root CA certificate in the event of a compromise. The recommendation is to separate the issuing CA to another Vault deployment or PKI path and only leverage the root CA PKI path to sign the intermediate CSRs.

When Vault should Manage the Root CA(s)

In a production environment, the recommendation is to use an external Root CA to sign the intermediate CA that Vault uses to generate certificates.

Hashicorp recommends using the `pki` secret engine's root CA in non-production environments such as development and staging that reflects as closely as possible your production CA setup. For production environments, Hashicorp recommends leveraging an organization's existing root/issuing CA that leverages additional validation (e.g., Active Directory Domain Services), policy management, and compliance management.

This is not to say that Vault is not used in production by organizations. On the contrary, there are instances where Vault is used as the Root CA in tandem with the HSM and Entropy augmentation that protects data both at rest and in-flight for compliance.

The `pki` engine's root CA runs in standalone mode where it does not require domain services and is mainly intended as a CSR signing authority to the pki Engine's intermediate CAs to increase automation in non-production environments. In most cases, requesting a CSR to be signed in production environments requires manual policy approvals that limit the velocity.

The initial motivation for the Vault `pki` Engine was the generation of short-duration client and server certificates for mutual authentication, rather than the generation of certificates for a larger purpose such as PKI infrastructure. After receiving multiple requests from the Vault community, the feature was created that allows an entity to generate an internal self-signed CA that allows:

1. A complete end to end automation in certificate generation and CSR Signing
2. Defining certificate constraints (e.g., Pathlength via BasicConstraints - which is the number of allowed CAs in the path) as well as stronger protection exposing the private keys.

The Vault Root PKI path acts as a standalone CA where it does not require any additional validations like domain membership (e.g., Active Directory Domain Services) as part of the PKI infrastructure management of certificates.

Lifecycle Management

The lifecycle of a root certificate goes through the following phases:

1. Preparation
2. Root Generation
3. Certificate Expiration, Renewal, and Revocation

Preparation

At a minimum, defining the Common Name (CN) is required. Optionally, gathering other Distinguished Name fields like Organization (O), Organization Unit (OU) is considered best practice to accurately reflect your registered organization information in the enrollment process. Please see recommendations below on typical configurations referring to key length and cipher.

Root Certificate Generation

In this phase, the creation of the root CA involves a three-step process within Vault:

```
—$ vault secrets enable pki

—$ vault secrets tune -max-lease-ttl=8760h pki

—$ vault write pki/root/generate/internal \

common_name=hashidemos.com \
ttl=8760h format=pem_bundle \
private_key_format=pem \
max_path_length=1 \
other_sans=”2.5.4.5;utf8:*” \
--format=json

Success! Enabled the pki secrets engine at: pki/
Success! Tuned the secrets engine at: pki/
{
    “request_id”: “aaa11df3-887c-b1cf-55ee-1652fca709f2”,
    “lease_id”: “”,
    “lease_duration”: 0,
    “renewable”: false,
    “data”: {
        “certificate”: “-----BEGIN CERTIFICATE-----\nMIIDRjCCAi6gAwIBAgIUYco/sI5e5DT/
ICLzYyoT8PqChz4wDQYJKoZIhvcNA…
YE+HWoYzUWadTiM1A8/Gdw9zSevBolqn\nvjIppcHrscebsCX4ZMdJtVtiqd8jNmoC+o4=\n
```

Figure 1-x

```
-----END CERTIFICATE-----",1
        "expiration": 1638389843,
        "issuing_ca":
"-----BEGIN CERTIFICATE-----\nMIIDRjCCAi6gAwIBAgIUYco/sI5e5DT/
ICLzYyoT8PqChz4wDQYJKoZIhvcNA..
YE+HWoYzUWadTiM1A8/Gdw9zSevBolqn\nvjIppcHrscebsCX4ZMdJtVtiqd8jNmoC+o4=\n
-----END CERTIFICATE-----",
        "serial_number":
    "61:ca:3f:b0:8e:5e:e4:34:ff:20:22:f3:63:2a:13:f0:fa:82:87:3e"
 },
"warnings": null
}
```

Another option is to create a self-signed certificate and private key outside of Vault (e.g., via OpenSSL, cfssl, etc.) and import this through Vault via the `pki/config/ca` path. This would require a JSON payload that concatenates both the cert and the private key. Please note that when parsing the certificate from the JSON output, the newline "`\n`" escape character needs to be parsed correctly to maintain the proper certificate format. For further details, please see the "Submit CA Information" section on Vault PKI API.

Certificate Expiration, Renewal, and Revocation

Vault automatically revokes the generated root at the end of its lease period (TTL), and the CA certificate signs its own Certificate Revocation List (CRL).

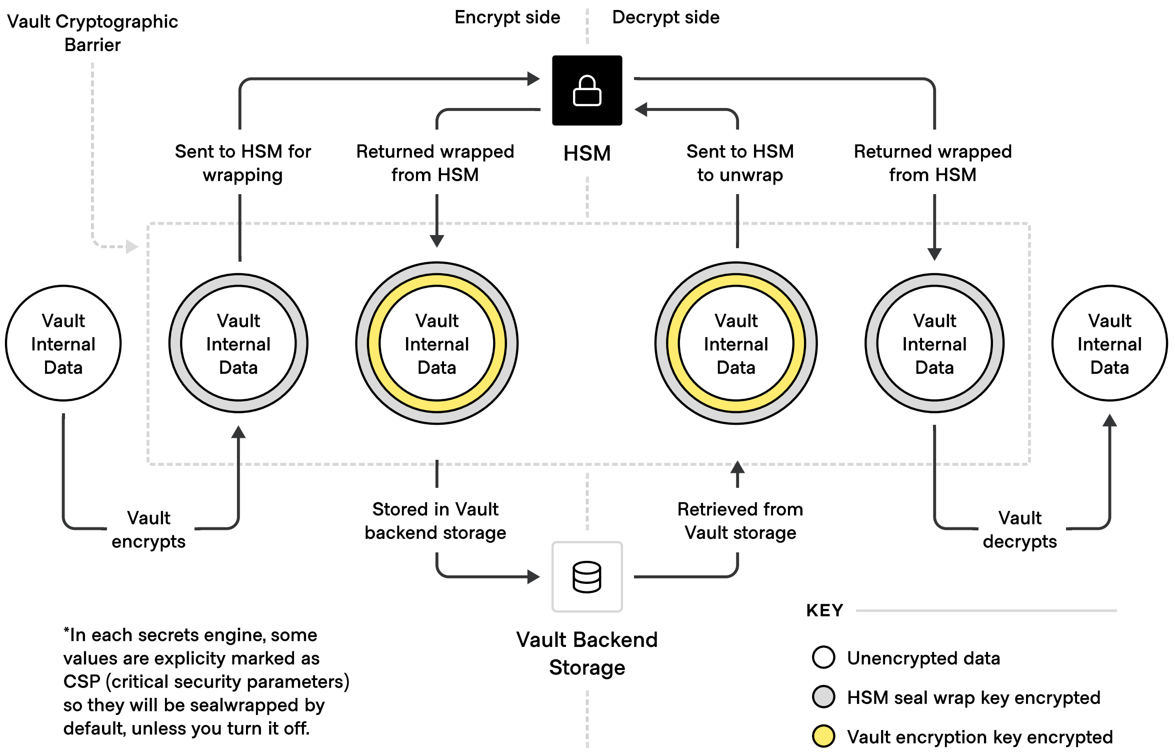
The root certificate can be renewed out-of-band by creating a self-signed certificate using the existing private key and submitting the newly created PEM bundle through the /pki/config/ca path to replace the existing one while maintaining the chain of trust.

Root CA certificates cannot be revoked and added to the CRL using the `/pki/revoke` path by virtue of being self-signed and the topmost trusted identity of the entire chain. In the event that a revocation event is required, replacing the CA information with a new CA bundle (via submit or through regeneration after deleting the root) is recommended.

Common Configurations

Securing the Vault CA using an HSM

Vault supports the concept of "Seal Wrap," which provides FIPS KeyStorage-conforming functionality for Critical Security Parameters such as the CA key. Seal wrapping facilitates the concept of envelope encryption, which simply provides another layer of encryption by the HSM on top of Vault's encryption barrier via PKCS#11 version 2.20+. For the root PKI mount enabled in Vault, both the root certificate and private key are re-encrypted and stored back into Vault's storage (e.g., Raft or Consul).


Vault Cryptographic Barrier
Encrypt side
Decrypt side
HSM
Sent to HSM for wrapping
Returned wrapped from HSM
Sent to HSM to unwrap
Returned wrapped from HSM
Vault Internal Data
Vault Internal Data
Vault Internal Data
Vault Internal Data
Vault Internal Data
Vault Internal Data
Vault encrypts
Stored in Vault backend storage
Retrieved from Vault storage
Vault decrypts
Vault Backend Storage
*In each secrets engine, some values are explicity marked as CSP (critical security parameters) so they will be sealwrapped by default, unless you turn it off.
KEY
Unencrypted data
HSM seal wrap key encrypted
Vault encryption key encrypted

Securing Vault with an HSM

In addition, if the HSM in use is FIPS 140-2 compliant, Vault stores Critical Security Parameters (CSPs) for the root PKI mount in a manner that is compliant with the FIPS 140-2 guidance for both Key Storage (FIPS 140-2 IG 7.16) and Key Transport (FIPS 140-2 IG D.9). For further details on the guidance for Key Storage and Key Transport, please refer to the NIST Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program guide.

Securing the Root Private Key Generation

Generating the root certificate and private key after the creation of the PKI mount provides the administrator two options: using either the exported type, which returns the private key in the response, or using the internal type, which doesn't return the private key in the response or later in response to CLI or API commands. Due to the highly sensitive nature of the cryptographic material and the impact of the root private key, it is highly recommended to set the generation to internal and leverage the HSM integration for seal wrapping.

As an example, Integrating an HSM such as Thales with Vault requires the following modification to Vault's configuration file (config.hcl):

```
# PKCS11 seal
seal “pkcs11” {
    lib = “<path to cryptoki library>”
    slot = “<slot number>”
    pin = “<partition password>”
    key_label = “HashiCorp”
    hmac_key_label = “HashiCorp_hmac”
    generate_key = “true”
}
```

Key Length and Algorithm Selection

The root generation endpoint (/pki/root/generate/:type) allows the capability to select a desired cipher (RSA or ECDSA) with a corresponding key size.   Per NIST recommendations for a CA asymmetric key pair used to sign and verify certificates, the recommended key length and cipher are as follows:

- When using RSA, recommended minimum key length are 2048 or 3072bits

```
vault write -field=certificate pki/root/generate/internal \
common_name="hashidemos.io" \
ttl=87600h \
key_type=rsa \
key_bits=3072
```

- When using ECDSA, recommended minimum key lengths are Curves P-256 or P-384

```
vault write -field=certificate pki/root/generate/internal \
common_name="hashidemos.io" \
ttl=87600h \
key_type=ec \
key_bits=384
```

Hashicorp recommends ECDSA cipher with the key length specified above as being a newer algorithm (2005); it offers the advantage of better key complexity (useful against brute force attacks) and performance as the key length is much smaller than that of RSA to offer the same protection. For example, to provide 128-bit security, RSA needs a key length of 3072 while ECC provides the same protection with 384 bits.

Root Certificate Validity Period

A root certificate's validity period can be defined either through the PKI engine mount (/pki) or through the root certificate generation endpoint:

Via secrets mount endpoint (sys/mount):

```
vault secrets tune -max-lease-ttl=8760h pki
```

Via the PKI Engine endpoint (/pki/root/generate):

```
vault write -field=certificate pki/root/generate/internal \
    common_name="hashidemos.io" \
    ttl=87600h \
```

While Hashicorp recommends short-lived leaf certificates, root certificates generally have a longer validity period since they are the top-most dependency in the verification of the chain. The primary consideration for either a rekey or rotation of the certificate depends on an organization's sensitivity to the risk for potential malicious activity upon exposure of the CA's corresponding private key. Vault can mitigate the risks of longer-lived CA keys with the following safeguards:

1. Non-exposure of the private key upon generation by using the internal type.
2. Additional layered encryption with an HSM.
3. A defined policy to rotate both the root certificate and its private key upon expiration.
4. Provide additional gating capabilities for multi-approval access to the mount using Control Groups, simulating an access pattern of an offline certificate authority.

Defining Root CA Constraints

Per best practices, CA constraints are typically delegated to the sub-CAs (e.g., policy CA). Basic Constraints, CA Path Length, and Name Constraints are discussed in detail in section 3.2.3 for "Managing Intermediate Certificates."

Enabling Entropy Augmentation

Entropy augmentation allows Vault Enterprise to supplement its system entropy with entropy from an external cryptography module. By definition, entropy is a measure of randomness for generating cryptographic material in a system. By default, Vault sources its randomness via the underlying operating system (/dev/urandom) via the GoLang crypto libraries and uses a cryptographically secure random number generator. Hosts have a low entropy pool for a source of randomness and weaknesses, such as utilizing pseudo-random generation that can potentially allow a compromise.   With an external source such as an HSM, we can now take advantage of a stronger variability with a much larger pool of random data to leverage true random number generation when securing Vault crypto barrier as a whole in the generation of keys such as Vault's master key, root tokens, and auto-unseal. For more information, please refer to the list of Critical Security Parameters protected by this feature.

To enable entropy augmentation, the following conditions must be met:

1. The Vault Enterprise HSM binary is installed since both Seal wrapping (HSM) and Entropy Augmentation are enterprise features

2. An existing HSM and Seal Configuration in Vault's server configuration

3. Enable the feature in the server configuration:

```
entropy “seal” {
    mode = “augmentation”
}
```

4. Upon enabling the Root CA PKI mount, the `-seal-wrap` and `-external-entropy-access` flags must both be set:

```
vault secrets enable \
    -max-lease-ttl=8760h \
    -seal-wrap \
    -external-entropy-access \
    pki
```

Monitoring Recommendations

We recommend monitoring Vault's audit log and alerting on the following key events. Set Signed and Generate Intermediate operations should only happen once during the typical CA lifecycle; other operations are infrequent admin actions.

Operation	**Example path and data snippets**
Generate Root: This generates a new public/private key pair for the CA.	`“path”: “root_ca_v1/root/generate/internal”`

Create/Update Role: This allows creating new endpoints to sign or generate certificates. Some of the attributes could be unmasked to ensure they are expected.	`“path”: “pki/roles/example-dot-com”, “data”: { “allow_subdomains”: “hmac-sha256:2a4d34b8b4cae8625b75f4d38fa35e829d149649d22429cda5addbc204db5caf”, “allowed_domains”: “hmac-sha256:aeeb6c1d36b621d3d7a5302fb1f8a417edf7c788f88298fd2d9da1c19424e921”, “allowed_domains_template”: “hmac-sha256:2a4d34b8b4cae8625b75f4d38fa35e829d149649d22429cda5addbc204db5caf”, “allowed_other_sans”: “hmac-sha256:09408da5dc714fb83ac514dfd1aa618098fb5771832601fbeba6277de461b814”, “max_ttl”: “hmac-sha256:ec4ad65694ac84acd1aa53035dc421186f6007899808764cef1354193a6174f5”, “policy_identifiers”: “hmac-sha256:0e9e0148ce60b452956970c94f61886df36f3de892b72abd64f6b52291ac1deb”`
Set URLs: Configure URLs that are encoded in leaf certificates.	`“path”: “pki/config/urls”, “data”: { “crl_distribution_points”: “hmac-sha256:47d4121e9ea32ebcc66f33243182a5956f76b979a43648ca4cfc20f1d342e41a”, “issuing_certificates”: “hmac-sha256:70288776790b039dfbf97c368dbe85fe52004649804a957c85f631c47c85fd42” }`

Issue:

Generate an intermediate or leaf certificate

```
"path": "pki/issue/example-dot-com",
    "data": {
      "--format": "hmac-sha256:c1ad2434b3c0df12345d24bd12c5b03
8bf843f9dc834cd445ec608c0e17d39ce",
      "common_name": "hmac-sha256:e7d674718d78d627e25998c41495
c0334d8dfbbd7a8fb4a29e342cee8eacbf84"
}

"response": {
    "mount_type": "pki",
    "data": {
      "certificate": "hmac-sha256:5188f4837cf3a7063b026927e3850
5f85dec25516fc5a2bec51e78b5a54f60ad",
      "expiration": 1610658846,
      "issuing_ca": "hmac-sha256:365049ef56fe1a7d473524abdab70
8b7c8442027a18197f64f8b8558749183a2",
      "private_key": "hmac-sha256:e8824ef5e13c93e7abf91ec43b97
bf53ab74a938a6276c21db0152eb3aec8d7d",
      "private_key_type": "hmac-sha256:0df8353207683e97ad685a6
ed1339bfdb328aa3ee1eed77f52c522e7fcef2983",
      "serial_number": "hmac-sha256:261d3a7b00c0e68ac0d77e877f
7391af4c704b56e4887445a92e7806299684f9"
    }
  }
```

Other CA-related events not listed here, such as Generate Certificates, should be logged, but an alert does not need to be generated every time. Below is an example Audit snippet from Intermediate set-signed:

```
{
  “time”: “2021-01-11T21:09:52.86336Z”,
  “type”: “response”,
  “auth”: {
    “client_token”:
“hmac-sha256:4021fb2a3d2d9f58d262182a4fabff76a187ba2923cdc3d626a52c4df3cfa2d1”,
    “accessor”:
“hmac-sha256:dffafb11324d0ad5db2ac8b732f8a27caa63abc62b58cac8fce8446cbcd9412f”,
    “display_name”: “token”,
    “policies”: [
      “root”
    ],
    “token_policies”: [
      “root”
    ],
    “token_type”: “service”,
    “token_issue_time”: “2021-01-11T16:07:57-05:00”
  },
  “request”: {
    “id”: “9775cf64-bf2b-91dd-e503-b357c679ffbc”,
    “operation”: “update”,
    “mount_type”: “pki”,
    “client_token”:
“hmac-sha256:4021fb2a3d2d9f58d262182a4fabff76a187ba2923cdc3d626a52c4df3cfa2d1”,
    “client_token_accessor”:
“hmac-sha256:dffafb11324d0ad5db2ac8b732f8a27caa63abc62b58cac8fce8446cbcd9412f”,
    “namespace”: {
      “id”: “root”
    },
    “path”: “pki/root/generate/internal”,
    “data”: {
      “--format”:
“hmac-sha256:c1ad2434b3c0df12345d24bd12c5b038bf843f9dc834cd445ec608c0e17d39ce”,
      “common_name”:
“hmac-sha256:5cf75da92d5529865c8b7549d0d3a7d41eff35ca9a4776aa9b4a2b7c6dc907fc”,
      “format”:
“hmac-sha256:c3d1265d9907e9633f7853c3d14d7951890728addb1eae7346027409db43950f”,
      “max_path_length”:
“hmac-sha256:e6e896a7fab41914aac7cac0e79d096d49bba53f80f308e46b6d581cf0463fbe”,
      “other_sans”:
“hmac-sha256:09408da5dc714fb83ac514dfd1aa618098fb5771832601fbeba6277de461b814”,
      “private_key_format”:
“hmac-sha256:47a63f2ed3ec3ef88ce95571f9bd4f7967f1d2cdac5e514ab303cebae150018f”,
      “ttl”:
“hmac-sha256:3da16e92e062617194ac5a28e2321e6cd48691869292f5e9761f8c6155fddd10”
    },
    “remote_address”: “127.0.0.1”
  },
  “response”: {
    “mount_type”: “pki”,
    “data”: {
      “certificate”:
“hmac-sha256:365049ef56fe1a7d473524abdab708b7c8442027a18197f64f8b8558749183a2”,
      “expiration”: 1641935392,
      “issuing_ca”:
“hmac-sha256:365049ef56fe1a7d473524abdab708b7c8442027a18197f64f8b8558749183a2”,
      “serial_number”:
“hmac-sha256:eef7f137329081f5657237554f3104d67f24cfa516e60c2b3581499719581c45”
    }
  }
}
```

Patterns: Managing Intermediate CA

An Intermediate CA is an authority that has its certificate signed by a parent CA. It follows the X.509 standards in defining the public key certificate formats and cryptographically signs other sub-CA certificates, and generates leaf certificates. When used for the latter, an Intermediate CA is also an Issuing CA. This section covers patterns for managing Intermediate CAs with the Vault PKI Secrets Engine.

CA Hierarchy and Recommendations

How many levels of intermediates are necessary?

- Things to consider to make this decision if not a prescriptive opinion on layer numbers.
- Topology and Subordinate CA Roles in Vault.
- Cross-signing for two intermediates, failover for the root of trust, and related topics.
- Root CA as the issuing CA.

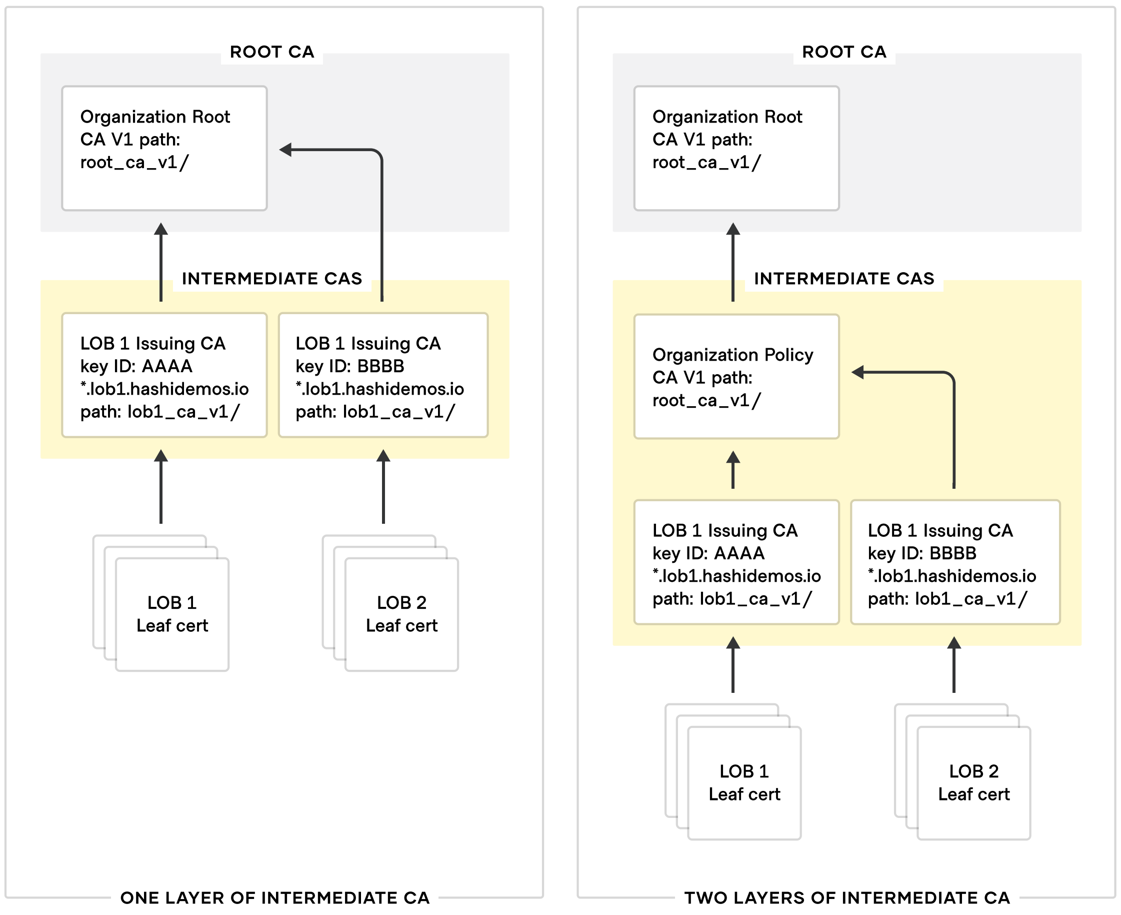
A CA Hierarchy entails having a Root CA and one or more levels of Intermediate CAs. We review some of these design patterns below, starting with all CAs being inside Vault and also integrating with existing CAs outside Vault.

1. **Two CA Levels:** For simple use-cases, having the root CA and one level of Intermediate CA is sufficient. In this case, the Intermediate CA also becomes an issuing CA since it can generate leaf certificates. Generally, there are multiple Issuing CAs - each corresponding to a project, LOB, or other defined purpose.
2. **Three CA Levels:** For large organizations, two CA levels can be challenging to scale since each Issuing CA must be configured correctly to enforce organizational security policies. Adding another layer of Intermediate CA allows for better flexibility.

   With this approach, one or more intermediate policy CAs can be introduced to allow central security teams to configure X.509 policies such as Basic Constraints, Key Usage, and Extended Key Usage. Another important configuration is the Path Length which enforces the CA hierarchy depth of valid certificate paths. We recommend that security policies be established for CA Path Length and enforced via CA configuration.

We examine the above options further in the section Common Intermediate Certificate Management Patterns. Issuing CA management can be delegated to LOB Admins.

These approaches are shown in the example below; each CA is a new instance of the PKI Secrets Engine mounted on a different path. As shown in the next section, often the Root CA, and sometimes the Intermediate Policy CA, reside outside of Vault in an existing PKI system.


ROOT CA
Organization Root CA V1 path: root_ca_v1/
INTERMEDIATE CAS
LOB 1 Issuing CA key ID: AAAA *.lob1.hashidemos.io path: lob1_ca_v1/
LOB 1 Issuing CA key ID: BBBB *.lob1.hashidemos.io path: lob1_ca_v1/
LOB 1 Leaf cert
LOB 2 Leaf cert
ONE LAYER OF INTERMEDIATE CA
ROOT CA
Organization Root CA V1 path: root_ca_v1/
INTERMEDIATE CAS
Organization Policy CA V1 path: root_ca_v1/
LOB 1 Issuing CA key ID: AAAA *.lob1.hashidemos.io path: lob1_ca_v1/
LOB 1 Issuing CA key ID: BBBB *.lob1.hashidemos.io path: lob1_ca_v1/
LOB 1 Leaf cert
LOB 2 Leaf cert
TWO LAYERS OF INTERMEDIATE CA

Two and Three-Level CA Hierarchy Examples

3. **Issuing from a Root CA:** In this design, there is a single self-signed CA which is both the Root and the Issuing CA. While we do not recommend this pattern for production use, Vault does support it. Similar to Intermediate CAs, one or more Roles can be defined under the Root CA for certificate issuance. This pattern may be used for simple use-cases that are limited in scope, such as a development environment for a single project.

   As a best practice, Root and Issuing CAs should have different security policies which cannot be implemented in this design. Unlike a tiered approach where Issuing CAs are limited to administrative boundaries, the blast radius of a private key compromise event with this pattern is greater. Therefore when promoting to UAT, we recommend using one of the tiered approaches discussed earlier.

Other designs may include additional CA levels to allow for further separation of security controls and administrative tasks. Note that each additional level of CA will introduce more administrative complexity.

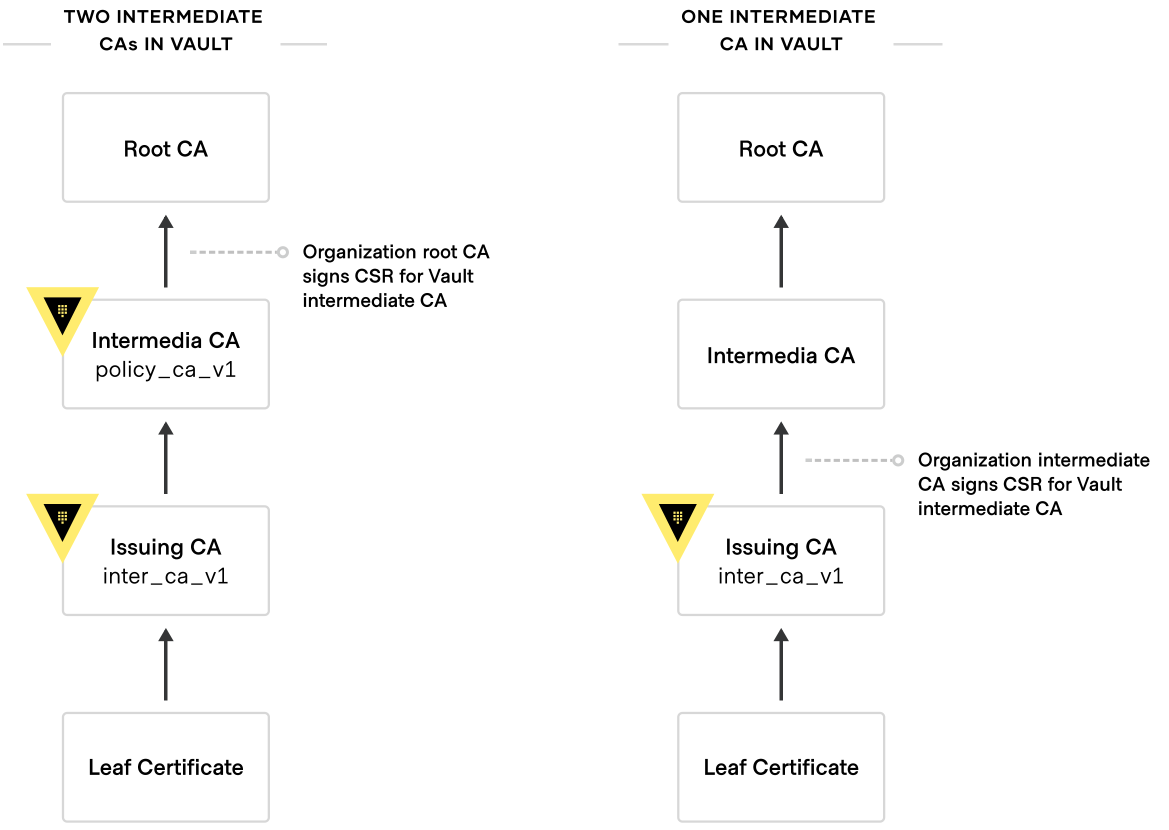
Integrating with Existing CAs

Some use-cases require that the root of trust is anchored within an existing company Root CA outside of Vault. For example, often the Root CA has key protection controls that require the CA and its key pair to be offline. Vault Intermediate CAs can be integrated with existing company Root or Intermediate CAs.

- One benefit of this design pattern is that the organization Root and Intermediate CA certificates are likely to be present in application truststores already. Only the Vault Intermediate CA needs to be added to complete the chain of trust. Clients will benefit from rapid certificate issuance using Vault's API and Authentication framework.
- If integrating with company CAs, there may be additional security requirements placed on your overall Vault deployment.

Generally there are two patterns for integrating with existing CAs depending on where in the trust chain this integration happens: the Root CA outside Vault, or multiple CAs outside Vault. These are shown in the diagram below.

External CA Intergration Workflow


TWO INTERMEDIATE CAs IN VAULT
Root CA
Organization root CA signs CSR for Vault intermediate CA
Intermedia CA policy_ca_v1
Issuing CA inter_ca_v1
Leaf Certificate
ONE INTERMEDIATE CA IN VAULT
Root CA
Intermedia CA
Organization intermediate CA signs CSR for Vault intermediate CA
Issuing CA inter_ca_v1
Leaf Certificate

1a. If creating a new CA in Vault: Use the Generate Intermediate endpoint to create the RSA key pair and the CSR. In the snippet below, we have specified the key type to be internal; therefore, the private key is not returned.

```
### Mount Secret Engine, generate Keys and CSR
vault secrets enable -path=inter_ca_v1 pki
vault write -format=json inter_ca_v1/intermediate/generate/internal \
  common_name=dev.hashidemos.io > inter_ca_v1.json

cat inter_ca_v1.json | jq -r .data.csr > inter_ca_v1-csr.pem
```

1b. Or, if using an existing Vault CA: Use the previously generated CSR from the Generate Internal API call, or use `openssl` to generate a new CSR. The example below shows how to generate a new CSR using `openssl` and the private key.

```
openssl req -new -key inter_ca_v1-privatekey.pem -out inter_ca_v1-csr.pem
```

2. Sign the CSR using the existing parent CA. These steps vary depending on the existing PKI in use and are out of scope for this whitepaper.

3. Use the Set Signed Intermediate endpoint [link] to associate the CA certificate with Vault CA. We are assuming that the signed certificate was saved to the file: inter_ca_v1.cert.pem.

```
### Use set Signed Intermediate to associate the CSR
vault write inter_ca_v1/intermediate/set-signed   certificate=@./inter_ca_v1.cert.pem
```

4. Display the CA certificate to validate all that all the fields are correct

```
curl -s "$VAULT_ADDR/v1/inter_ca_v1/ca/pem" > inter_ca_v1_ca.pem \
  && openssl x509 -in inter_ca_v1_ca.pem -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            21:81:69:e1:5c:c2:f9:5a:04:2f:66:b1:89:ad:2f:d4:8b:0a:75:62
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=hashidemos.io
        Validity
            Not Before: Dec 14 03:16:12 2020 GMT
            Not After : Dec 10 02:16:12 2021 GMT
        Subject: CN=dev.hashidemos.io
...
```

CA Hierarchy Design Recommendations

- Root CA usage: We recommend limiting the Root CA for only signing Intermediate CSRs. Following this recommendation minimizes the chance of the Root CA private key is exposed.
    - If using Vault as the Root CA, when possible, the CA type should be internal - this means that the private key is never revealed or exported. This setting also implies that during CA succession, the private key must be replaced with the new CA. Please see the CA Succession section for more details.
- **Security of the CA hierarchy:** A compromise within the CA chain can compromise the entire chain of trust. Vault provides multiple ways to ensure the security of the CA hierarchy:
    - Access to configure the Root and Intermediate CAs should be controlled via Vault ACL Policies.
    - The Root CA and first levels of Intermediate CAs may be placed in a Vault Namespace separate from Issuing CAs. This ensures a degree of isolation from the Issuing or LOB CAs used by applications.
    - Consider applying Vault Enterprise Control Groups for major lifecycle milestone operations such as Set Signed Intermediate, Generate Root, Generate Intermediate, and Sign Intermediate to mitigate potentially destructive operations.
- **Multiple Issuing CAs:** Issuing CAs should serve their own functional domain; there may therefore be many of them present within an Organization based on Administrative boundaries such as Lines of Business. This limits the blast radius of the affected chains of trust in case of a compromise or misconfiguration. Vault has a range of CA configuration options to customize each CA with a precise security policy. These are further examined in section Common Intermediate Cert Management Patterns.

Intermediate CA Anti-patterns

- **Short-lived CAs:** CA certificates are usually valid for one or more years. Using short-lived CA certificates that are valid for days or months will incur additional operational overhead. We recommend establishing organization policies on CA certificate lifetimes. Below are some example CA lifetimes from Let's Encrypt [link]:

    - Root ISRG Root X1 and ISRG Root X2 have validity periods of 20 years.
    - Intermediate issuing CAs RSA Let's Encrypt Authority X3, Let's Encrypt R3 and Let's Encrypt E1 all have validity periods of 5 years.

PKI use cases for internal communication may have shorter Issuing CA lifespans. For Issuing CAs, you would want to start issuing from the new CA at a duration before the previous expires. If the private key of the Issuing CA changes, then there is an additional burden of updating client truststores with new certificate chains.

Common Intermediate CA Management Patterns

Continuing from the patterns mentioned in earlier sections, the following configurations define how to enable Vault features that correspond to the X.509 specification to control how an intermediate CA functions when issuing additional sub-CAs or Leaf certificates.

- **Basic Constraints**
    - Basic Constraints is an X.509 extension in a certificate that defines whether the certificate itself is a certificate authority or a leaf/end entity.
    - For intermediate certificates, generating a certificate signing request (CSR) for the root CA automatically embeds a Basic Constraints with the value CA:true

```
X509v3 extensions:
    X509v3 Key Usage: critical
        Certificate Sign, CRL Sign
    X509v3 Basic Constraints: critical
        CA:TRUE, pathlen:1
```

    - If another CA is expected below the current intermediate CA in a multi-tier PKI configuration, the CSR signing process automatically assumes that the certificate should be used as a certificate authority and adds the Basic Constraints.
    - On the issuing CA for the leaf certificates, the certificates generated will not have an explicit Basic Constraints defined in the extension created through the role. In order to define this in the certificate, set the flag basic_constraints_valid_for_non_ca to true. This will allow validation for

the non-CA certificate to issue child certificates:

```
vault write pki_int/roles/hashidemo-dot-com \
    allowed_domains=hashidemos.com \
    allow_subdomains=true \
    max_ttl=4380h \
    basic_constraints_valid_for_non_ca=true
```

Expected Result:

```
X509v3 Basic Constraints: critical
    CA:FALSE
```

- Path Length, by definition, is the number of Certificate Authorities allowed in the chain of trust. In the PKI hierarchy, this is typically set on the first sub/intermediate CA that is signed by the root CA. The Path Length is not defined on the Root CA in more complex environments to allow flexibility.
- The parameter max_path_length indicates the CA depth encoded in the certificate; the default is -1, which means there is no limit. A path length of 2 means that a sub-CA can have a maximum of two CAs underneath it in the hierarchy, while a path length of 0 means that it can only issue leaf certificates.

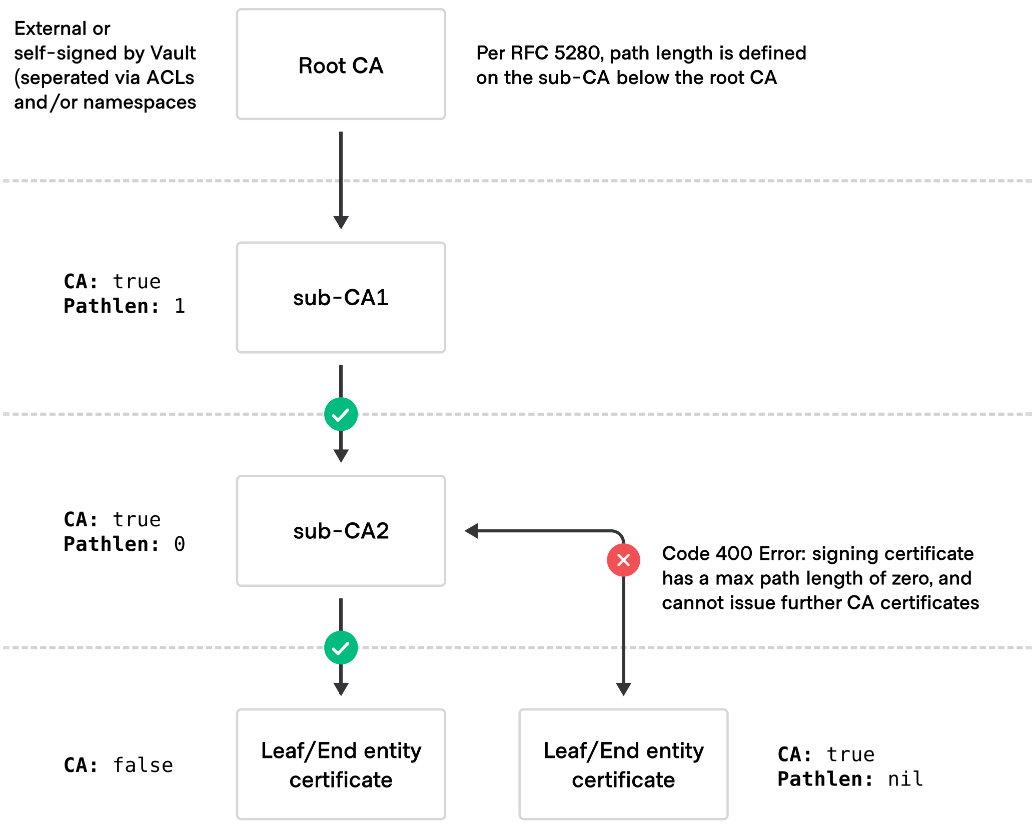
Encoded certificate:

```
X509v3 Basic Constraints: critical
CA:TRUE, pathlen:2
```

- If the CA has a path length of 0, a CSR request for this CA will result in an error:

```
URL: PUT http://localhost:8200/v1/pki_int_a/root/sign-intermediate
Code: 400. Errors:

* signing certificate has a max path length of zero, and cannot issue further
CA certificates
```

Example: basic constraint in sub-CAs



External or self-signed by Vault (seperated via ACLs and/or namespaces
Root CA
Per RFC 5280, path length is defined on the sub-CA below the root CA
CA: true Pathlen: 1
sub-CA1
CA: true Pathlen: 0
sub-CA2
Code 400 Error: signing certificate has a max path length of zero, and cannot issue further CA certificates
CA: false
Leaf/End entity certificate
Leaf/End entity certificate
CA: true Pathlen: nil

- **Key Usage and Extended Key Usage**
  - By definition, Key Usage defines the cryptographic operations that a certificate can do. Extended Key Usage allows the security team to define the certificate's purpose.
  - Vault configures Key Usage for "DigitalSignature", "KeyAgreement", and "KeyEncipherment" by default. Extended Key Usage defines both TLS Web Server and Client authentication
  - Alternatively, we can also specify the Extended Key Usage through its corresponding object identifier (OID). For example:

    ```
    Server authentication - OID 1.3.6.1.5.5.7.3.1
    Client Authentication - OID 1.3.6.1.5.5.7.3.1
    ```

  - Please refer to the Object Identifier Repository for further information
  - Key Usage and Extended Key Usage is defined in Vault through the corresponding PKI mount role name, e.g., `pki_int/roles/hashidemo-dot-com`
  - Key Usage in Vault
    - To define an extended key, the key_usage needs to be supplied as part of the parameter:

      ```
      vault write pki_int/roles/hashidemo-dot-com \
          allowed_domains=hashidemos.com \
          key_usage="DigitalSignature,KeyEncipherment
          "
      ```

      For reference on the list of Key Usage, please see the golang KeyUsage type.

  - Extended Key Usage in Vault
    - To define an extended key, the `ext_key_usage` or the `ext_key_usage_oids` needs to be supplied as part of the parameter:

      ```
      vault write pki_int/roles/hashidemo-dot-com \
      ```

```
    allowed_domains=hashidemos.com \
    ext_key_usage="OCSPSigning"
```

-or-

```
vault write pki_int/roles/hashidemo-dot-com \
    allowed_domains=hashidemos.com \
    ext_key_usage_oids="1.3.6.1.5.5.7.3.9"
```

For reference on the list of Extended Key Usage, please see the golang ExtKeyUsage type.

- TLS Web Server Authentication and TLS Web Server Client Authentication are enabled by default. In order to remove them from the certificate, the server_flag (TLS Web Server) and client_flag (TLS Client Server) needs to be disabled.

```
vault write pki_int/roles/hashidemo-dot-com \
    allowed_domains=hashidemos.com \
    server_flag=false \
    client_flag=false
```

- Additionally, we can use email_protection_flag and code_siging_flag to enable the Email Protection and Sign Executable Code extended keys, respectively.
- These flags are additive, which means we can selectively turn on the Extended Key Usages individually or all together to encode in the certificate.

- **Certificate Policy Identifier**
  - This section defines how to declare policies about PKI usage using Policy Identifiers.

```
vault write pki/roles/example-dot-com \
        allowed_domains="hashidemos.com" \
        max_ttl=72h \
        policy_identifiers="1.3.6.1.4.1.8072.2.1.1, 1.3.6.1.4.1.8072.2.1.2"
```

- Currently, there is no facility to include policy qualifiers, either the Certificate Practice Statement (CPS) Pointer or User Notice, as part of the policies extension.

- **Name Constraints**
  - Name Constraints policies provide restrictions in a PKI to control subject names in issued certificates. Vault PKI provides a simple implementation of Name Constraints to restrict the `DNS Name`:

```
X509v3 Name Constraints: critical
       Permitted:
              DNS:hashidemo.io
              DNS:vault.io
```

  - The primary purpose is to allow a sub-CA to issue a CSR with a constraint to allow it to restrict issued/signed certificates with domains/subdomains within its scope.
  - Name Constraints can be defined during generation of a root CA or through signing a CSR request using the `permitted_dns_domains` parameter:

```
vault write pki_int/root/sign-intermediate \
       csr=@pki_int_a_child_intermediate.csr \
       permitted_dns_domains=”hashidemos.io,vault.io” \
       format=pem_bundle ttl=43800h
```

  - `The permitted_dns_domains` parameter needs to be set on the issuing CA generating the certificate. In a multi-level PKI environment, this setting is not propagated to the lower sub-CAs.
  - If a certificate is issued outside the scope of the Name Constraint, the following error occurs:

```
vault write pki_int_child/issue/new_leaf \
       common_name=blah.nomad.com
```

```
error 47 at 0 depth lookup:permitted subtree violation
```

- More complex Name Constraints syntax and components (multi-level subtrees, other SANS extension, Processing rules) are currently out of scope.
- HashiCorp recommends setting Name Constraints when requesting a CSR to attest the validity of the scope of the CA. This setting can be layered with other controls to restrict domains/subdomains on the role via allowed_domains and allow_glob_domains when issuing end-entity certificates.

- **Subject Name and Subject Alternative Name extension**
  - We can define the SANs in the pki/role before issuing an X.509 certificate as alt_name (Hostnames and email addresses SANs), `ip_sans` or `uri_sans` (e.g., `spiffe//trust-domain/ns/test-namespace/svc/...`)
  - When defining issuing policies for the CA, the following attributes in the role provide a proactive approach to governing the scope of certificates issued. Note as well that each of the properties can be stacked when defined:
    - A CA administrator can specify a list of `allowed_domains` for the issuing role and create a wildcard certificate using `allow_subdomains` on the list. Enabling these flags determines the extent of the certificate's scope, either as a wildcard which allows multiple hosts to leverage the same certificate template or a SAN certificate, capable of being deployed across domains.
    - We can further refine the list of domains to use glob patterns such as "`web-*.hashidemos.io`".
    - Before restricting the certificates to the desired domain pattern, a permissive policy like `allow_any_name` can be used to test other parameters first (e.g., keyUsage). We can also use this parameter as a means of troubleshooting as needed.
    - Other controls like `enforce_hostnames` and `allowed_ip_sans` are enabled by default and can be turned off as needed for further refinement. Reversely, we can specify a value for the `allowed_uri_sans` to limit what's allowed in the `uri_sans`.

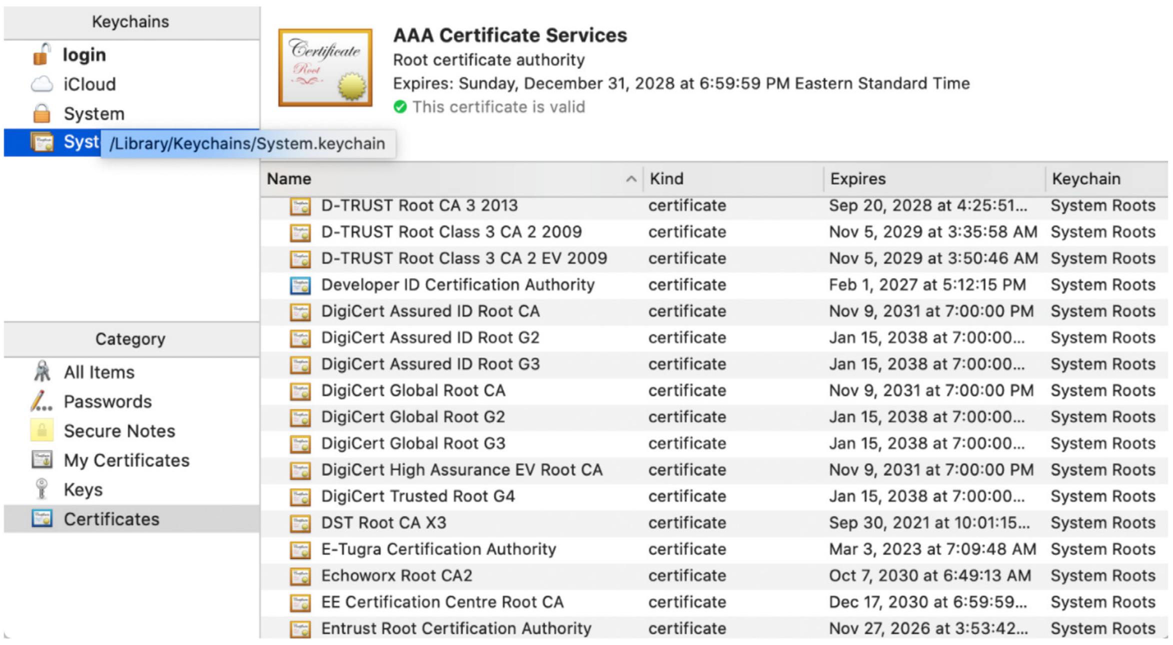
- In some scenarios, additional attributes or policies are needed to be satisfied in the certificate in order for an application that needs to be defined either in the Subject or SANs extension. Examples are:
    - To add an OID attribute for ElasticSearch Search Guard to determine valid trusted connections incoming nodes
    - Additional serial numbers
    - Additional email addresses

  Vault can accommodate these requirements by defining the respective OIDs of the desired attributes using the `allowed_other_sans` field:

```
# Example of adding additional OIDs for User Principal Name and Serial Number

vault write pki/roles/example-dot-com \
       allowed_domains=”hashidemos.com” \
       max_ttl=72h \
       allowed_other_sans-”1.3.6.1.4.1.311.20.2.3;utf8:1.2.3.4.5.5,
2.5.4.5;utf8:ff:ee:dd:cc:bb:aa:99:88:77:66:55:44:33:22:11:00”
```

  Hashicorp advises that you verify your application requirements and how it implements certificates as part of its overall workflow. The Search guard example above can be referenced under the “TLS for Production” section of the Search Guard documentation.

- **Inclusion of the Root Certificate in the CA Chain**
    - To verify the chain of trust, the verification process requires the entire certificate chain to the CA cert file. The public root certificate is not included in the chain since the best practice is to add it separately in the end entity’s `truststore`. For example, popular browsers like Chrome, Firefox, and Safari have root certificates installed by default in the System Roots keychain:



Keychains
- login
- iCloud
- System
- Syst /Library/Keychains/System.keychain

Category
- All Items
- Passwords
- Secure Notes
- My Certificates
- Keys
- Certificates

AAA Certificate Services
Root certificate authority
Expires: Sunday, December 31, 2028 at 6:59:59 PM Eastern Standard Time
This certificate is valid

Name	Kind	Expires	Keychain
D-TRUST Root CA 3 2013	certificate	Sep 20, 2028 at 4:25:51...	System Roots
D-TRUST Root Class 3 CA 2 2009	certificate	Nov 5, 2029 at 3:35:58 AM	System Roots
D-TRUST Root Class 3 CA 2 EV 2009	certificate	Nov 5, 2029 at 3:50:46 AM	System Roots
Developer ID Certification Authority	certificate	Feb 1, 2027 at 5:12:15 PM	System Roots
DigiCert Assured ID Root CA	certificate	Nov 9, 2031 at 7:00:00 PM	System Roots
DigiCert Assured ID Root G2	certificate	Jan 15, 2038 at 7:00:00...	System Roots
DigiCert Assured ID Root G3	certificate	Jan 15, 2038 at 7:00:00...	System Roots
DigiCert Global Root CA	certificate	Nov 9, 2031 at 7:00:00 PM	System Roots
DigiCert Global Root G2	certificate	Jan 15, 2038 at 7:00:00...	System Roots
DigiCert Global Root G3	certificate	Jan 15, 2038 at 7:00:00...	System Roots
DigiCert High Assurance EV Root CA	certificate	Nov 9, 2031 at 7:00:00 PM	System Roots
DigiCert Trusted Root G4	certificate	Jan 15, 2038 at 7:00:00...	System Roots
DST Root CA X3	certificate	Sep 30, 2021 at 10:01:15...	System Roots
E-Tugra Certification Authority	certificate	Mar 3, 2023 at 7:09:48 AM	System Roots
Echoworx Root CA2	certificate	Oct 7, 2030 at 6:49:13 AM	System Roots
EE Certification Centre Root CA	certificate	Dec 17, 2030 at 6:59:59...	System Roots
Entrust Root Certification Authority	certificate	Nov 27, 2026 at 3:53:42...	System Roots

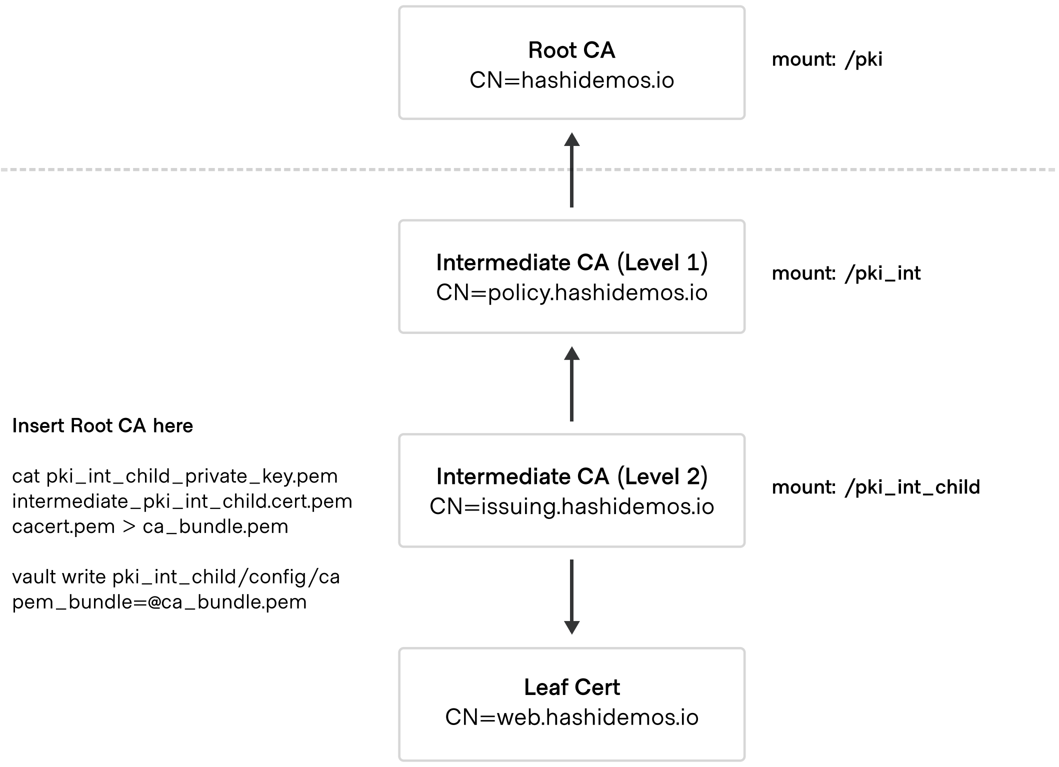
Example of Root CAs in an OS X System Root KeyChain (`truststore`)

- The CA cert file can be deployed through one of the following methods:
  - An out-of-band process upon the root cert generation using deployment (e.g., Terraform, Packer) and configuration management tools (e.g., Ansible, Puppet) and deploying this upon resource creation in the application truststore. An example would be to import the CA Cert to the Java Key Store using the "`keytool -import -keystore clientkeystore`" command.
  - Another option is to add the CA Cert as part of the ca_chain when issuing a certificate. By default, the CA Cert is omitted from the ca_chain of intermediate and leaf certificates. This property can be set using the following:

```
vault write pki_int/config/ca \
pem_bundle=@ca_bundle.pem
```

where the pem_bundle file contains the concatenated root certificate and private key.

Constructing a unified CA chain pem file



```
cat pki_int_child_private_key.pem
intermediate_pki_int_child.cert.pem
cacert.pem > ca_bundle.pem

vault write pki_int_child/config/ca
pem_bundle=@ca_bundle.pem
```

- The deployment of the CA Cert depends on the application deployment process and revocation methods. In the event that a root certificate is compromised in the first method, a separate deployment push of the CA cert to the trust store and delivery of the new certificate is required to establish the chain of trust. The second method provides a more streamlined deployment through a unified ca_chain (as shown in the diagram above) but may violate an institution's security requirements to export the policy/issuing CA private key to install in a pem_bundle in the intermediate CA configuration.

CA Lifecycle Management

Each Intermediate CA typically has the following lifecycle stages.

Action	Steps
1. Create CA by mounting the Vault PKI Secrets Engine	`vault secrets enable -path=inter_ca_v1 pki`

2. Establish a chain of trust	- Generate key pair and the CSR
	```
	`vault write -format=json inter_ca_v1/intermediate/generate/`
	`internal common_name=dev.hashidemos.io > inter_ca_v1.json`

	`cat inter_ca_v1.json | jq -r ‘.data.csr’ > inter_ca_v1.csr`
	```
	- Sign CSR by the parent CA. This action depends on whether the parent CA is in Vault or external (if Root CA, it signs itself)
	```
	`vault write -format=json \`
	`    root_ca_v1/root/sign-intermediate \`
	`    csr=@./inter_ca_v1.csr \`
	`    common_name=dev.hashidemos.io ttl=17520h | tee \`
	`>(jq -r .data.certificate > inter_ca_v1.pem) \`
	`>(jq -r .data.issuing_ca > issuing_ca.pem)`
	```
	
	continued...

	- Set the signed certificate for this CA ``` vault write inter_ca_v1/intermediate/set-signed certificate=@./inter_ca_v1.pem ``` - Distribute CA certificate to client trust stores
3. Configure the CA and create Roles to Issue certificates	- Create one or more Roles that can be used to issue certificates or sign a CSR. Security policies are applied by configuring the CA. Use the [Create/Update Role endpoint]() - Adjust TTLs for the CA. Please see the next section for more details on TTLs. ``` # Create the app1 Role vault write inter_ca_v1/roles/app1 \ allowed_domains="dev.hashidemos.io" \ allow_subdomains="true" \ max_ttl=24h # Adjust the default and max TTL for this role vault secrets tune \ -default-lease-ttl=24h \ -max-lease-ttl=48h inter_ca_v1/roles/app1 ```

4. Use CA to issue leaf certificates or sign certificates (CSR)	- Clients log in to Vault and use the Sign Certificate or Generate Certificate endpoints. The Generate Certificate endpoint encapsulates multiple steps: generating the public and private key pair, the CSR, and signing the certificate by the CA:
	
	\`\`\`
	`# Generate a new certificate`
	`vault write inter_ca_v1/issue/app1 \`
	`  common_name=app1.dev.hashidemos.io`
	\`\`\`
	
	We recommend using the Generate Certificate endpoint where possible as there are fewer steps for the client. In contrast, if the client has pre-created the key pair and the CSR, the Sign Certificate endpoint can be used to sign the CSR and generate the leaf certificate:
	
	\`\`\`
	`# Sign certificate with existing CSR`
	`vault write inter_ca_v1/sign/app1 \`
	`  csr=”$(cat app1-csr.pem)”`
	\`\`\`

5. CA Succession	- At approximately 50% of the current CA lifetime, we begin signing using a new CA certificate. To achieve this, we can either use a new CA or replace the certificate for the existing CA. - If using a new CA, we create and configure a new pki secrets engine and increment the version or generation number. \`\`\` `vault secrets enable -path=inter_ca_v2 pki` \`\`\` *continued...*

	- When using the existing CA, we replace the CA certificate using the Set Signed API call.
	```
	`vault write inter_ca_v1/intermediate/set-signed certificate=@./inter_ca_new_ttl.cert.pem`
	```
	
	- For more detailed steps and considerations for which type of strategy to use, please see the CA Succession section.
	
	- Follow steps 3 and 4 as before.

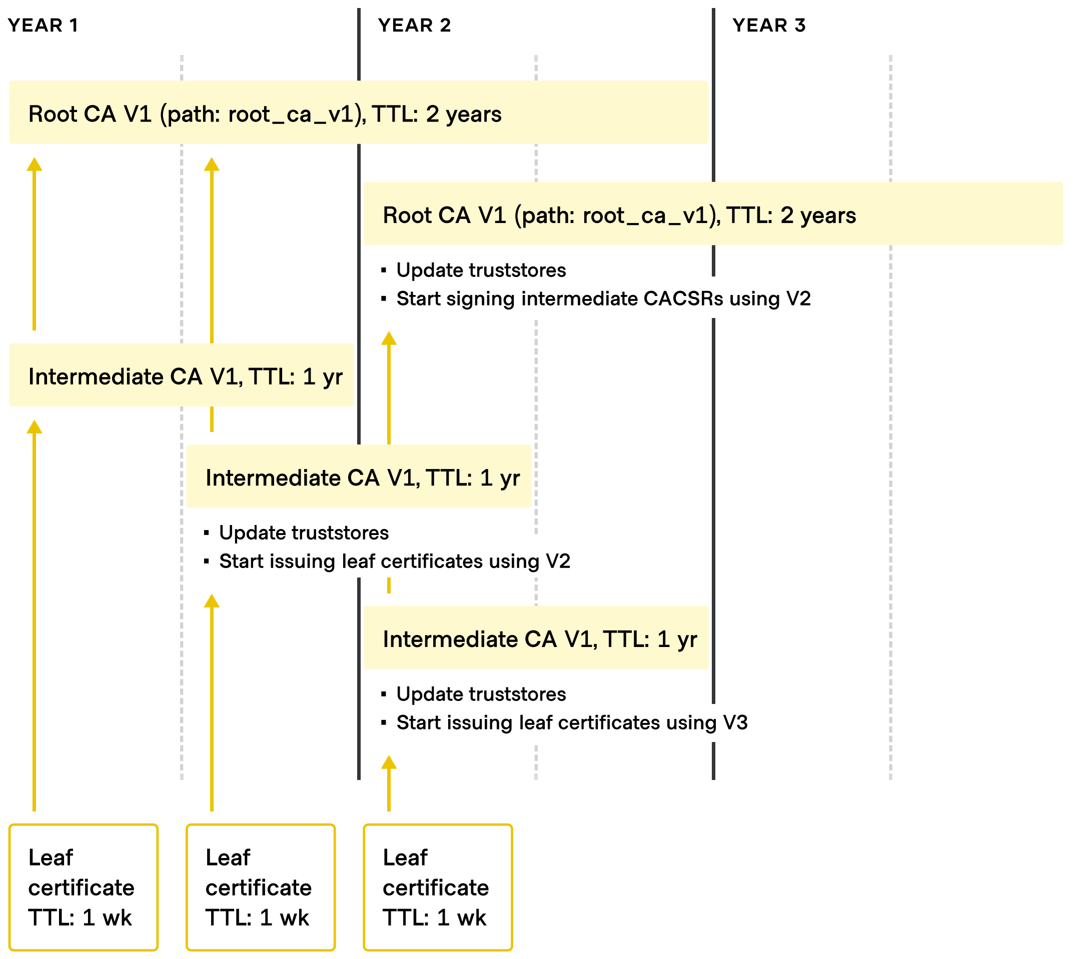
6. Start issuing or signing certificates from the new CA.	Same as step #4
7. Disable access to old CA	Once all application `truststores` have been updated with the next generation CA certificate, disable application access to the old CA using Vault ACLs.

CA Validity Time Periods

We recommend carefully planning the validity period for the entire CA hierarchy. Per RFC 5280, an X.509 certificate cannot be valid past that of its Signer/Issuer. Therefore, at each level of the CA hierarchy, the validity period is shorter than its parent.

We provide an example below of a two-tier CA hierarchy, where each tier has twice the lifetime of its child. The CA renewal workflow begins at approximately half the validity time period remaining. This method is a fairly common strategy, although it varies per organization.

Example CA and certificate validity periods


YEAR 1
YEAR 2
YEAR 3
Root CA V1 (path: root_ca_v1), TTL: 2 years
Root CA V1 (path: root_ca_v1), TTL: 2 years
▪ Update truststores
▪ Start signing intermediate CACSRs using V2
Intermediate CA V1, TTL: 1 yr
Intermediate CA V1, TTL: 1 yr
▪ Update truststores
▪ Start issuing leaf certificates using V2
Intermediate CA V1, TTL: 1 yr
▪ Update truststores
▪ Start issuing leaf certificates using V3
Leaf certificate TTL: 1 wk
Leaf certificate TTL: 1 wk
Leaf certificate TTL: 1 wk

When issuing certificates, Vault caps the `Not After` date of an issued certificate to the max-ttl setting of the Secrets Engine or the Role. ***We recommend explicitly setting the maximum TTL on each CA.*** Otherwise, Vault uses the system max TTL setting (default 32 days), which may result in unexpected Not After dates on certificates.

Using the example scheme in the diagram, we adjust the default and maximum TTLs as shown below. Note that only seconds and hours work as arguments for the TTL parameter.

```
# Root CA V1
vault secrets tune -default-lease-ttl=8760h -max-lease-ttl=17520h root_ca_v1

# Intermediate CA V1
vault secrets tune -default-lease-ttl=168h -max-lease-ttl=336h inter_ca_v1

# Intermediate CA V1 - Role
vault secrets tune -default-lease-ttl=168h -max-lease-ttl=336h inter_ca_v1/roles/app1
```

There are some scenarios where the effective Not After date can end up being past that of the signer. Vault handles these conditions as follows.

- If issuing a leaf certificate results in a `Not After` date later than the date associated with CA, an error is thrown. This outcome applies to the Generate Certificate and Sign Certificate endpoints.

```
Error writing data to inter_ca_v1/issue/app1: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/inter_ca_v1/issue/app1
Code: 400. Errors:

* cannot satisfy request, as TTL would result in notAfter 2021-01-12T17:35:24.828704-
05:00 that is beyond the expiration of the CA certificate at 2021-01-12T22:07:32Z
```

- When signing the CSR for Intermediate CA (Sign Intermediate endpoint), Vault allows the TTL to go past that of the Signing CA and issue a warning.

```
  “warnings": [
    “The expiration time for the signed certificate is after the CA’s expiration time.
If the new certificate is not treated as a root, validation paths with the certificate
past the issuing CA’s expiration time will fail.”
]
```

CA Succession

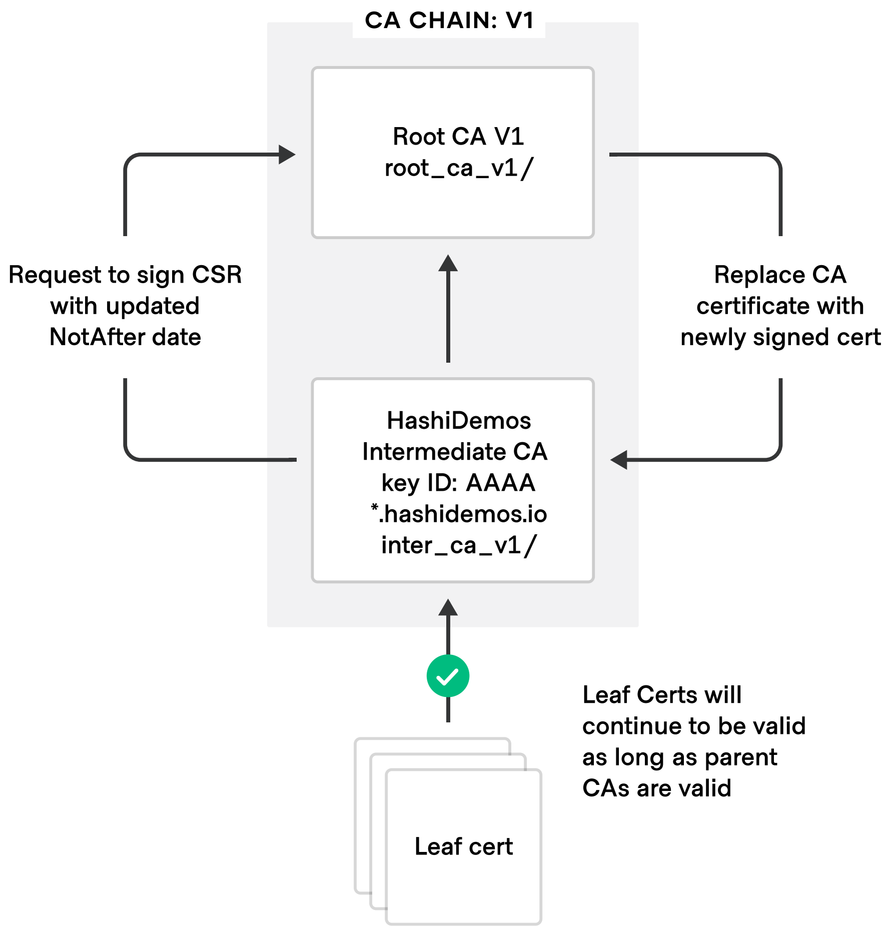
When the CA certificate expiration time is near, we need to replace it with a new certificate containing a later expiration date. We will review some common approaches to handling CA succession. One major decision is whether to also change the public/private key pair. We cover both scenarios below. If keeping the same key pair, then it is possible to simply update the same CA with a new certificate containing a later expiration date. We recommend planning for and testing CA succession well in advance of the actual CA certificate expiry date.

Please see a few considerations below:

1. We recommend implementing a versioning method for the CA name and mount path name in Vault, such as inter_ca_v1 or inter_ca_v2. This naming scheme provides predictable paths for Vault clients to request leaf certificates from the Vault server.
2. The examples below assume that the Intermediate CA is signed by a Root CA in Vault. Alternatively, if signing with an existing parent Intermediate CA, use the Sign Certificate API. Or, you may be signing with an existing external CA.
3. If the Intermediate CA type was internal when it was created, then the private key is inaccessible. In that case, the private key must be changed during CA succession.

1. Renew CA certificate TTL only

This step is the easiest in the CA succession strategy. We replace the CA certificate with a later TTL and continue to issue leaf certificates from the same CA. There is no need to mount a new Intermediate CA or update client truststores. A disadvantage is that the CA private key is not rotated, and the CA version number stays the same.


CA CHAIN: V1
Root CA V1
root_ca_v1/
Request to sign CSR with updated NotAfter date
Replace CA certificate with newly signed cert
HashiDemos Intermediate CA
key ID: AAAA
*.hashidemos.io
inter_ca_v1/
Leaf Certs will continue to be valid as long as parent CAs are valid
Leaf cert

Workflow Steps:

1. Locate the original CSR for Intermediate CA V1, or create a new CSR using openssl and the private key.

   ```
   openssl req -new -key inter_ca_v1-privatekey.pem -out inter_ca_v1-csr.pem
   ```

~~~
```
~~~

2. Sign the CSR with the existing Root CA using the Sign Intermediate API.

3. Use the Set Signed Intermediate API to associate the newly signed certificate with this CA.

~~~
```
### Sign CSR using Root CA
vault write -format=json root_ca_v1/root/sign-intermediate \
  csr=@./inter_ca_v1-csr.pem \
  common_name=dev.hashidemos.io \
  ttl=8760h \
  | jq -r '.data.certificate' > inter_ca_new_ttl.cert.pem

### Set Signed Intermediate
vault write inter_ca_v1/intermediate/set-signed certificate=@./inter_ca_new_ttl.
cert.pem

```
~~~

4. Verify that the CA TTL has been updated

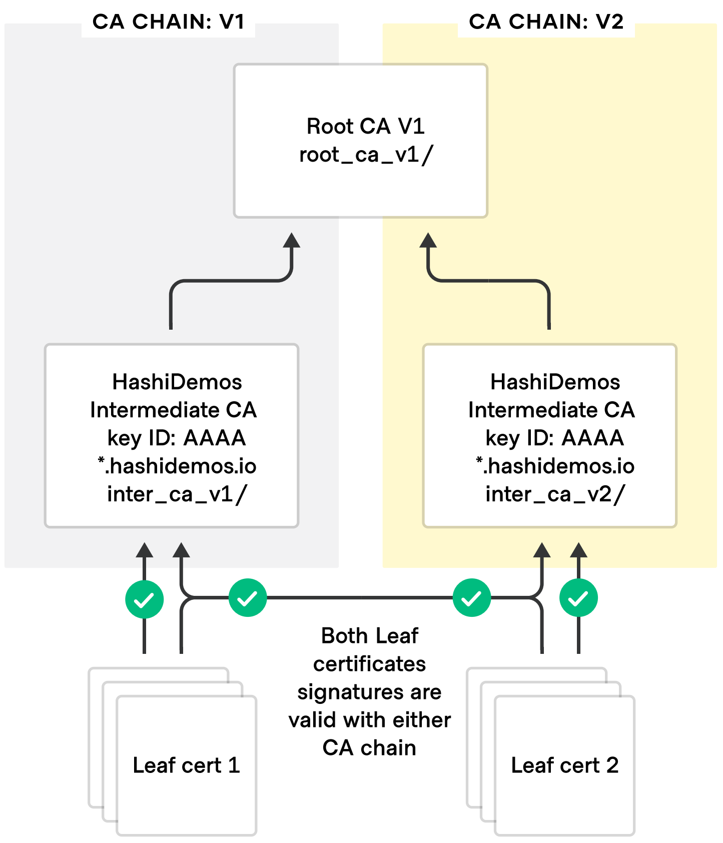
~~~
```
curl -s "$VAULT_ADDR/v1/inter_ca_v1/ca/pem" > inter_ca_v1_ca.pem \
  && openssl x509 -in inter_ca_v1_ca.pem -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            21:81:69:e1:5c:c2:f9:5a:04:2f:66:b1:89:ad:2f:d4:8b:0a:75:62
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=hashidemos.io
        Validity
~~~

```
            Not Before: Dec 14 03:16:12 2020 GMT
            Not After : Dec 10 02:16:12 2021 GMT
        Subject: CN=dev.hashidemos.io
...
```

5. Continue to issue leaf certificates from the same Intermediate CA

2. Replace CA without changing the private key

This action is an extension of the previous strategy. Here we create a new CA and cross-sign it with the existing Root CA. The public/private key pair for the CA is identical; therefore, clients can validate leaf certificates issued from either CA. This process is shown in the diagram below.



Workflow Steps:

1. Mount a new CA, incrementing the version number in its path.

```
vault secrets enable -path=inter_ca_v2 pki
```

2. Locate the original CSR from Intermediate CA V1, or create a new CSR using openssl and the private key.

   ```
   openssl req -new -key inter_ca_v1-privatekey.pem -out inter_ca_v1-csr.pem
   ```

3. Sign the CSR with the existing Root CA using the Sign Intermediate API.

4. Use Submit CA information to associate the private key and newly-generated certificate with this CA.

   ```
   ### Sign CSR
   vault write -format=json root_ca_v1/root/sign-intermediate \
     csr=@./inter_ca_v1-csr.pem \
     common_name=dev.hashidemos.io \
     ttl=8760h \
     | jq -r ‘.data.certificate’ > inter_ca_v2.cert.pem

   ### Concatenate the private key and CA certificate
   cat inter_ca_v1-privatekey.pem inter_ca_v2.cert.pem > inter_ca_v2.pem

   ### Submit CA information for CA V2
   vault write inter_ca_v2/config/ca pem_bundle=@inter_ca_v2.pem
   ```

5. Issue leaf certificates from the new CA going forward.

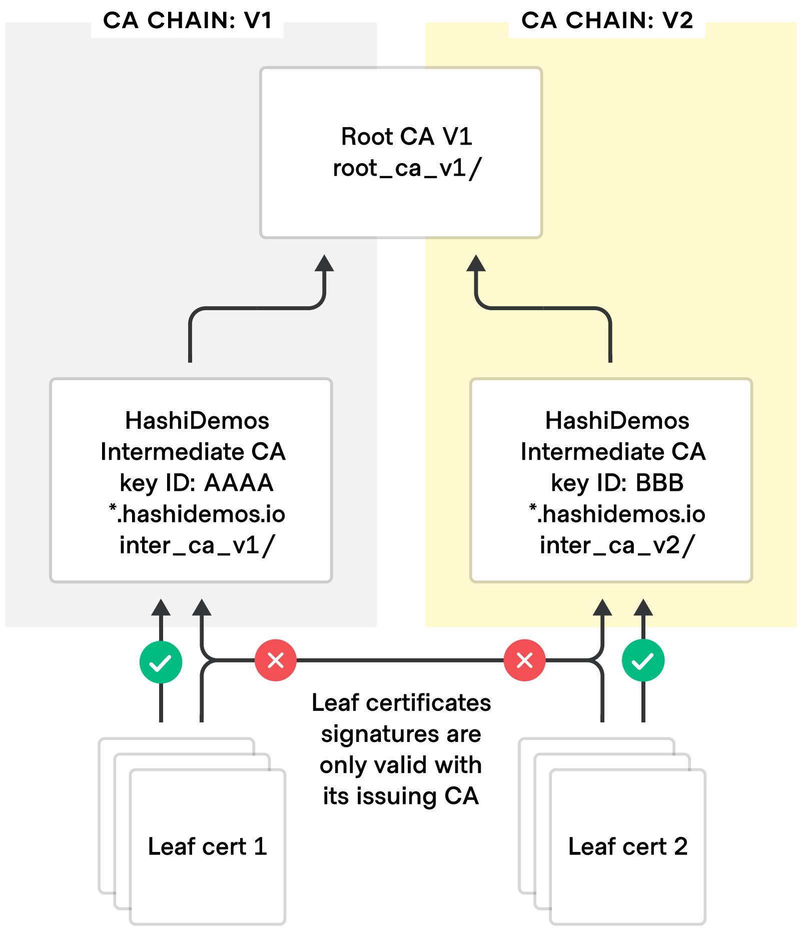
~~~~
```
### Create the App1 Role for CA V2
vault write inter_ca_v2/roles/app1 \
    allowed_domains=”dev.hashidemos.io” \
    allow_subdomains=”true” \
    max_ttl=24h

### Issue leaf cert from new CA
vault write -format=json inter_ca_v2/issue/app1 common_name=app1.dev.hashidemos.io
| jq -r ‘.data.certificate’ > leaf_certificate_v2.pem

### Validate leaf cert issued by Intermediate CA V2
openssl verify -verbose -CAfile root_ca_v1.pem \
  -untrusted inter_ca_v2.cert.pem  leaf_certificate_v2.pem

leaf_certificate_v2.pem: OK
```
~~~~

6. Update client truststores with the new CA chain containing the v2 CA certificate.

The advantage of this method is that new leaf certificates issued from v2 CA are usable immediately. Over time client truststores should be updated with the new Intermediate CA.

3. Replace CA and rotate the private/public key pair

In this scenario, the public/private key pair for the Intermediate CA has to be rotated, often due to a security requirement. Leaf certificates issued by the newer CA cannot be validated by the original issuing CA as shown in the diagram below.



Workflow Steps:

1. Mount a new Intermediate CA

```
vault secrets enable -path=inter_ca_v2 pki
```

2. Generate a new key pair and a CSR using Generate Intermediate.

3. Sign the CSR with the existing parent CA using the Sign Intermediate.

```
### Generate Keys and CSR
```

```
vault write -format=json inter_ca_v2/intermediate/generate/internal \
  common_name=dev.hashidemos.io > inter_ca_v2.json

### Sign CSR with existing Root CA
cat inter_ca_v2.json | jq -r .data.csr > inter_ca_v2.csr

vault write -format=json root_ca_v1/root/sign-intermediate \
  csr=@./inter_ca_v2.csr \
  common_name=dev.hashidemos.io \
  ttl=8760h \
  | jq -r ‘.data.certificate’ > inter_ca_v2.cert.pem

### Set Signed Intermediate for Intermediate CA V2
vault write inter_ca_v2/intermediate/set-signed certificate=@inter_ca_v2.cert.pem
```

4. Update client truststores with CA chain V2.
5. Issue leaf certificates from the new CA for clients with updated truststores (please see previous workflow steps for command snippets).

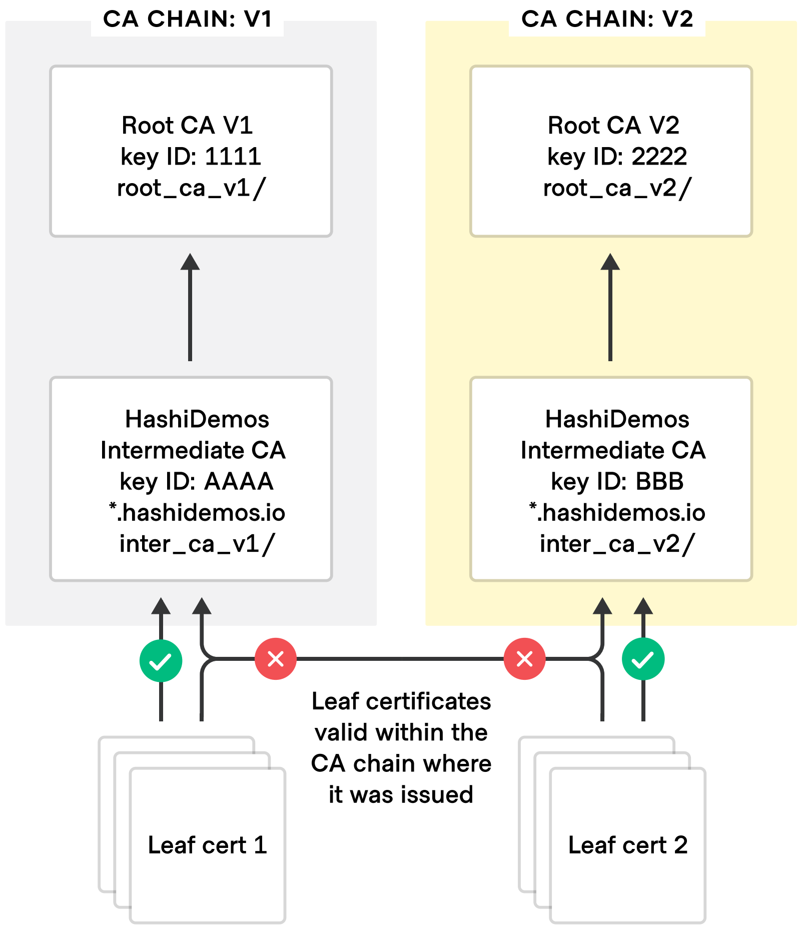
Leaf certificates issued from the new CA can only be used with TLS clients that have updated their truststores. Therefore there may be a period of overlap where the original CA continues to issue certificates until all truststores can be updated.

Updating truststores is a complex operation since there may be many applications to update, and some may need to be restarted for them to pick up the changes. Therefore these steps should be carefully planned in advance of CA rotation to minimize application downtime.

Ideally, these steps can be achieved in an automated manner with configuration management tools or through service configuration from Consul. Certain clients may be able to use the **Authority Information Access** section of the X.509 leaf certificate to download the new CAs automatically.

4. Replace Intermediate CA and Root CAs

The previous examples cover one layer of CA rotation. Depending on how CA TTLs align, multiple CAs may need to be rotated. We provide an example below of moving to a new Root CA and a new Intermediate CA. For simplicity, we keep the Intermediate private CA the same as before, similar to the first strategy. Below is a diagram showing this arrangement.



Workflow Steps:

This workflow combines steps from the previous two examples.

1. Mount a new Root CA, incrementing the version number. Generate the key pair and CA certificate using the Generate Root endpoint.

```
vault secrets enable -path=root_ca_v2 pki
vault write root_ca_v2/root/generate/internal \
    common_name=hashidemos.io \
    ttl=17520h format=pem_bundle \
    private_key_format=pem --format=json > root_ca_v2.json

cat root_ca_v2.json | jq -r ‘.data.certificate’ > cacert-v2.pem
```

2. Mount a new Intermediate CA

```
vault secrets enable -path=inter_ca_v2 pki
```

3. Locate the original CSR from Intermediate CA V1, or create a new CSR using openssl and the private key.

```
openssl req -new -key inter_ca_v1-privatekey.pem -out inter_ca_v1-csr.pem
```

4. Sign the CSR with the new Vault Root CA using the Sign Intermediate API. Use Submit CA Information to associate the private key and new certificate with this CA.

```
### Sign CSR with the new Vault Root CA
vault write -format=json root_ca_v2/root/sign-intermediate \
  csr=@./inter_ca_v1-csr.pem \
  common_name=dev.hashidemos.io \
  ttl=8760h \
  | jq -r ‘.data.certificate’ > inter_ca_v2.cert.pem

### Concatenate the private key and CA certificate
cat inter_ca_v1-privatekey.pem inter_ca_v2.cert.pem > inter_ca_v2.pem

### Submit CA information
vault write inter_ca_v2/config/ca pem_bundle=@inter_ca_v2.pem
```

5. Issue leaf certificates from the new CA going forward (please see previous workflow steps for command snippets).
6. Update client truststores with the new CA chain containing V2 CA certificate.

Similar to the previous example, application truststores must be updated before certificates issued from the new CA can be used. Updating the truststore entails adding both Root CA V2 and Intermediate CA V2.

Certificate Revocation List and OCSP

In this section, we review the CRL and OCSP related configuration of the CA. A more extended discussion on revocation is provided in the Leaf Certificate Revocation section.

Vault CAs publish a Certificate Revocation List (CRL) to allow for revocation checking by clients. Currently, Vault does not operate an OCSP responder; therefore, a separate OCSP server infrastructure must be deployed. This server can periodically query Vault for revoked certificates via CRLs/Vault APIs, or a push-based solution can be implemented.

By default, any Vault CA publishes a CRL at the Read CRL endpoint. To disable this behavior, the Set CRL Configuration endpoint can be used. Below is an example display of the CRL from an Intermediate CA using `openssl`. Note that the CRL is an unauthenticated endpoint (similar to viewing the CA certificate).

```
# Write the CRL to a file for CA inter_ca
curl -s \
    --header "X-Vault-Token: $VAULT_TOKEN" \
    "http://127.0.0.1:8200/v1/inter_ca/crl" \
    --output inter_ca.crl

# Display full CRL via openssl:
openssl crl -inform DER -text -in inter_ca.crl
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: /CN=hashidemos.io
        Last Update: Nov 30 20:41:35 2020 GMT
        Next Update: Dec  3 20:41:35 2020 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:5C:10:6C:2C:1D:F5:D7:05:CB:EA:E6:C8:EE:1F:3F:A5:69:3F:81:DE

Revoked Certificates:
    Serial Number: 0B5167CE491380C09D59ED16D656BD2C0F16F0E7
        Revocation Date: Nov 27 15:10:53 2020 GMT
    Serial Number: 508DAAFFEAB939DC38E96FA3693E75DF7BE9BF78
        Revocation Date: Nov 30 20:41:35 2020 GMT
    Serial Number: 5FAEEC765A225B8ECB4FE73BBB88D08D591568F0
        Revocation Date: Nov 28 18:45:26 2020 GMT
<truncated>
```

It’s important to note that each Vault CA maintains its own CRL, and the CRL is not replicated across Vault Performance Secondary clusters. Therefore, we recommend encoding the CRL endpoint(s) into X.509 certificates via the Set URLs endpoint.

Below is an example of configuring both the CRL and OCSP endpoints. Each URL field is optional, so you only need to set whichever ones are relevant for your organization.

```
vault write inter_ca/config/urls \
       issuing_certificates=”https://vault.example.org:8200/v1/inter_ca_v1/ca” \
       crl_distribution_points=”https://vault.example.org:8200/v1/inter_ca_v1/crl” \
       ocsp_servers=”https://ocsp.example.org”
```

The above information will be encoded in X.609 leaf certificates that are issued by the CA. If needed, multiple URLs can be encoded.

Revoking CA Certificates

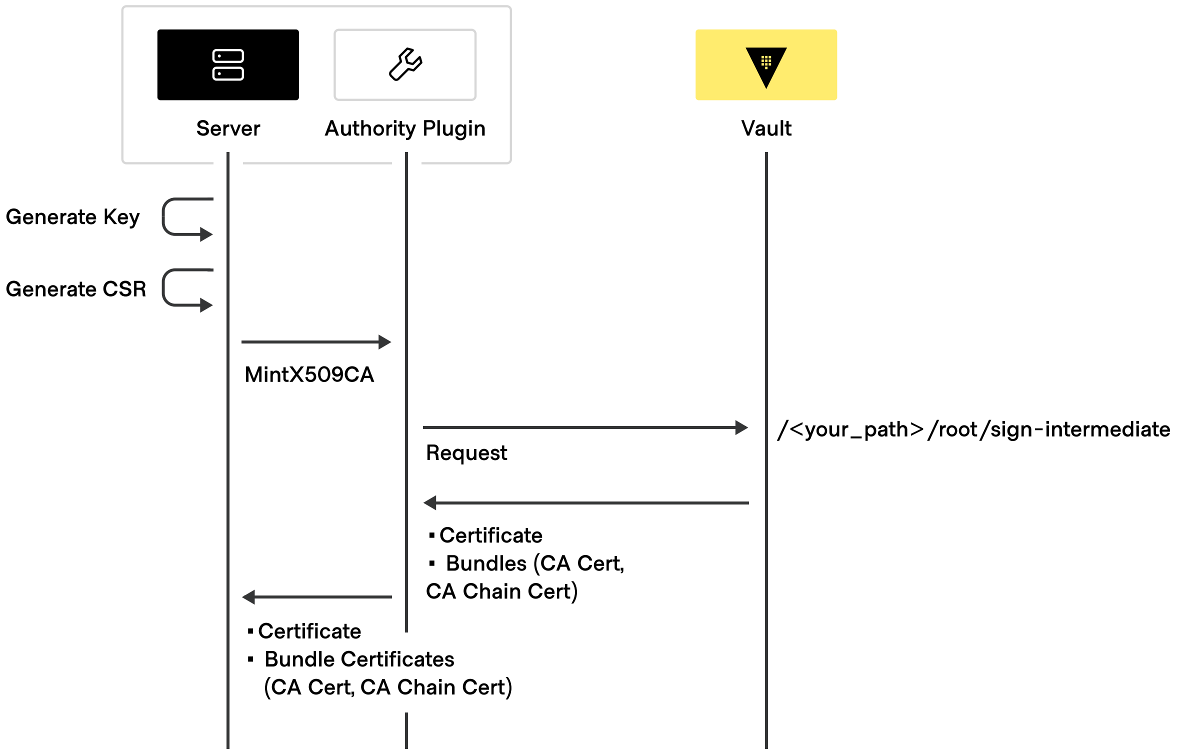
Similar to leaf certificates, a CA certificate can be revoked by sending the revocation request to its parent CA. Note that this also effectively revokes all leaf certificates issued by the Intermediate CA. Clients consider the trust chain as invalid if a revoked CA is present anywhere in the chain. Below is an example of revoking an Intermediate CA using the Revoke Certificate API.

```
# Attempting to revoke a CA’s own certificate will result in an error
cat <<EOF >payload.json
{ “serial_number”: “7c:bd:10:4b:97:f5:d7:4a:78:20:5d:99:a8:4a:d4:86:96:c6:75:69” }
EOF

# CA Certificate revocation request should be sent to the Root CA
curl --header “X-Vault-Token: ${VAULT_TOKEN}” \
    --request POST \
    --data @payload.json \
    ${VAULT_ADDR}/v1/root_ca/revoke
{“request_id”:”501819cb-2ca3-fd18-ea0d-7eacb59c2b6c”,”lease_
id”:””,”renewable”:false,”lease_duration”:0,”data”:{“revocation_
time”:1607926251,”revocation_time_rfc3339”:”2020-12-14T06:10:51.622027Z”},”wrap_
info”:null,”warnings”:null,”auth”:null}
```

Vault and SPIFFE/SPIRE

A popular and fast-growing concept in the workload identity space is the adoption of the Secure Production Identity Framework for Everyone (SPIFFE), which provides a framework for service identities across workloads or environments. The SPIFFE Runtime Environment (SPIRE) is an implementation of this framework and standards to perform attestations by providing identities to the designated workloads.

As a pluggable module, SPIRE can use an external Certificate Authority to sign and create an intermediate certificate and ca_bundle using its custom Authority plugin. As of version 0.10.1, SPIRE now has Hashicorp Vault as an Upstream Authority target. The Vault Authority Plugin authenticates with a provided method of authentication such as a certificate, token or Approle credential in order to submit a CSR to Vault. For further details on the project, please review the Upstream Authority “Vault” Plugin repository.


Server
Authority Plugin
Vault
Generate Key
Generate CSR
MintX509CA
Request
/<your_path>/root/sign-intermediate
▪ Certificate
▪ Bundles (CA Cert, CA Chain Cert)
▪ Certificate
▪ Bundle Certificates (CA Cert, CA Chain Cert)

Reference: https://github.com/zlabjp/spire-vault-plugin **by Tomoyo Usami**

Outside of Vault being the Intermediate CA for SPIRE, Vault itself has a workflow that has been production ready for the past couple of years that provides the same end-state in securing workload identity using Vault Agent. When it comes to SPIRE attestation, Vault Agent can use auto-auth (Vault Agent authentication to Vault) to register the node/virtual machine using a cryptographic identity (e.g. GetCallerIdentity query using the AWS Signature v4 algorithm via the AWS IAM Auth Method) that would perform secure introduction without storing a secret to authenticate to Vault to get a secret.

Monitoring/Audit Recommendations

We recommend monitoring Vault's audit log and alerting on the following key events. Set Signed and Generate Intermediate operations should only happen once during the typical CA lifecycle; other operations will be infrequent admin actions.

Operation	Example path and data snippets
Set Signed Intermediate: This overwrites the CA certificate. An incorrect certificate could break the chain of trust, resulting in application outages.	`“path”: “inter_ca/intermediate/set-signed”`
Generate Root: This generates a new public/private key pair for the CA.	`“path”: “root_ca_v1/root/generate/internal”`
Generate Intermediate: This generates a new public/private key pair for the CA.	`“path”:”inter_ca_v1/intermediate/generate/internal”`

Create/Update Role: This allows creating new endpoints to sign or generate certificates. Some of the attributes could be unmasked to ensure they are expected	`“path”:”inter_ca_v1/roles/app1”,”data”:{ “allow_subdomains”: “hmac-sha256:57379a3b26e972e866c6b8b911d0bb51ebb346bbbcba72d86c329316e54e9727”, “allowed_domains”: “hmac-sha256:b90ebe31b59f7ff1800e7be42f18a9b138eb309e918404eff52eab2f440622e5”, “generate_lease”: “hmac-sha256:57379a3b26e972e866c6b8b911d0bb51ebb346bbbcba72d86c329316e54e9727”, “max_ttl”: “hmac-sha256:7d9dcad5d9014488edae69e48896e6ee128ff97c6f91815f88cf517ee2fca43d” }`
Set URLs: Configure URLs that are encoded in leaf certificates.	`“path”:”inter_ca_v1/config/urls”,”data”: { “crl_distribution_points”: “hmac-sha256:9118438101abf78fee011b5b17d7f73e2b1669516ac54bbde5451c3df9637cdc”, “issuing_certificates”: “hmac-sha256:d00789f531a4b6438296a0f3a91f04c3b976708013c0478a7266e8626ae237a8”, “ocsp_servers”: “hmac-sha256:b771569ba2b5065b35db3488f3115f0c2c4ada3485552446fe47ec5be2de6a2a”}`
Tidy: Delete expired or revoked certificates from Vault’s storage backend	`“path”:”inter_ca_v1/tidy”,”data”: {“safety_buffer”:”hmac-sha256:965d91fd3afb2d7805b8de0a78f118e8613bce4373eb9ceaec0ea9c5e24c33e4”, “tidy_cert_store”:”hmac-sha256:57379a3b26e972e866c6b8b911d0bb51ebb346bbbcba72d86c329316e54e9727”, “tidy_revoked_certs”:”hmac-sha256:57379a3b26e972e866c6b8b911d0bb51ebb346bbbcba72d86c329316e54e9727”}`

Other CA-related events not listed here, such as Generate Certificates, should be logged, but an alert does not need to be generated every time. Below is an example Audit snippet from Intermediate set-signed:

```
“request”: {
    “id”: “71db2b4c-db98-6bcb-e140-99aa2b163c69”,
    “operation”: “update”,
    “mount_type”: “pki”,
    “client_token”: “hmac-sha256:ab3382cd72ca13813e1d9659898b8e3f86bd53b897e5a7535b68
c82372f5991a”,
    “client_token_accessor”: “hmac-sha256:250f3fab6a425945fcf2fea28ab1050b9598747483d
6978f69a73987e1ded883”,
    “namespace”: {
      “id”: “root”
    },
    “path”: “inter_ca/intermediate/set-signed”,
    “data”: {
      “certificate”: “hmac-sha256:6640556feaab3eb2aaffc3ee8b63029d51635f7904d5925ed769
f0e2a5b106ee”
    },
    “remote_address”: “127.0.0.1”
  },
  “response”: {
    “mount_type”: “pki”
  }
```

Patterns: Managing Leaf Certificates

The Vault PKI secrets engine allows clients to request X.509 certificates from an issuing CA configured within Vault. It is a dynamic secrets engine in which Vault manages the lifecycle of the secrets, in this case, X.509 leaf certificates. In this section, we review how we can use Vault for common tasks associated with the leaf certificate management, including provisioning, distribution, and revocation.

Anti-Patterns

- **Long-lived leaf certificates:** Having long-lived leaf certificates can result in a higher risk of the certificate leaking. Vault's PKI secrets engine streamlines the certificate issuance process, commonly called enrollment, into a single API call. This concept is discussed in greater detail below in the Common Configurations section.
- **Using manual processes:** Manual distribution and rotation of certificates hampers productivity, limits scale, and greatly impacts the speed of application delivery. Manual processes also conflict with the best practice of having short-lived certificates. Automation is discussed further below in the Deployment/Automation section.
- **Sharing leaf certificates:** We recommend that a separate leaf certificate be provisioned for each application and instance. E.g., if there is an application being served by multiple VMs, a separate certificate should be generated by Vault for each VM. Sharing a leaf certificate increases the blast radius if the private key is compromised and the certificate needs to be revoked. In addition, when the certificate expires, updates for multiple instances must be coordinated carefully. When architected correctly, a Vault cluster can issue certificates at a high velocity and scale.
- **Using wildcard subdomains ('*') excessively:** Having globbed subdomains for the certificate CN can reduce the number of leaf certificates needed and reduce overall certificate management overhead. While this may be acceptable for certain environments, audit considerations should be reviewed carefully. For example, it can be more challenging to understand how leaf certificates are actually being used since they can be applied to any application/use-cases covered by `` `*.` `` Limiting the leaf certificate CN and/or SANs to well-defined sub-domains prevents reusing/sharing certificates which is an anti-pattern.

Lifecycle Management

The lifecycle of a leaf certificate generally includes the following phases. We will review how the Vault PKI Secrets Engine can help with each of these.

1. Preparation
2. Certificate Issuance (Enrollment)
3. Usage and Validation
4. Renewal / Rotation

Revocation and auditing are asynchronous lifecycle events covered in the sections Revocation and Monitoring / Audit recommendation.

Preparation

The Vault PKI secrets engine configuration and role configuration are performed during this phase. As with other Vault secrets engines, these steps are usually completed in advance by an operator or configuration management tool.

Often, the preparation phase also includes generating a key pair and then creating a Certificate Signing Request (CSR) with the appropriate identity information and other certificate properties. With Vault, those steps are streamlined as part of the issuance process covered in the next section.

Certificate Issuance

This phase is also referred to as Certificate Enrollment. In this phase, the issuing CA verifies the requester's identity and sign the CSR to issue a valid certificate. Often this is a manual or ticket-driven process that slows down application delivery.

Vault already has a mature authentication model to establish the identity of a client (user or application) via one of its authentication methods. A Vault client authenticates to Vault, then performs an API call to the Generate Certificate endpoint. Thus, Vault encapsulates the actual certificate issuance workflow in a single REST API call.

The Generate Certificate API endpoint has the path, /pki/issue/<role-name>, where pki is the path on which the PKI secrets engine for the Intermediate CA is mounted. The client's ACL Policy must allow the request of a certificate via the full path, which includes both the intermediate CA mount path and the role name. Below is an ACL snippet showing an example path and the relevant permissions.

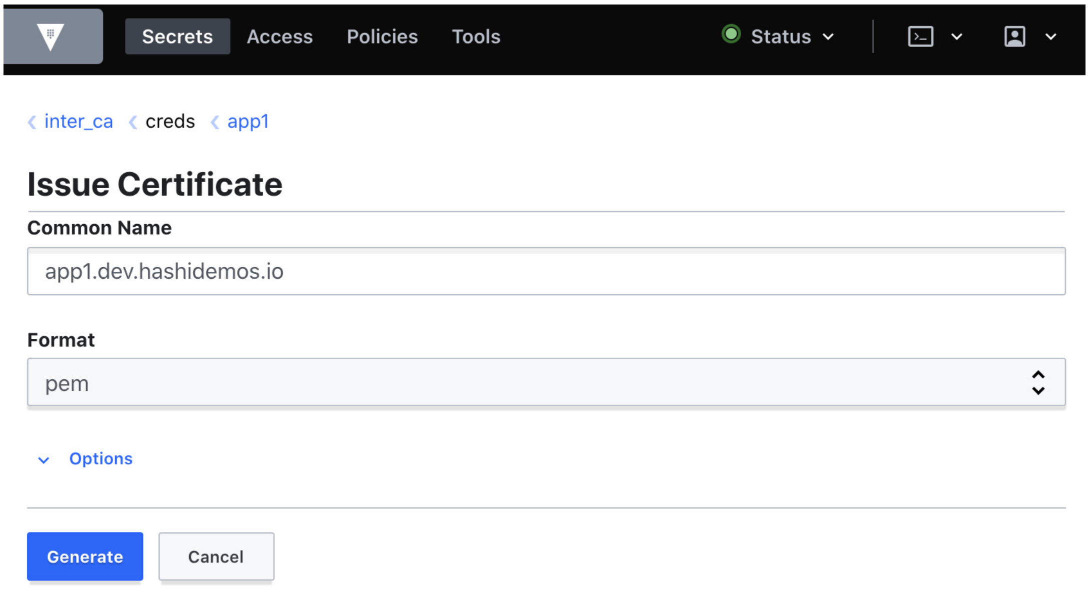
```
# Allow generating leaf certificate from inter_ca secrets engine and role app1
path “inter_ca/issue/app1” {
  capabilities = [“create”,”update”]
}
```

Below are some examples of generating a certificate for the role name app1-hashidemos. (The output is truncated for brevity.) Note that the private key can no longer be displayed after the certificate is issued and that it is up to the application to store the private key securely. With short certificate lifetimes, often, the private key is treated as ephemeral and never written to disk. Each application receives a unique certificate (and public/private key pair) to use during its lifetime.

As shown below, the CA chain is provided in the ca_chain field as part of this API call. Typically, the CA chain is pre-distributed to TLS client truststores by configuration management tools or other methods. Note that when CA certificates are updated, new CA chains are pushed to client truststores.

~~~
```
# Via the CLI
$ vault write inter_ca/issue/app1 common_name=app1.dev.hashidemos.io
Key                 Value
---                 -----
lease_id            inter_ca/issue/app1/6b0CVuI9cVgU686bziPpmm3B
lease_duration      8h
lease_renewable     false
ca_chain            [-----BEGIN CERTIFICATE-----
MIIDOzCCAiOgAwIBAgIUTKBEXzgk/Ckfsc7x3O3PciPSKfEwDQYJKoZIhvcNAQEL
BQAwGDEWMBQGA1UEAxMNaGFzaGlkZW1vcy5pbzAeFw0yMDExMTkxOTA1NDdaFw0y
...bTFLVVW0wQOgPsvbek3b
-----END CERTIFICATE-----]
certificate         -----BEGIN CERTIFICATE-----
MIID2TCCAsGgAwIBAgIUJ4v8VizelZ5+IkFF5GMH+oCI+7YwDQYJKoZIhvcNAQEL
BQAwGDEWMBQGA1UEAxMNaGFzaGlkZW1vcy5pbzAeFw0yMDExMTkxOTA4MDVaFw0y
...N1kiVRgfd1GVwtMGCQsLbhb7AAa5dyFw+6/MckU=
-----END CERTIFICATE-----
expiration          1605841715
issuing_ca          -----BEGIN CERTIFICATE-----
MIIDOzCCAiOgAwIBAgIUTKBEXzgk/Ckfsc7x3O3PciPSKfEwDQYJKoZIhvcNAQEL
BQAwGDEWMBQGA1UEAxMNaGFzaGlkZW1vcy5pbzAeFw0yMDExMTkxOTA1NDdaFw0y
...bTFLVVW0wQOgPsvbek3b
-----END CERTIFICATE-----
private_key         -----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEA0TIu8aORGYErwjMi7uqamtiqME+Pvzq6YDXRr66eDhv5aO72
cc9UhCjIlX27tP2jKcsvvaMOEixPVzVqkWwjPJId3CKIBy822kLjH6EOguUEPRqN
...CZzTobvPJPrhsdqsSty2siWn7n3iY7pTtOClledqbqSZGHim2oI=
-----END RSA PRIVATE KEY-----
private_key_type    rsa
serial_number       27:8b:fc:56:2c:de:95:9e:7e:22:41:45:e4:63:07:fa:80:88:fb:b6

# API invocation via curl
$ cat <<EOF >payload.json
~~~

```
{
  “common_name”: “app1.dev.hashidemos.io”
}
EOF
$ curl \
    --header “X-Vault-Token: <vault-token>” \
    --request POST \
    --data @payload.json \
    http://127.0.0.1:8200/v1/inter_ca/issue/app1
{
  “request_id”: “0d2e7ea8-ed58-2c1a-2a1e-dd97c9cafc67”,
  “lease_id”: “inter_ca/issue/app1/Xovs9D85LFJgc824OnIgRBAu”,
  “renewable”: false,
  “lease_duration”: 28800,
  “data”: {
    “ca_chain”: [
      “-----BEGIN CERTIFICATE-----\nMIIDOzCCAiOgAwIBAgIUTKBEX....2qfZ1CCErThtZI7E7V\
nbTFLVVW0wQOgPsvbek3b\n-----END CERTIFICATE-----”
    ],
    “certificate”: “-----BEGIN CERTIFICATE-----\nMIID2TCCAsGgAwIBAgIUGStM4....
tnSYp4Tr8n\ni5hk7r/16oyZjDfZ5eARcsjsq9Jo3aLxFSgi5xY=\n-----END CERTIFICATE-----”,
    “expiration”: 1605842327,
    “issuing_ca”: “-----BEGIN CERTIFICATE-----\
nMIIDOzCCAiOgAwIBAgIUTKBEXzgk....2qfZ1CCErThtZI7E7V\nbTFLVVW0wQOgPsvbek3b\n-----END
CERTIFICATE-----”,
    “private_key”: “-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAvkIiu31Y2mRgHVuo
Gxpc....4OexRmSu9oi17nnnr33wqhT5WjE9+lY7pfrQ\n-----END RSA PRIVATE KEY-----”,
    “private_key_type”: “rsa”,
    “serial_number”: “19:2b:4c:e2:d8:8b:5c:36:47:7e:95:57:be:c9:62:3b:54:a6:c1:31”
  },
  “wrap_info”: null,
  “warnings”: null,
  “auth”: null
}
```



Common Name: app1.dev.hashidemos.io

Format: pem

Screen snippet of generating a certificate from the Vault UI

Usage and Validation

X.509 leaf certificates are issued for a range of use-cases, including encrypted communication channels (TLS, IPSec), authenticating servers and clients (applications, users, or IoT devices), and code signing. TLS communication is the most commonly-observed use case for Vault PKI. During the TLS handshake process, the leaf certificate is validated by the client's cryptographic library to ensure that a proper chain of trust is present up to the root CA. Below is an example snippet showing manual validation of a Vault-issued leaf certificate and CA chain.

```
### Issue leaf cert from Intermediate CA
vault write -format=json inter_ca_v1/issue/app1 common_name=app1.dev.hashidemos.io |
jq -r '.data.certificate' > leaf_certificate_v1.pem
```

```
### Validate leaf cert issued
openssl verify -verbose -CAfile root_ca_v1.pem \
  -untrusted inter_ca_v1.cert.pem  leaf_certificate_v1.pem

leaf_certificate_v1.pem: OK
```

As part of certificate validation, clients may use the OCSP protocol or view the CRL to check that a certificate has not been revoked. This process is covered more in the Revocation section.

If the leaf certificate is lost and needs to be reread, we can retrieve it via the certificate serial number as shown in the example below.

```
# Read certificate via serial_number
$ vault read inter_ca/cert/52:91:db:ef:bf:a4:a3:f2:f8:b1:c9:1b:d5:10:4f:1b:be:c3:76:bd

Key                Value
---                -----
certificate        -----BEGIN CERTIFICATE-----
MIIEDTCCAvWgAwIBAgIUUpH …
qq2xkPOIzn5ZF2SIUrQnhHTvxulIl3zGyv4I+23740UI
-----END CERTIFICATE-----
revocation_time    0
```

Renewal/Rotation

When a leaf certificate is approaching its expiration time, it must be either renewed or rotated. Both actions result in a new certificate with an updated expiration date. Renewal does not change the public/private key pair, whereas rotation entails generating a new key pair.

The Vault PKI Secrets Engine allows easy certificate rotation by generating a new certificate using the same issuing CA and role name used to generate the original certificate. We recommend this strategy over a renewal to lower the chances of the private key getting compromised.

Looking Up Expiration Time

In this section, we discuss a few ways for viewing the expiration time for a leaf certificate.

- When the leaf certificate .pem file is readily available, we can view its expiration time using the openssl tool. If needed, we can obtain the .pem file for a previously issued certificate using its serial number with the Read Certificate endpoint.

```
# (Optional) Read a certificate using the serial number and save the pem file
serial_number=73:10:0d:14:8f:90:c8:1f:08:ad:72:46:94:be:13:a9:ea:7f:64:14
vault read -field=certificate inter_ca_v1/cert/$serial_number > cert.pem

# Display certificate using openssl
openssl x509 -in ./cert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            73:10:0d:14:8f:90:c8:1f:08:ad:72:46:94:be:13:a9:ea:7f:64:14
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=dev.hashidemos.io
        Validity
            Not Before: Dec 23 12:51:36 2020 GMT
            Not After : Dec 23 20:52:06 2020 GMT
        Subject: CN=app1.dev.hashidemos.io
...
```

- Another way to view the expiration time is to look up the remaining time for the Vault lease associated with the certificate. This method may be easier since we do not need to know the certificate serial number or have the .pem file. However, we can only use this method when the Intermediate CA Role has lease generation enabled. In the example below, first, we list the lease IDs for a given CA and Role name using the List Leases endpoint. Then we display an individual lease.

~~~~
```
# Lookup up lease IDs for inter_ca_v1 and the app1 role
vault list sys/leases/lookup/inter_ca_v1/issue/app1
Keys
----
1Q3kZdKx0fZMK5zu9jHFuEPE
4IKCCbqnBD7v1GyHtudP8C5A
5oyOHUFQS0MT1v31C1NzwVVJ
7ygHhbukREUB1zUJpGw5YKeW
...

# Display the lease using the lease ID
curl -s --header “X-Vault-Token: $VAULT_TOKEN” \
    --request PUT \
    --data ‘{“lease_id”:”inter_ca_v1/issue/app1/1Q3kZdKx0fZMK5zu9jHFuEPE”}’ \
    $VAULT_ADDR/v1/sys/leases/lookup | jq ‘.data’

{
  “expire_time”: “2020-12-23T15:58:37.000055-05:00”,
  “id”: “inter_ca_v2/issue/app1/1Q3kZdKx0fZMK5zu9jHFuEPE”,
  “issue_time”: “2020-12-23T07:58:37.370543-05:00”,
  “last_renewal”: null,
  “renewable”: false,
  “ttl”: 27954
}
```
~~~~

- Finally, we can also view the leaf certificate expiry time from the Audit log entry. We have provided a snippet in the Monitoring/Audit Recommendations section. The advantage of this method is that we do not need to login to Vault or need the certificate pem file. Additionally, customized reports of all certificates nearing expiration can be generated from the log monitoring system.

Rotating/Generating a New Certificate

With Vault PKI, this process is the same as certificate issuance using the Generate Certificate API endpoint. We covered this process earlier in the Certificate Issuance section. Vault-aware tools such as Consul Template or Vault Agent do this automatically when the certificate approaches its expiration time.

Renewal

There are some scenarios where it is desirable to keep using the same public/private key pair. For example, long-lived application instances that are unable to reload new keys until the next scheduled restart. In that case, we can generate a new CSR and sign it using Vault's Sign Certificate endpoint. This change is reflected in the example below for CN=app1.dev.hashidemos.io.

Note: the Certificate serial number is updated (since it is a new leaf certificate).

```
# Generate a CSR using existing private key
$ openssl req -new -key app1.dev.hashidemos.io-myprivatekey.pem \
  -out app1.dev.hashidemos.io-csr.pem

# Now use the Sign Certificate endpoint to generate certificate
$ vault write inter_ca/sign/app1 csr="$(cat app1.dev.hashidemos.io-csr.pem)"

Key                 Value
---                 -----
ca_chain            [-----BEGIN CERTIFICATE-----
MIIDOzCCAiOgAw...
LvV//SSgVNPXf8dHsdPZ
-----END CERTIFICATE-----]
certificate        -----BEGIN CERTIFICATE-----
MIIEDTCCAvWgAwIBA...
mxHa1xOTFcOqvs75wqmtihF52/YsrL32XN+HAXdwrn6d
-----END CERTIFICATE-----
expiration          1606562607
issuing_ca          -----BEGIN CERTIFICATE-----
MIIDOzCCAiOgAwIBAgIUChflUh...
LvV//SSgVNPXf8dHsdPZ
```

```
-----END CERTIFICATE-----
serial_number      2b:cb:80:e6:0f:c8:2b:25:34:41:01:0a:65:46:b2:b8:bf:d4:d5:8a
```
```

Recommendations:

- Monitor and alert if the certificate rotation process failed. For example, if the Vault Agent is used to render leaf certificates, the login credentials (such as AppRole) may have expired. Vault Agent produces errors in its log when this happens. We recommend ingesting logs into a log monitoring system and alerting on this type of error.
- Use Vault-aware tools such as Vault Agent or Consul Template that are aware of the certificate TTL and will rotate the certificate automatically.
- There should be mechanisms in place to deploy or reload the application when the certificate has changed. Vault Agent and Consul Template both support issuing commands after the new certificate is rendered.

Deployment / Automation

Typically, leaf certificates are issued via a pipeline during the application deployment. For short-lived applications such as containers, functions, and ephemeral VMs, the lifetime/TTL of the certificate is at least as long as that of the application. In the case of long-lived applications, tools are used to periodically refresh the leaf certificate.

Application deployment tools can directly interact with Vault via the API to issue certificates on behalf of the application. There are also a few commonly used tools to help with this process:

- Vault Agent []
- Consul Template []
- EnvConsul []
```

These tools are aware of the certificate TTL and will refresh the certificate before it expires. This approach mitigates the risk of application outages due to expired certificates. Another benefit is that they alleviate manual processes by automating the certificate refresh workflow.

Config management tools such as Ansible, Chef, or Puppet can also be used to login to Vault and fetch certificates for applications. These tools can also play a role in updating application truststores with the CA chain when CAs are configured or updated. This blog post provides an example of reading secrets from Vault using Chef. For Ansible, reference the HashiVault module, which supports reading certificates from the PKI secrets engine.

## Example Vault Agent Configuration

The snippet below shows an example *template { }* stanza for Vault Agent to render the TLS certificate, private key, and the CA chain.

Vault Agent renders a new leaf certificate automatically before its expiry time. If lease generation was enabled for the role, it renders the new leaf certificate when approximately 85% of the lifetime has elapsed. Otherwise, it renders the certificate just before the `validTo` time is reached.

```
# TLS SERVER CERTIFICATE
template {
  contents = “{{ with secret \”inter_ca/issue/app1\” \”common_name=nginx.dev.
hashidemos.io\” }}{{ .Data.certificate }}{{ end }}”
  destination = “/etc/ssl/nginx.dev.hashidemos.io.crt”
}

# TLS PRIVATE KEY
template {
  contents = “{{ with secret \”inter_ca/issue/app1” \”common_name=nginx.dev.
hashidemos.io\” }} {{ .Data.private_key }}{{ end }}”
  destination = “/etc/ssl/nginx.dev.hashidemos.io.key”
}
```

```
# TLS CA CERTIFICATE
template {
  contents = “{{ with secret \”inter_ca/issue/app1\” \”common_name=nginx.dev.
hashidemos.io\” }} {{ .Data.issuing_ca }}{{ end }}”
  destination = “/etc/ssl/ca.crt”
}
```

When Vault Agent renders the leaf certificate successfully, it logs an INFO message. Vault Agent will not explicitly revoke the previous lease; rather, it is cleaned up by Vault when the lease TTL has been reached. An INFO message is also logged by Vault showing lease expiration.

```
# Output from Vault agent log rendering a new leaf certificate
2020/12/23 13:37:52.341494 [INFO] (runner) rendered “/tmp/certs/dynamic-cert.tpl” =>
“/tmp/certs/app-certs.txt”

# Output from Vault revoking the previous lease
2020-12-23T08:38:36.016-0500 [INFO]  expiration: revoked lease: lease_id=inter_ca_v2/
issue/app1/1kj4t8iy6XVQeVY4Ns510hpQ
```

Revocation

Vault allows us to revoke leaf certificates issued by the Issuing CA. There are two ways to achieve this: using individual certificate IDs or revoking the associated leases. When a certificate is revoked, Vault immediately updates the CRL with this information.

When do I need to revoke leaf certificates?

While all expired certificates are considered invalid, sometimes unexpired certificates should also be considered invalid. This force expiration can be achieved by revoking the leaf certificate and using a mechanism to let clients know about the revocation.

X.509 PKI RFC 5280 defines a set of possible reason codes associated with X.509 certificate revocation. The most important reason is if the private key of the certificate was compromised while the certificate has not yet expired (reason code keyCompromise). This error code can appear, for example, if the associated private key was revealed in an error log or was placed in a known compromised location while the certificate has not yet expired.

Another reason could be that the certificate has incorrect properties and is therefore unusable. For example, if the subject DN is wrong or it does not have the proper extensions. For human-oriented use cases, a reason could be that someone left the company and their assigned leaf certificate has not expired.

We recommend keeping certificate lifetimes short to lower the likelihood of a compromise while a certificate is still valid. With low TTLs (such as a few hours), chances are that when a private key is exposed, either maliciously or accidentally, the corresponding certificate has already expired. If the certificate has already expired, there is generally no need to explicitly revoke it since TLS clients should be rejecting the certificate based on its "Not After" time.

Revoking leaf certificates using the lease ID

To use this method, the generate_lease option must be enabled for the Role configuration under the PKI secrets engine. By default, this option is set to false. Please see the Intermediate CA Common Configurations section for more details.

The lease ID is provided when the certificate was generated using the pki/issue/<role_name> endpoint. Alternatively, outstanding lease IDs can be looked up using the List Leases API endpoint. Below are examples showing these.

```
# lease_id when generating a certificate
$ vault write inter_ca/issue/app1 common_name=app1.dev.hashidemos.io
Key                 Value
---                 -----
lease_id            inter_ca/issue/app1/h5s7BlLwh03SOcoUjsu6rtiL
lease_duration      8h
lease_renewable     false
```

```
ca_chain            [-----BEGIN CERTIFICATE-----
<output truncated>
…
-----END RSA PRIVATE KEY-----
private_key_type    rsa
serial_number       64:11:09:b4:93:5a:f1:62:7a:a0:9b:f2:7c:94:ba:fa:f2:12:b8:2c

# Showing all lease IDs from the app1 role
$ curl -s --header “X-Vault-Token: $(vault print token)” \
  --request LIST \
  “http://127.0.0.1:8200/v1/sys/leases/lookup/inter_ca/issue/app1” | jq .data.keys
[
  “2ZiLdKPlgpa4ykihhpudcLeS”,
  “6b0CVuI9cVgU686bziPpmm3B”,
  “AJU7m11HdU7QJ6YxclXiAUZB”,
]
```

Once we have the lease ID(s), we can use the **vault lease revoke** command to revoke certificates individually. Alternatively, we can destroy all outstanding leases issued from the role or the entire issuing CA. Below are example commands showing these actions.

```
# Revoking a certificate via the corresponding lease ID
$ vault lease revoke inter_ca/issue/app1/AJU7m11HdU7QJ6YxclXiAUZB
All revocation operations queued successfully!

# Revoking all certificates issued from a role
$ vault lease revoke -prefix inter_ca/issue/app1/
All revocation operations queued successfully!
```

Revoking leaf certificates via certificate serial number

Another way to revoke a certificate is by using the Revoke Certificate API and providing the certificate serial number. The serial number is provided when it is first issued by Vault. It can also be retrieved later from the audit log. Note that if lease generation was disabled for the Role, then this method is the only option for revoking certificates. The snippet below shows an example.

```
# Write a payload with certificate serial #
$ cat <<EOF >payload.json
{ “serial_number”: “2a:dc:19:12:3e:53:c2:b2:e1:63:81:c6:8a:95:00:24:02:db:76:62” }
EOF

# Send a request to the revoke API endpoint
$ curl \
    --header “X-Vault-Token: $(vault print token)” \
    --request POST \
    --data @payload.json \
    ${VAULT_ADDR}/v1/inter_ca/revoke

{“request_id”:”0efc5643-fa8a-8294-68bf-ec4717aeb112”,”lease_
id”:””,”renewable”:false,”lease_duration”:0,”data”:{“revocation_
time”:1606489853,”revocation_time_rfc3339”:”2020-11-27T15:10:53.299995448Z”},”wrap_
info”:null,”warnings”:null,”auth”:null}
```

Revocation checking for clients

When using PKI at scale with Vault, there are a few options for checking revoked certificates:

- None – rely on low certificate TTLs
- Use CRLs
- Implement OCSP responder servers

Some organizations may use a mix of the above strategies based on the required level of compliance.

Whenever possible, our recommendation is to use short certificate lifetimes. They are treated consistently across different crypto libraries and are simple and predictable. With the streamlined PKI issuance from Vault, certificate lifetimes can be as short as you want to make them.

For compliance-driven organizations, there is typically a need to publish CRL endpoints and/or implement OCSP responders. Even with these requirements in place, using short certificate lifetimes means fewer certificates end up on the revocation list, thereby allowing a more straightforward implementation of revocation checking mechanisms.

In contrast to certificate expiry checking, revocation checking behavior using CRL or OCSP varies by client libraries. There have been various improvements in this area, such as using CRLSets and OCSP Stapling; these are out of scope for this whitepaper.

The Vault PKI Secrets engine publishes a CRL from each issuing CA as covered previously in the section discussing intermediate CA Common Configurations. Currently, Vault does not operate an OCSP responder; therefore, a separate OCSP server infrastructure must be deployed. This server can periodically query Vault for revoked certificates via CRLs, Vault APIs, or a push-based solution can be implemented.

If configured as part of the intermediate CA role, CRL endpoints and OCSP servers are listed in the X509 certificate under the “`CRL Distribution Points`” (X509v3) and “`Authority Information Access`” sections, respectively. Below is an example snippet from a Vault-issued X.509 certificate showing CRL and OCSP information.

```
$ openssl x509 -in ./cert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            50:8d:aa:ff:ea:b9:39:dc:38:e9:6f:a3:69:3e:75:df:7b:e9:bf:78
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=hashidemos.io
        Validity
            Not Before: Nov 30 15:30:06 2020 GMT
            Not After : Nov 30 23:30:36 2020 GMT
        Subject: CN=*.dev.hashidemos.io
<truncated>
            Authority Information Access:
                OCSP - URI:https://ocsp.hashidemos.io/v1/inter_ca
                CA Issuers - URI:http://127.0.0.1:8200/v1/inter_ca/ca
            X509v3 Subject Alternative Name:
                DNS:*.dev.hashidemos.io, URI:https://*.dev.hashidemos.io
            X509v3 CRL Distribution Points:
                Full Name:
                  URI:http://127.0.0.1:8200/v1/inter_ca/crl
```

Viewing revoked certificates in the CRL

Below is a simple example of checking whether a leaf certificate is valid by examining the CRL against the Certificate serial #. The URL for the CRL endpoint was retrieved previously from the **X509v3 CRL Distribution Points** section of the certificate (see the previous snippet). Programmatic revocation checking for TLS clients varies depending on the crypto library implementation.

```
$ serial_number=2c:db:db:bf:09:2e:3d:5c:6f:b2:ff:d0:ce:1a:e2:85:00:21:e6:4f
$ curl -s “${VAULT_ADDR}/v1/${CA}/crl” \
    --output ${CA}.crl \
    && openssl crl -inform DER -text -in ${CA}.crl \
    | grep -i $(echo ${serial_number} | tr -d ‘:’)
    Serial Number: 2CDBDBBF092E3D5C6FB2FFD0CE1AE2850021E64F
```

Validating certificates via OCSP

Checking the CRL and searching for a revoked certificate is an expensive operation for TLS clients. OCSP provides a more streamlined way to check for revoked certificates that have not yet expired.

OCSP responder URLs should be configured as part of the issuing CA configuration. Once configured, Vault includes this information under the **Authority Information Access** section of X.509 certificates. The TLS client should examine this section and query the OCSP responder URI(s) for certificate validation. The exact implementation of how OCSP servers are queried depends on the client crypto library being used.

Tidying

Vault stores all issued certificates in its storage backend with encryption at rest using the cryptographic barrier. Over time accumulated certificates can lead to high utilization of the storage backend, especially in environments with a high velocity of certificate issuance.

The Tidy API endpoint allows one to remove expired and revoked leaf certificates from Vault's storage backend. Please note some important considerations below for using this endpoint.

- Once the certificate is removed, it can no longer be read from Vault. We recommend carefully establishing some data retention policies regarding expired and revoked certificates. Once these timelines are established, we can use the `safety_buffer` parameter to tell Vault that only certificates older than the buffer time should be cleaned up.
- Consider adding a Vault Enterprise Sentinel Endpoint Governing Policy (EGP) to enforce that the `safety_buffer` aligns with established data retention policies. This policy prevents the accidental removal of recently-expired certificates.
- Depending on how many certificates need to be removed, the Tidy endpoint can be an extremely I/O intensive task for the storage backend (Consul or Raft). We recommend carefully planning and testing a tidy operation in advance, especially in environments with a high certificate issuance volume. Running Tidy operation on a scheduled interval allows for a more predictable load for Vault and Consul.

In the following snippet, we show two example API calls to the Tidy endpoint to delete expired and revoked certificates older than 6 months and one year, respectively (the safety_buffer for these time frames are expressed as hours).

```
# Remove expired certificates older than 6 months
$ curl \
    --header “X-Vault-Token: $VAULT_TOKEN” \
    --request POST \
    --data ‘{“tidy_cert_store”: “true”, “safety_buffer”:”4382h”}’ \
    “${VAULT_ADDR}/v1/inter_ca/tidy”

# Remove revoked certificates older than 1 year
$ curl \
    --header “X-Vault-Token: $VAULT_TOKEN” \
    --request POST \
    --data ‘{“tidy_revoked_certs”: “true”, “safety_buffer”:”8764h”}’ \
    “${VAULT_ADDR}/v1/inter_ca/tidy”
```

Common Configurations

In this section, we will review some common configurations and recommendations for scaling PKI with Vault.

Leaf Certificate Time-To-Live

If the private key of a leaf certificate is compromised while the certificate is still valid (i.e., it has not expired), a man-in-the-middle attack could be launched. To lower the chances of a compromise like this, we recommend keeping the certificate lifetime small. This rule has a range of benefits, some of which are outlined below:

- A shorter certificate TTL means a lower risk that the TLS Certificate is still valid if the corresponding private key was accidentally exposed or maliciously accessed. Expired certificates do not need to be explicitly revoked as crypto stacks should reject them. Revoking expired certificates would result in productivity loss due to executing additional workflows.
- For some use cases, lower certificate TTLs may alleviate the need to implement CRLs and/or OCSP servers (see more on this in the next section). In case CRLs are implemented, it will at least result in a

smaller CRL size.

- Finally, lowering TTLs imply that automation must be in place to issue certificates both during application deployment and when the certificate is near expiry.

For ephemeral application instances such as containers, functions, or short-lived VMs, the certificate lifetime may match the application's lifetime. For example, a VM-based application that is restarted in a rolling manner every two weeks might have certificates that are valid for 2.5 weeks. Containers and functions-as-a-service have shorter lifetimes on the order of hours and days.

If the application is long-lasting, then we recommend using Vault-aware tools to render new certificates periodically during the application lifetime.

Time-To-Live Hierarchy

The PKI Secrets Engine serves Dynamic Secrets that Vault creates just-in-time and actively manages their lifecycle. Dynamic secrets in Vault are usually associated with a Lease and a well-defined TTL. However, because X.509 Certificates have an explicit expiration time (the "Not After" date as specified in RFC 5280 [reference]), generating the actual lease object within Vault is optional. Please see the Intermediate CA Common Patterns section regarding the generate_lease option for role configuration.

Any certificate issued by Vault contains a Not After time-based property on the effective TTL at the time of certificate generation. The TTL can be specified in a variety of ways, as summarized in the table below.

Maximum TTL for the leaf certificate from highest to lowest.

There is a default-lease-ttl setting that is used as the actual TTL when generating a certificate.

Precedence (highest to lowest)	**Example Configuration**
The ttl parameter supplied when generating a certificate. This will be capped at the max-lease-ttl setting for the role or Intermediate CA PKI secrets engine.	`vault write inter_ca/issue/app1 common_name=app1.dev.hashidemos.io` **ttl=2m**

Maximum TTL configured for the role	`vault secrets tune -max-lease-ttl=4h inter_ca/roles/app1`
Maximum TTL configured for the Intermediate CA PKI secrets engine	`vault secrets tune -max-lease-ttl=8h inter_ca`
Vault system: defaults to 32 days but it can be overridden	`default_lease_ttl=7000h` `max_lease_ttl=8760h`

Other Leaf Certificate Considerations

Discovery: Vault's audit log can help you discover all leaf certificates issued by Vault and associate them with the requester. However, there is often a requirement to have an enterprise-wide inventory of all leaf certificates issued by both Vault and non-Vault CAs. This type of capability is outside the scope of Vault and better served by existing Certificate Lifecycle Management (CLM) solutions. These solutions often integrate with Vault PKI.

- For example, KeyFactor has several ways to discover certificates across the enterprise, including direct CA integration, SSL/TLS endpoints, and Certificate Stores. When integration is enabled with Vault, KeyFactor can discover certificates across all Vault instances, namespaces, and issuing CAs and bring them into a single enterprise-wide inventory. Please view this Key Factor + HashiCorp Vault solution brief for more details.

Compliance: Vault's intermediate CA can be configured with various options to ensure that any certificate issuing policy can be accommodated. There may be additional requirements for continuous monitoring, reporting, and alerting. While Vault can help enforce certificate compliance, additional solutions may need to be in place to demonstrate enterprise-wide compliance and provide monitoring and alerting functionality for all certificates and CAs across the enterprise.

CLM solutions such as KeyFactor and Venafi can integrate with Vault PKI to provide such capabilities. Please see KeyFactor and Venafi solution briefs. Both include Vault Secret Engine plugins that can be easily accessed using Vault APIs.

Monitoring / Audit Recommendations

Vault will log audit entries for all PKI secrets engine activity. In this section, we provided some audit log snippets from key activities. Using the audit log, you can create a report of all certificates issued or revoked by Vault. As with all Vault audit log entries, most of the fields are HMAC'd for security reasons.

We recommend unmasking certain fields, such as the certificate serial number, which can be used for log correlation. To unmask all certificate IDs in the audit log for an intermediate CA, issue the command: "**vault secrets tune -audit-non-hmac-response-keys=serial_number <ca-name>**". To only do this for a specific role, please adjust the path to <ca-name>/issue/<role-name>. If there is a need to read a certificate later via the certificate serial number, the Read Certificate API endpoint can be used.

The expiration date is already unmasked and shown as UTC; monitoring this may be desirable for long-lived certificates in order to avoid outages.

Generating a certificate

The "response" portion from the audit entry that is logged when a certificate is generated:

~~~
```
“response”: {
  “mount_type”: “pki”,
  “data”: {
    “ca_chain”: [
     “hmac-sha256:f6aa3109ab199614276ecaa8f0ee343c81f5affb5e9379be00382b59f22128be”
    ],
    “certificate”: “hmac-sha256:acbe95f4be1c5e5de1238ef484ece9d7fc072ac06ec6fb4e39bff6
51e4f32975”,
    “expiration”: 1606135867,
    “issuing_ca”: “hmac-sha256:f6aa3109ab199614276ecaa8f0ee343c81f5affb5e9379be00382b
59f22128be”,
    “private_key”: “hmac-sha256:2c11c3cb20de14bd6cc9ab452289b1505b7824f9cb5f8d913192b
854f56b5764”,
    “private_key_type”: “hmac-sha256:79c6244f4f0ee6ed6f099f5173f934e74d6b1c70f5a05d5f
fefddbdde6f70989”,
~~~

```
    “serial_number”: “hmac-sha256:6bd62bc537dcbee7a1d88ba9cea3025f1e6f273189bf37e6bae
9346dd9721665”
  }
}
```

If lease generation is configured for the role, the lease ID is also shown under the secret stanza:

```
“response”: {
  “mount_type”: “pki”,
  “secret”: { “lease_id”: “inter_ca/issue/app1/blaQkQk0usJtDR6jOsLMGUum” },
  “data”: {
    “ca_chain”: [“hmac-sha256:f6aa3109ab199614276ecaa8f0ee343c81f5affb5e9379be00382b5
9f22128be”]
```

Revoking a certificate

As we covered in the Revocation section, there are two ways to revoke a leaf certificate. Below are example audit log snippets for both methods.

- Using the Revoke Certificate endpoint with certificate serial #

```
“request”: {
    “id”: “e98e16d8-baba-1ddf-92b1-4ed3ea6ea14c”,
    “operation”: “update”,
    “mount_type”: “pki”,
    “client_token”: “hmac-sha256:b1b22f0b88af6ae5f8203748603d40769a9367d1cda5836b953f
f564105e2b8b”,
    “client_token_accessor”: “hmac-sha256:4a9434577775336a48f9c970d627a6564e402a99d7e
cfb9dff402f64dbc6c8be”,
    “namespace”: {
      “id”: “root”
    },
    “path”: “inter_ca_v2/revoke”,
    “data”: {
      “serial_number”: “hmac-sha256:95395ef7fe9918355b9bbc7b921507218e9ac46248bf31a01
38d29843ac1e681”
    },
    “remote_address”: “127.0.0.1”
  },
  “response”: {
    “mount_type”: “pki”,
    “data”: {
      “revocation_time”: 1608736803,
      “revocation_time_rfc3339”: “hmac-sha256:555a981a8e3020be30ebf5dd17a4e92911453aa
d77506cddece3fef3c95f5b24”
    }
  }
}
```

- Using the Revoke Lease endpoint. In the example below, we are revoking a specific lease ID. Leases can be revoked at the Secret Engine or Role level as well.

```
“request”: {
    “id”: “6d11c678-8ca4-0118-9936-386b41b4856d”,
    “operation”: “update”,
    “mount_type”: “system”,
    “client_token”: “hmac-sha256:74c00f75226be5fd711e499462dee91e6cb2e48a3a508f364fe7
21191a5af617”,
    “client_token_accessor”: “hmac-sha256:a50149ce2d79eee037e36b0ed2c593b2281df735dbe
c0f606beca44a5077cfa4”,
    “namespace”: {
      “id”: “root”
    },
    “path”: “sys/leases/revoke/inter_ca_v2/issue/app1/B5V0JjG3DH548KJHneerJeqk”,
    “data”: {
      “sync”: false
    },
    “remote_address”: “127.0.0.1”
  },
  “response”: {
    “mount_type”: “system”,
    “data”: {
      “http_content_type”: “hmac-sha256:71e81a7a525e8b4d0d025687c4d16bd9987a7aad89d99
5db5ea821a47967b3fe”,
      “http_status_code”: 202
    }
  }
}
```

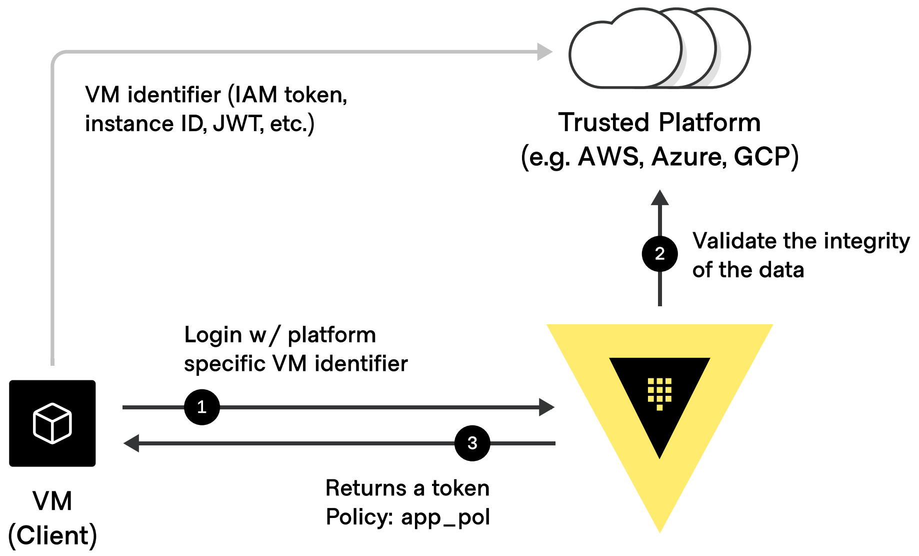
Secure Introduction

Securely Introducing Vault Clients

When authenticating to Vault, a client needs to provide some type of secret to identify itself, be it an AppRole RoleID/SecretID combo, a TLS certificate, or cloud provider keys. Using and transmitting these secrets creates the risk of exposure to unauthorized entities. In the hands of an adversary, these secrets can result in unauthorized data access. This secret introduction method creates a challenge: How does the secret consumer (an application or machine) prove that it is the legitimate recipient of a secret so that it can acquire a token? There are two basic approaches to securely authenticating a secret consumer to Vault.

Platform Integration

Using the Platform Integration approach, Vault is configured to trust the underlying platform (ex: AWS, Azure, GCP), which assigns an identifier (such as an AWS IAM token or a signed JWT) to a virtual machine, container, or serverless function. Vault uses the provided identifier to verify the client's identity by interacting with the underlying platform that assigns the identifiers. Once the platform confirms the identity with Vault, the identity is considered verified, and Vault returns a token to the client, bound to the specified roles.

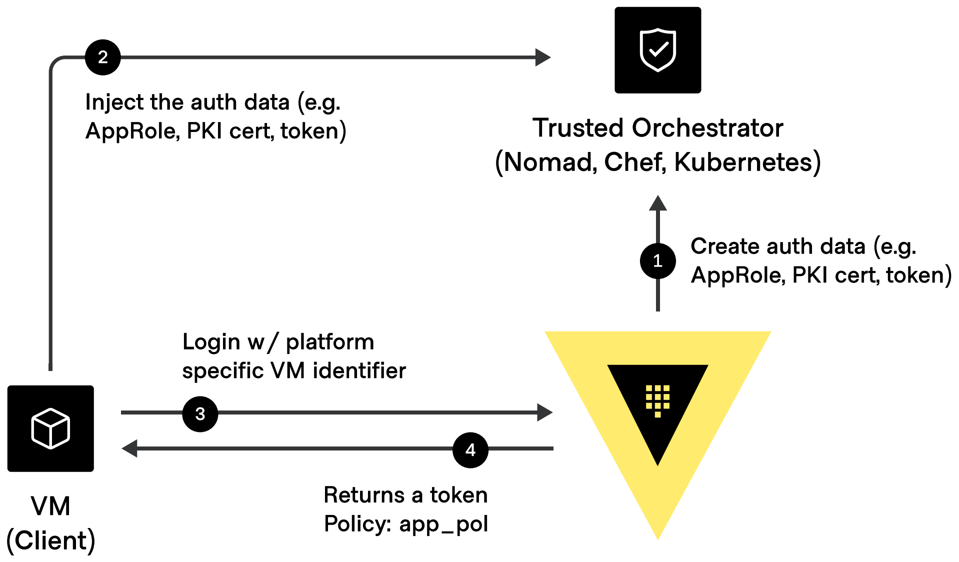

VM identifier (IAM token, instance ID, JWT, etc.)
Trusted Platform (e.g. AWS, Azure, GCP)
2
Validate the integrity of the data
Login w/ platform specific VM identifier
1
3
VM (Client)
Returns a token Policy: app_pol

This approach is ideal when the client application is running on a VM hosted on a cloud platform that has a corresponding auth method. Some example auth methods for this approach are:

- AWS Auth Method
- Azure Auth Method
- GCP Auth Method

Trusted Orchestrator

With the Trusted Orchestrator approach, you have a tool or service that has been manually configured to interact with Vault and has privileged permissions. When the orchestrator launches new applications or services, it can inject the secrets required to authenticate that application or service with Vault.


2
Inject the auth data (e.g. AppRole, PKI cert, token)
Trusted Orchestrator (Nomad, Chef, Kubernetes)
1
Create auth data (e.g. AppRole, PKI cert, token)
Login w/ platform specific VM identifier
3
4
Returns a token Policy: app_pol
VM (Client)

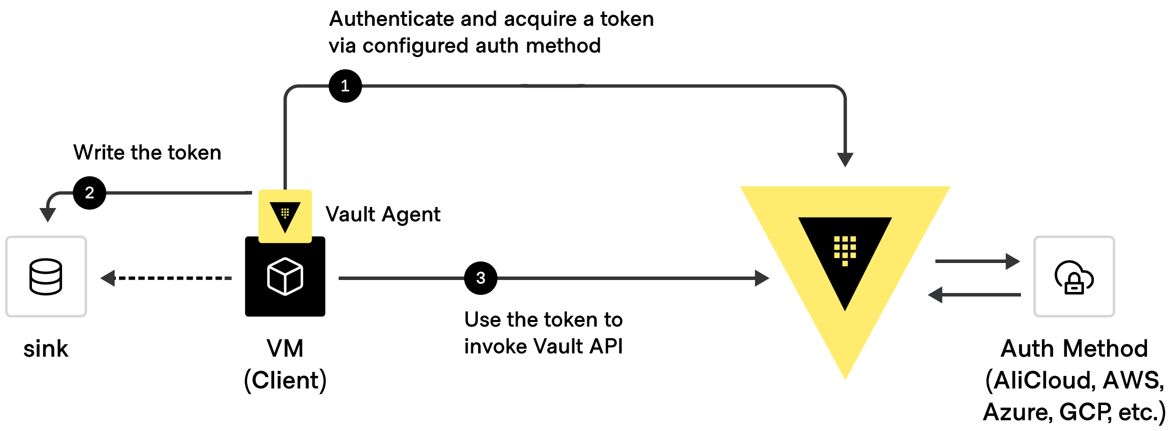
This approach is ideal when using an orchestrator tool such as Chef to launch applications and can be applied regardless of where the applications are running, as orchestrators are built to be platform-agnostic. Some example auth methods for this approach are:

- AppRole Auth Method
- TLS Certificates Auth Method
- Token Auth Method

Automating Secure Introduction

Vault Agent Auto-Auth

The Vault Agent Auto-Auth approach leverages a client daemon to automate the above platform integration or trusted orchestrator approaches. With Auto-Auth, Vault Agent automatically authenticates to the Vault service using one of the supported auto-auth methods. Once Vault Agent receives its token after successful authentication, it continues to manage token renewals until it reaches the end of its allowed lifecycle and can no longer be renewed.


Authenticate and acquire a token via configured auth method
1
Write the token
2
Vault Agent
sink
VM (Client)
3
Use the token to invoke Vault API
Auth Method (AliCloud, AWS, Azure, GCP, etc.)

To leverage this feature, run the Vault binary in agent mode on the client, with a configuration file that specifies which auth method to use and which sink location(s) to store the token(s). When the agent is started, it attempts to acquire a Vault token using the specified auth method. If successful, the resulting token is written to the configured sink location. If the token value changes, the agent captures the change and writes the new token to the sink. If authentication fails, the agent falls into a wait-retry loop. More information on this approach below:

- Vault Agent documentation
- Auto-Auth documentation

PKI Solution Architecture

Deployment Reference Architecture

This section covers the best-practice reference architecture for running the Vault service in a production environment. A Vault cluster consists of a set of Vault processes running across a group of hosts that together provide a single service endpoint. The Vault service consists of a more extensive collection of clusters across multiple locations. These processes can be running on physical or virtual servers or in containers.

Vault supports many different storage engines; however, the only two supported by HashiCorp with Vault Enterprise are the Consul storage engine and the Raft storage engine. HashiCorp typically recommends Consul over Raft as the latter is relatively new to Vault, so this reference architecture uses Consul as the preferred storage backend. A Consul cluster consists of a set of Consul processes running across a group of hosts that together provide a single service endpoint. They can be running on physical or virtual servers or in containers.

It is recommended that the Vault and Consul services are operated on separate clusters. By separating Vault from the Consul storage backend, users can independently scale the server types in each cluster. Additionally, operational issues are simpler to troubleshoot when the two services are isolated. Since Consul is a memory-intensive tool, it is advantageous to have a dedicated and separate cluster, to avoid resource contention or starvation. The Consul cluster should not be used for other Consul-specific functionality (like service discovery) as it could potentially introduce utilization spikes.

The goal of this reference architecture design is to provide maximum flexibility, scalability, and resilience across the components of the Vault service.

Hardware Considerations

Below, the server size recommendations are split into two categories: small and large.

- **Small:** appropriate for initial production deployments or non-production deployments.
- **Large:** for production workloads with a highly consistent workload (a large number of transactions, secrets, or a combination).

In general, processing requirements are dependent on encryption workload and messaging workload (operations per second and types of operations).

Memory requirements are dependent on the total number of secrets/keys stored in memory and should be sized according to that data (as should the hard drive storage). Vault has minimal storage requirements, but the underlying storage backend should have a relatively high-performance hard disk subsystem. If many secrets are being generated or rotated frequently, this information needs to be written to disk often and can impact performance if slower hard drives are used. In the case of using Vault for PKI, this would include x.509 certificates generated by Vault.

Consul's role in a Vault deployment is to serve as the storage backend for Vault. This means that all content stored for persistence in Vault is encrypted by Vault and written to Consul's in-memory key-value store. As such, memory can be a constraint in scaling as more data is written to Vault. This also has the effect of requiring vertical scaling on the Consul server's memory if additional space is required.

Consul Server Requirements

Size	CPU	Memory	Disk
Small	2 cores	8-16 GB RAM	50 GB
Large	4-8 cores	32-64+ GB RAM	100 GB

Vault Server Requirements

Size	CPU	Memory	Disk
Small	2 cores	4-8 GB RAM	25 GB
Large	4-8 cores	16-32 GB RAM	50 GB

Network Requirements (for both Vault and Consul)

Network throughput is a common consideration for Vault and Consul servers. As both services are fronted with an HTTPS API, all incoming requests, communications between Vault and Consul, underlying gossip communication between Consul cluster members, communications with external systems (per auth or secret engine configuration, and some audit logging configurations) and responses consume network bandwidth.

Due to network performance considerations in Consul cluster operations, replication of Vault datasets across network boundaries should be achieved through Performance or DR Replication, rather than spreading the Consul cluster across network and physical boundaries. If a single Consul cluster is spread across network segments that are distant or inter-regional, this can cause synchronization issues within the cluster, leader re-elections, or generate additional data transfer charges in some cloud providers. The latency between availability zones should be less than 8ms for a round trip.

The following table outlines the network traffic requirements for Vault and Consul cluster servers:

Source	**Destination**	**port**	**protocol**	**Direction**	**Purpose**
Consul clients and servers	Consul server	7300	TCP	incoming	Server RPC
Consul clients	Consul clients	7301	TCP and UDP	bidirectional	LAN gossip
Vault clients	Vault servers	8200	TCP	incoming	Vault API
Vault servers	Vault servers	8201	TCP	bidirectional	Vault replication traffic, request forwarding

Communicating with the Vault Cluster

- There are several options for communicating with the Vault cluster:
- Using host IP addresses or hostnames that are resolvable via a standard named subsystem.
- Using load balancer IP addresses or hostnames that are resolvable via a standard named subsystem.
- Using the attached Consul cluster DNS as service discovery to resolve Vault endpoints.
- Using a separate Consul service discovery cluster DNS as service discovery to resolve Vault endpoints.

The implementation details of these options are outside the scope of this document.

Consul Reference Architecture

Since we need to have an underlying Consul cluster to use as the storage backend for our Vault cluster, we should first review the Consul reference architecture. The Consul reference architecture is slightly different when deploying Consul OSS than when deploying Consul Enterprise. Those differences are covered below.

Because Consul relies upon raft consensus to organize and replicate information, we want to provide three unique resilient paths to provide meaningful reliability. Essentially, a consensus system requires a simple majority of servers to be available at any time. It is recommended that the Consul servers are spread across three availability zones to ensure the best chance for healthy consensus.

When deploying Consul OSS, it is recommended to run with a total of five Consul servers, spread across three availability zones in a 2-1-2 configuration (2 servers in Zone A, 1 server in Zone B, and 2 servers in Zone C). Consul can lose any single zone and remain operational.

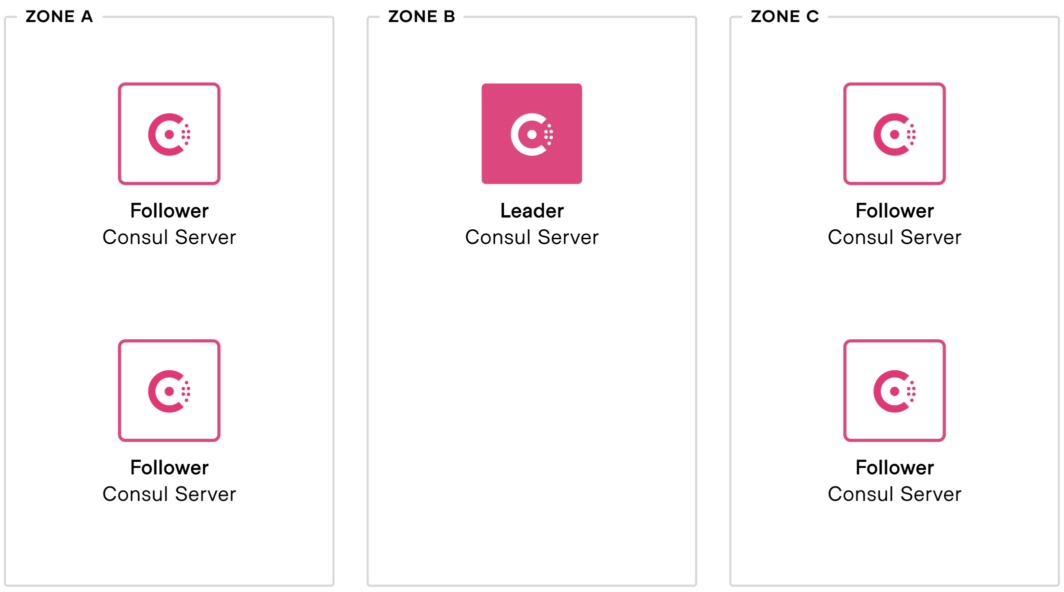


Image: Consul OSS Cluster

When deploying Consul Enterprise, the deployment changes a bit to take advantage of Consul Enterprise's Autopilot with Redundancy Zones feature. Redundancy zones run a single Consul voter and any number of non-voters in a defined zone. Non-voters still receive data from cluster replication to help horizontally scale out reads, but they do not participate in quorum election operations. If the voter is lost in a zone, Autopilot steps in and automatically promotes the non-voter to become a voter and start participating in quorum election operations, restoring the full quorum participation almost immediately.

This also allows us to speed up quorum operations by reducing the quorum size of the cluster from five with Consul OSS to three with Consul Enterprise. Similarly to the preferred Consul OSS deployment, we spread the Consul servers across three availability zones. Distributing the cluster across availability zones allows us to lose an entire availability zone and maintain quorum, but we can also now survive the failure of many individual servers within a given redundancy zone as well. At least two servers per redundancy zone are recommended (a total of 6 at minimum).

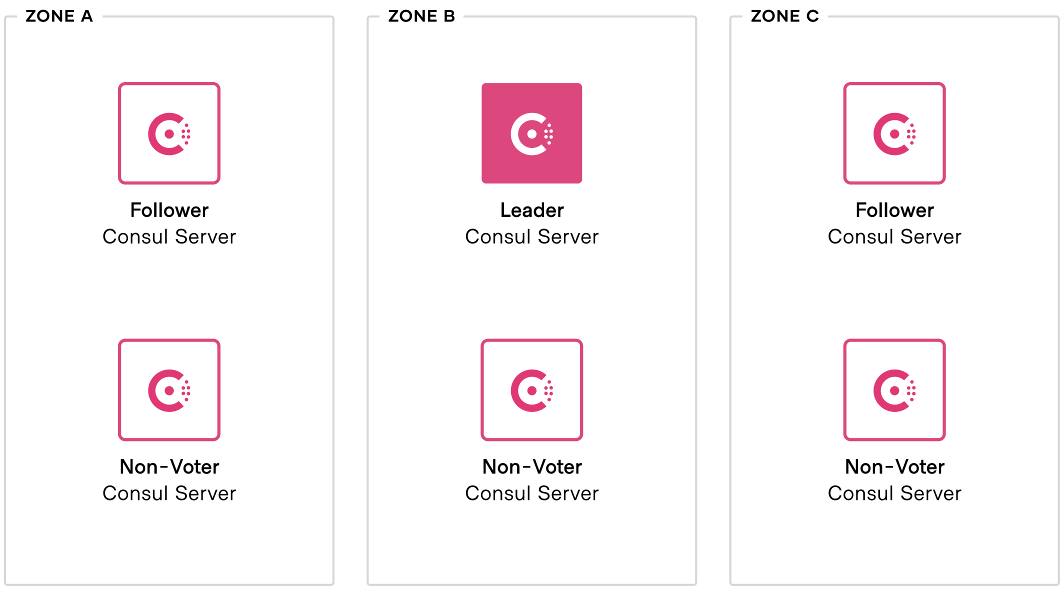


Image: Consul OSS Cluster

Vault Reference Architecture

To meet the lofty resiliency goals put forth for Vault, we recommend that no less than three Vault servers are used. These servers should be spread across three availability zones, similar to the Consul cluster deployment model. With Vault Enterprise, HashiCorp recommends leveraging performance standbys to help scale out read-only workloads. More information on this is available in the Scale and Performance Considerations section below. In both cases, you would have three or more Vault servers in your Vault cluster.

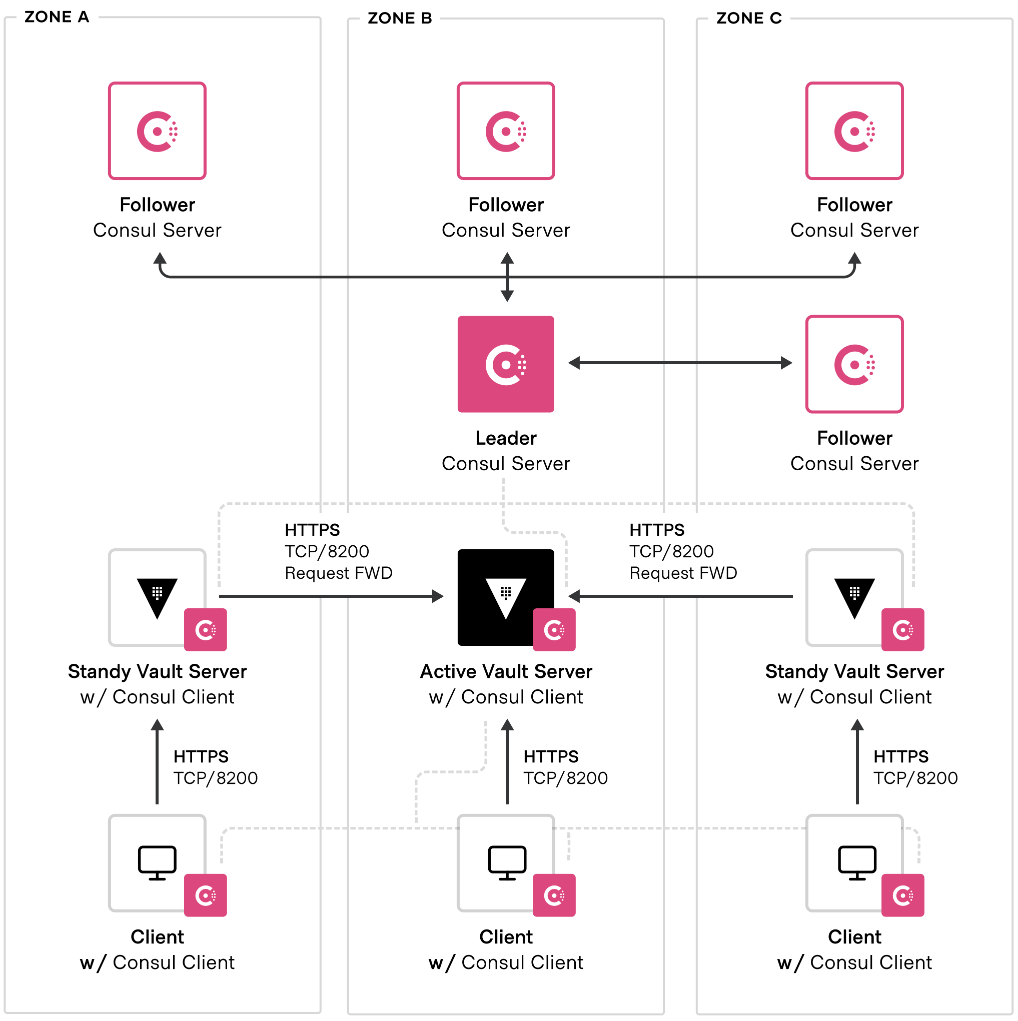


Image: Vault OSS and Consul OSS

In the above scenario, the servers in the Vault cluster and the associated Consul cluster are hosted between three availability zones. This solution has an n-2 redundancy at the server level for Vault and Consul and n-2 redundancy for Vault at the availability zone level. This also has an n-1 redundancy at the availability zone level for Consul and, as such, is considered the most resilient of all architectures for a single Vault cluster with a Consul storage backend for the OSS product.

For a cluster using Vault Enterprise, the reference architecture changes slightly as it uses the Consul Enterprise cluster described above, leveraging Autopilot and redundancy zones.

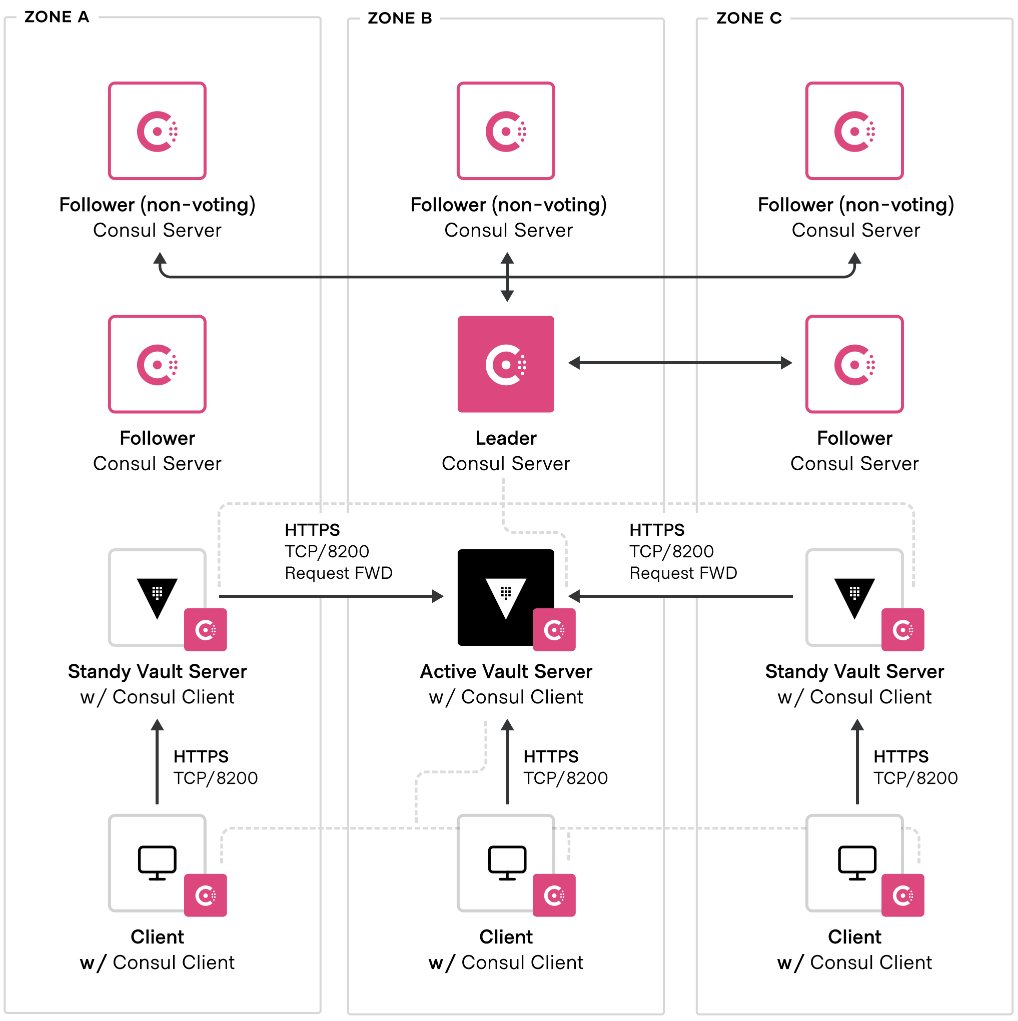


Image: Vault Enterprise and Consul Enterprise

In this scenario, the servers in the Vault cluster and its associated Consul cluster are hosted between three availability zones. This solution has an n-2 redundancy at the server level for Vault and an n-3 redundancy at the server level for Consul. At the availability zone level, Vault is at n-2 redundancy and Consul at n-1 redundancy. This model differs from the OSS design in that the Consul cluster contains six servers, with three of them as non-voting members. The Consul cluster is set up using redundancy zones so that if any server were to fail, a non-voting member would be promoted by Autopilot to become a full member and so maintain quorum.

Multi-Region Deployment Reference Architecture

To this point, we've been talking about Vault within the context of a single cluster spread across multiple availability zones. This design is excellent for surviving individual server failures and availability zone failures; however, it still leaves the Vault cluster exposed to region failure. To protect against region-level failure, Vault Enterprise provides two features: Performance Replication and Disaster Recovery Replication.

Performance Replication

In performance replication, the secondary cluster shares the same underlying configuration, policies, roles, and static secrets (such as key-value pairs or PKI keys) as the primary cluster. A performance replication secondary, however, can keep track of its own tokens and leases (decoupled from the primary). If a user action would cause an underlying write to the shared state, the replication secondary forwards that operation to the primary, which is transparent to the end-user. In practice, most high-volume workloads (such as key-value reads or transit operations) can be satisfied by the secondary, enabling horizontal scale-out.

Disaster Recovery Replication

In disaster recovery replication, the secondary cluster shares the same underlying configuration, policies, roles, and static secrets (such as key-value pairs or PKI keys) as the primary cluster; however, it also shares the same tokens and leases as the primary cluster. This mode is designed to allow for continuous operation of applications in the event of region failure. The disaster recovery cluster cannot be used for reads or writes until it has been promoted to primary.

For the most resilient possible multi-region deployment, HashiCorp recommends at least three performance clusters, spread across three separate regions, each with its own disaster recovery cluster.

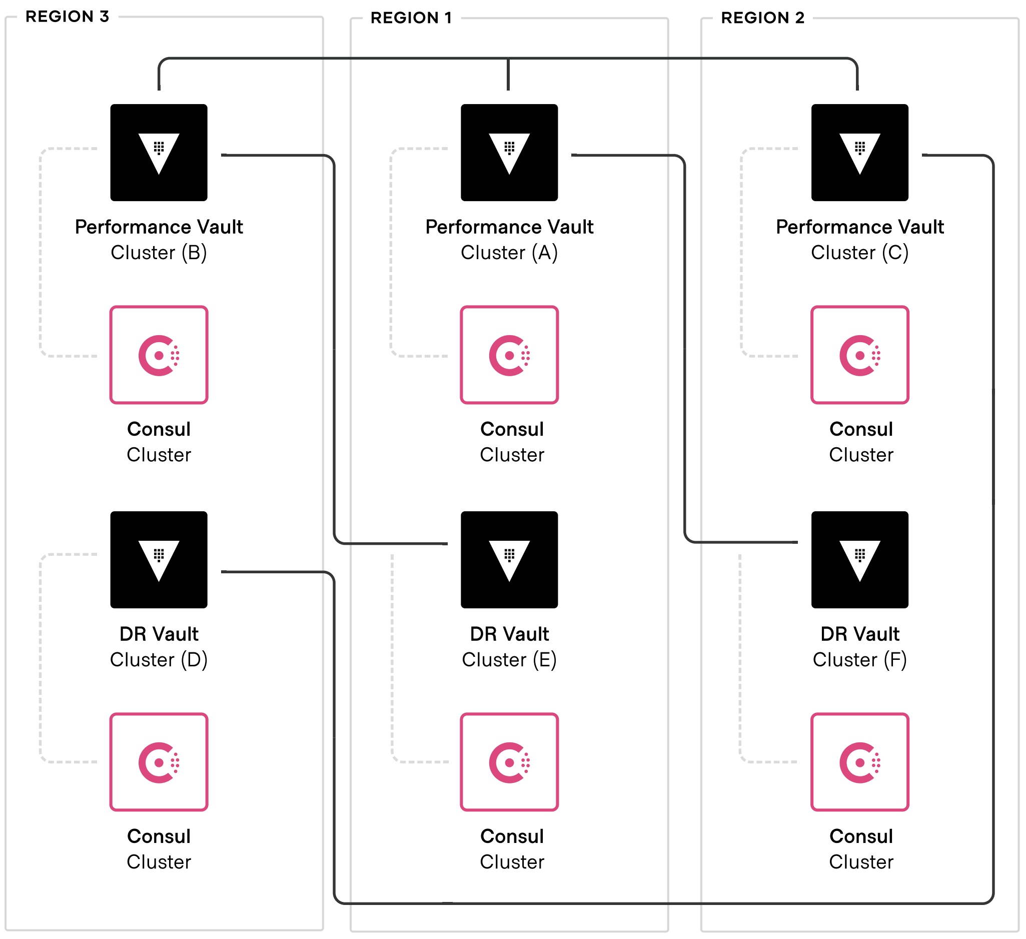


Image: A resilient Vault service deployment

Production Hardening

There are several other recommendations from HashiCorp that are based on Vault's security model and focus on defense in depth.

- **End-to-End TLS.** Vault should always be used with TLS in production. If intermediate load balancers or reverse proxies are used to front Vault, they should not terminate TLS. This way, traffic is always encrypted in transit to Vault and minimizes risks introduced by intermediate layers.
- **Single Tenancy.** Vault should be the only primary process running on a host. This method reduces the risk that another process running on the same host is compromised and can interact with Vault. Similarly, running on bare metal should be preferred to a VM, and a VM preferred to a container. This model reduces the surface area introduced by additional layers of abstraction and other tenants of the hardware. Both VM and container-based deployments work but should be avoided when possible to minimize risk.
- **Firewall traffic.** Vault listens on well-known ports, so use a local firewall to restrict all incoming and outgoing traffic to Vault and essential system services like NTP. This includes restricting incoming traffic to permitted subnets, and outgoing traffic to services Vault needs to connect to, such as databases.
- **Disable SSH / Remote Desktop.** When running Vault as a single-tenant application, users should never access the machine directly. Instead, they should access Vault through its API over the network. Use a centralized logging and telemetry solution for debugging. Be sure to restrict access to logs on a need-to-know basis.
- **Disable Swap.** Vault encrypts data in transit and at rest; however, it must still keep unencrypted sensitive data in memory to function. The risk of exposure should be minimized by disabling swap to prevent the operating system from paging sensitive data to disk. Vault attempts to "memory lock" physical memory automatically, but disabling swap adds another layer of defense.
- **Don't Run as Root.** Vault is designed to run as an unprivileged user. There is no reason to run Vault with root or Administrator privileges, which can expose the Vault process memory and allow access to Vault encryption keys. Running Vault as a regular user reduces its privilege. Configuration files for Vault should have permissions set to restrict access to only the Vault user.
- **Disable Core Dumps.** A user or administrator that can force a core dump and has access to the resulting file can potentially access Vault encryption keys. Preventing core dumps is a platform-

specific process; on Linux, setting the resource limit RLIMIT_CORE to 0 disables core dumps. This setting can be executed by process managers and is also exposed by various shells. In Bash, ulimit -c 0 accomplishes this.

- **Immutable Upgrades.** Vault relies on an external storage backend for persistence, and this decoupling allows the servers running Vault to be managed immutably. When upgrading to new versions, new servers with the upgraded version of Vault are brought online. They are attached to the same shared storage backend and unsealed. Then the old servers are destroyed. This method is similar to a blue-green deployment model and reduces the need for remote access and upgrade orchestration, which may introduce security gaps.
- **Avoid Root Tokens.** Vault provides a root token when it is first initialized. This token should be used to configure the system initially, particularly setting up auth methods so that users may authenticate. We recommend treating Vault configuration as code and using version control services to manage policies. Once set up, the root token should be revoked to eliminate the risk of exposure. Root tokens can be generated when needed and should be revoked as soon as possible once their need has been fulfilled.
- **Enable Auditing.** Vault supports several audit devices. Enabling auditing provides a history of all authenticated operations performed by Vault, generating a forensic trail in the case of misuse or compromise. Audit logs securely hash any sensitive data, but access should still be restricted to prevent any unintended disclosures.
- **Upgrade Frequently.** Vault is actively developed. Regular updates are essential to incorporate security fixes and any changes in default settings such as key lengths or cipher suites. Subscribe to the Vault mailing list, the Discuss forum, and the GitHub CHANGELOG for updates.
- **Configure SELinux / AppArmor.** Using additional mechanisms like SELinux and AppArmor can help provide additional layers of security when using Vault. While Vault can run on many operating systems, we recommend Linux due to the various security primitives mentioned here.
- **Restrict Storage Access.** Vault encrypts all data at rest, regardless of which storage backend is used. Although the data is encrypted, an attacker with arbitrary control can cause data corruption or loss by modifying or deleting keys. Access to the storage backend should be restricted to the Vault service to avoid unauthorized access or operations.
- **Disable Shell Command History.** You may want the Vault command itself to not appear in Bash history on a host. Refer to additional methods for guidance.

- **Tweak ulimits.** Your Linux distribution may have strict process ulimits. Consider reviewing ulimits for the maximum number of open files, connections, etc., before going into production; they may need increasing.
- **Docker Containers.** To leverage the "memory lock" feature inside the Vault container, you will likely need to use the overlayfs2 or another supporting driver.
- **No Clear Text Credentials.** The seal stanza of the Vault server configuration file configures the seal type that will be used for additional data protection, such as using HSM or Cloud KMS solutions to encrypt and decrypt the master key. DO NOT store your cloud credentials or HSM pin in clear text within the seal stanza.
    - If the Vault server is hosted on the same cloud platform as the KMS service, use the platform-specific identity solutions. For example:
        - Resource Access Management (RAM) on AliCloud
        - IAM Role or ECS task on AWS
        - Managed Service Identities (MSI) on Azure
        - Service Account on Google Cloud Platform
    - If that is not applicable, set the credentials as environment variables (e.g., VAULT_HSM_PIN).

Scale and Performance Considerations

Consul Memory Utilization

Since Consul stores all of its data in memory, it is necessary to keep an eye on memory consumption. If you see your memory utilization surpass 70%, consider moving to a machine with a larger memory capacity.

Shorten Certificate Lifetimes (or Managing CRL Size)

Vault's PKI secrets engine aligns with the Vault philosophy of short-lived secrets. As such, it is not expected that CRLs will grow to a large size. A private key is only ever returned to the requesting client. The PKI secrets engine does not store generated private keys, except for CA certificates. In most cases, if the key is lost, the certificate can simply be ignored, as it will expire shortly.

If a certificate must truly be revoked, the normal Vault revocation function can be used; alternately, a root token can be used to revoke the certificate using the certificate's serial number. Any revocation action causes the CRL to be regenerated. When the CRL is regenerated, any expired certificates are removed from the CRL (and any revoked, expired certificates are removed from the PKI secrets engine's storage). Please see the Managing Leaf Certificates section for more information.

Using Raft Storage Backend Instead of Consul Storage Backend

While Consul is HashiCorp's recommended storage backend for running Vault in production, there are some circumstances in which the network latency required to write and read from Consul is too penalizing for a use case. In this case, it may be preferable to use the Raft storage backend instead, allowing Vault to write to its local disk.

Resource Quotas

One side effect of the API-driven model is that applications and users can misbehave by overwhelming system resources through consistent and high volume API requests resulting in denial-of-service issues in some Vault nodes or even the entire Vault cluster. In this case, Vault provides a feature, resource quotas, that allows Vault operators to specify limits on resources used in Vault. Specifically, Vault allows operators to create and configure API rate limits.

Health Monitoring Recommendations for Vault

For a detailed tutorial on how to monitor Vault and Consul clusters, as well as what to monitor, visit the Vault Consul Monitoring Guide.

Deployment Anti-Patterns

- **Single Region Deployments (Enterprise):** Regional outages in the major cloud providers are not that uncommon. Thus, it is never recommended to run Vault Enterprise in a single region. Multiple regions should always be leveraged for robust, production Vault Enterprise deployments.
- **Single AZ Deployments (OSS / Enterprise):** While Vault OSS doesn't give the option for true multi-region deployments, it is still possible to provide resiliency against an availability zone failure by spreading the Vault servers across multiple availability zones as described above.
- **Non-Consul / Raft Backend (Enterprise):** When deploying Vault Enterprise into production, only two storage backends are officially supported by HashiCorp: Consul and Raft. This policy is in place because we have dedicated support teams of subject matter experts for both of those technologies at HashiCorp and cannot provide that same expertise for non-HashiCorp technologies.
- **Deploying Disaster Recovery clusters or Performance Replication clusters in the same region as their respective primaries (Enterprise):** It is never a good idea to pack your Disaster Recovery clusters or Performance Replication clusters into the same region as their respective primary clusters since the loss of the region would result in loss of the primary and the replica clusters.

Gathering Service Level Requirements

With every shared service, it is essential to begin the deployment by gathering requirements and expectations from those who will consume it. However, service consumption isn't expressly limited to an end-user integrating with the service. With PKI in particular, consumers will also likely include security and compliance teams.

Each user group can be classified into three different categories: operators (the team managing the service), policy owners (security and compliance teams), and users (consumers of the service in the traditional sense). To ensure the collection of as many relevant requirements as possible, select a representative sample from each of those categories to discuss the initial requirements.

The following questions can be used to determine what is essential to each of these groups regarding PKI and the ongoing management of SSL certificates:

Operators

- In which geographic areas will the service need to be consumed?
- What are the default TTLs that should be associated with generated leaf certificates?
- What are the rotation requirements and rotation process for root and intermediate certificates?
- What are the retention requirements for audit data associated with SSL management?
- How will user access be mapped to various actions?
- What is the plan/process for deploying the service in new regions?

Security and Compliance

- What are the lifecycle and rotation requirements for certificates and keys?
- Are there any external customer requirements that should be considered in the security design?
- How should administrative and user access be mapped to various actions?
- How should critical security and compliance events be captured from the service?

Users

- Who are the technical contacts for each group of consumers?
- How will service improvements and outages be communicated?
- Are there special regulations that need to be considered for specific regions, such as Europe or China?
- What is the onboarding/integration process for new users, teams, and applications?
- What is the recommended process for consuming and managing leaf certificates?
- How are the requirements and SLAs for external customers mapped to internal requirements and SLAs for the Vault service?
- Which platforms will be consuming certificates?

- What is the definition of “successful integration” with Vault?
- How are the terms “degraded” and “unavailable” defined with regard to the Vault service?

Determining Service Level Objectives

After requirements have been gathered, service level objectives can be created based on those requirements. The purpose of service level objectives is to specifically define the criteria for the operational success of a service. These can be related to availability, performance, or both.

The following are examples of service level objectives that could be used for SSL certificate management:

- Users should receive a valid response for a certificate request for any part of the certificate lifecycle in under 50ms to 99.9% of their requests in any given 30-day period.
- Users should receive a valid response for 99.9% of their certificate requests on up to 1,000 concurrent requests on any particular Vault cluster in any given 30-day period.
- The support team will respond to 99% of SEV 1 outages (complete service disruption) within 15 minutes.
- There should be no more than two minutes of interruption in the arrival of audit log data from any given cluster at any time.

Mapping Service Level Indicators

The purpose of service level indicators is to measure the service’s performance against the defined service level objectives. Additionally, service level indicators are used as a way to measure other critical functions in order to ensure the service is operational at all times.

While it is vital to measure hardware-based factors that can impact the service, such as CPU or RAM utilization, it is more critical to measure the behavior of service functionality, such as how long the service takes to complete a signing request. This method is a much better indicator of the actual health of the service because it provides a more accurate representation of user experience.

The following indicators can be used to measure the health and operations of the PKI secrets engine:

- Generate Certificate
- Sign Certificate
- Revoke Certificate
- Read CRL

A script can be created to mimic one or all of the above actions. Success and time measurements can be taken along with each of these actions to ensure the defined service level objectives are being met consistently.

It is recommended to use the "tidy" action in the API to clean up the certificate revocation list regularly, especially due to the creation and revocation of certificates as a mechanism to monitor the service.

PKI Use-Case Design

Overview

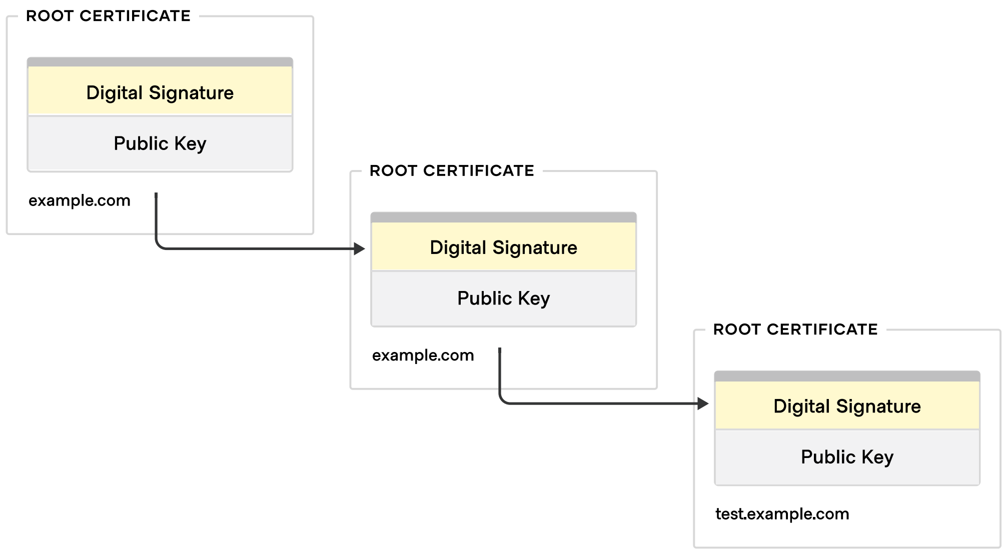
The purpose of this section is to provide concrete examples of how to configure and use Vault as a PKI Management solution. Throughout this section, we will enhance our understanding through additional assets like learning sites, blogs, and example platforms.

Assumptions

Before you begin going through this section, we are assuming the following:

- You have read through the previous sections on the issues with traditional PKI management, the solutions to those issues, and the supporting architecture for those solutions.
- You have access to the internet and can access general websites.
- You have some experience with Linux-based systems.
- A Vault instance or cluster has been deployed, and you can access and configure Vault.

Configuration



Overview

The PKI Secrets Engine generates dynamic X.509 certificates. With this secrets engine, services can get certificates without going through the usual manual process of generating a private key and CSR, submitting to a CA, and waiting for a verification and signing process to complete. Vault's built-in authentication and authorization mechanisms provide verification functionality.

By keeping TTLs relatively short, revocations are less likely to be needed, keeping CRLs short and helping the secrets engine scale to large workloads. This method, in turn, allows each instance of a running application to have a unique certificate, eliminating sharing and the accompanying pain of revocation and rollover.

In addition, by allowing revocation to mostly be forgone, this secrets engine allows for ephemeral certificates. Certificates can be fetched and stored in memory upon application startup and discarded upon shutdown without ever being written to disk.

References

- PKI Secrets Engine
- Build Your Own Certificate Authority (CA)

Steps

Create a Self Signed Root Certificate (Optional)

Use this step if you do not have a CA already.

1. Enable the pki secrets engine at the pki path

```
vault secrets enable pki
```

2. Tune the pki secrets engine to issue certificates with a maximum time-to-live (TTL)

```
vault secrets tune -max-lease-ttl=8760h pki
```

3. Generate the root certificate and save the certificate in CA_cert.crt.

```
vault write -field=certificate pki/root/generate/internal \
    common_name=”example.com” \
    ttl=8760h > CA_cert.crt
```

This process generates a new self-signed root CA certificate and private key. Vault automatically revokes the generated root at the end of its lease period (TTL); the CA certificate signs its own Certificate Revocation List (CRL).

4. Configure the CA and CRL URLs.

```
vault write pki/config/urls \
    issuing_certificates=”http://127.0.0.1:8200/v1/pki/ca” \
    crl_distribution_points=http://127.0.0.1:8200/v1/pki/crl
```

Generate an Intermediate CA

1. Enable the pki secrets engine at the pki_int path.

```
$ vault secrets enable -path=pki_int pki
```

2. Tune the pki_int secrets engine to issue certificates with a maximum time-to-live (TTL) of 4380 hours

```
$ vault secrets tune -max-lease-ttl=4380h pki_int
```

3. Execute the following command to generate an intermediate and save the CSR as pki_intermediate.csr.

```
$ vault write -format=json pki_int/intermediate/generate/internal \
    common_name=”example.com Intermediate Authority” \
    | jq -r ‘.data.csr’ > pki_intermediate.csr
```

4. Sign the intermediate certificate with the root certificate and save the generated certificate as intermediate.cert.pem.

a. **Self Signed Root Certificate**

```
vault write -format=json pki/root/sign-intermediate csr=@pki_intermediate.csr \
   format=pem_bundle ttl=”4380h” \
   | jq -r ‘.data.certificate’ > intermediate.cert.pem
```

b. **External CA Providers**

Send the Certificate Signing Request (CSR) to the CA provider and have a signed certificate generated.

5. Once the CSR is signed and the root CA returns a certificate, it can be imported back into Vault.

```
$ vault write pki_int/intermediate/set-signed certificate=@intermediate.cert.pem
```

Create a Role

A role is a logical name that maps to a policy used to generate those credentials. It allows configuration parameters to control certificate common names, alternate names, the key uses they are valid for, and more.

Here are a few noteworthy parameters:

Param	Description
allowed_domains	Specifies the domains of the role (used with allow_bare_domains and allow-subdomains options)
allow_bare_domains	Specifies if clients can request certificates matching the value of the actual domains themselves
allow_subdomains	Specifies if clients can request certificates with CNs that are subdomains of the CNs allowed by the other role options (NOTE: This includes wildcard subdomains.)
allow_glob_domains	Allows names specified in allowed_domains to contain glob patterns (e.g., ftp*.example.com)

1. Create a role named example-dot-com, which allows subdomains.

```
$ vault write pki_int/roles/example-dot-com \
    allowed_domains="example.com" \
    allow_subdomains=true \
    max_ttl="720h"
```

Integrations

While Vault can integrate into any system using the CLI or REST API methods, we examine two examples of more native system integrations.

Hashicorp Consul

Overview

A common practice is to integrate Consul Service Mesh with Vault to centralize the management of certificates generated for mutual TLS. Do not confuse this section with using Consul as a Storage Backend for Vault, which is out of scope for this document.

The Vault CA provider uses two separately configured PKI Secrets Engines for managing Connect certificates.

The **RootPKIPath** is the PKI engine for the root certificate. Consul uses this root certificate to sign the intermediate certificate. Consul never attempts to write or modify any data within the root PKI path.

The **IntermediatePKIPath** is the PKI engine used for storing the intermediate signed with the root certificate. The intermediate is used to sign all leaf certificates, and Consul may periodically generate new intermediates for automatic rotation. Therefore, Consul requires write access to this path.

If either path does not exist, then Consul attempts to mount and initialize it. This action requires additional privileges by the Vault token in use. If the paths already exist, Consul uses them as configured.

You can choose either Vault or Consul to manage the PKI Paths within Vault. We let Vault manage the PKI Paths to consolidate the management of certificates to Vault only.

There are additional configuration steps to consider when using a Vault / Consul / Kubernetes deployment which can be found at Configuring a Connect CA Provider. This section covers the Vault and Consul integration only.

References

- Vault as a Connect CA
- Configuring a Connect CA Provider

Steps

1. Create a Vault policy which allows Consul to use pre-existing PKI paths in Vault. Consul is granted read-only access to the PKI mount points and the Root CA, but is granted full control of the Intermediate or Leaf CA for Connect clients.

   In this example the RootPKIPath is connect_root and the IntermediatePKIPath is connect_inter. These values should be updated for your environment.

```
# Existing PKI Mounts

path “/sys/mounts” {
  capabilities = [ “read” ]
}

path “/sys/mounts/connect_root” {
  capabilities = [ “read” ]
}

path “/sys/mounts/connect_inter” {
  capabilities = [ “read” ]
}

path “/connect_root/” {
  capabilities = [ “read” ]
}
```

```
path "/connect_root/root/sign-intermediate" {
  capabilities = [ "update" ]
}

path "/connect_inter/*" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}
```

2. When you write this policy to Vault, it generates some information, including the token we use in the Consul configuration file.

```
$ vault token create -policy=pki_consul -ttl=24h
Key                             Value
---                             -----
token                           7PEv0FxJmdFyu0FQbsXzIXwi
token_accessor                  71MSF8PSUP03sw9h5GpxP7cv
token_duration                  24h
token_renewable                 true
token_policies                  ["default" "pki_consul"]
identity_policies               []
policies                        ["default" "pki_consul"]
```

3. Locate the Consul configuration file on all servers and clients attached to your service mesh. For stock installations, you may find them at /etc/consul.d on most Linux distributions.
4. Edit the configuration file on every node by setting **ca_provider** to **"vault"** within the connect stanza and provide the appropriate Vault settings. An example configuration is shown below:

```
connect {
    enabled = true
    ca_provider = "vault"
```

```
        ca_config {
            address = “http://localhost:8200”
            token = “7PEv0FxJmdFyu0FQbsXzIXwi”
            root_pki_path = “connect-root”
            intermediate_pki_path = “connect-intermediate”
        }
    }
```

5. Restart the Consul node agent to complete the changes.

Kubernetes

Overview

Kubernetes can be configured to use Vault as a certificate manager, which enables your services to establish their identity and communicate securely over the network with other services or clients internal or external to the cluster.

Jetstack's cert-manager enables Vault's PKI secrets engine to dynamically generate X.509 certificates within Kubernetes through an Issuer interface.

We're going to assume that you have instances/clusters of Vault and Kubernetes already deployed

References

- Running Vault with Kubernetes
- Integrate a Kubernetes Cluster with an External Vault
- Configure Vault as a Certificate Manager in Kubernetes with Helm

Steps

1. Enable the Kubernetes authentication method.

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

2. Configure the Kubernetes authentication method to use the service account token, the location of the Kubernetes host, and its certificate.

   ```
   $ vault write auth/kubernetes/config \
   token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
   kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \
   kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
   Success! Data written to: auth/kubernetes/config
   ```

   The `token_reviewer_jwt` and `kubernetes_ca_cert` reference files are written to the container by Kubernetes. The environment variable `KUBERNETES_PORT_443_TCP_ADDR` references the internal network address of the Kubernetes host.

3. Create a Kubernetes authentication role named issuer that binds the pki policy with a Kubernetes service account named issuer.

   ```
   $ vault write auth/kubernetes/role/issuer \
       bound_service_account_names=issuer \
       bound_service_account_namespaces=default \
       policies=pki \
       ttl=20m
   Success! Data written to: auth/kubernetes/role/issuer
   ```

The role connects the Kubernetes service account, issuer, in the default namespace with the pki Vault policy. The tokens returned after authentication are valid for 20 minutes. This Kubernetes service account name, issuer, is created in the Deploy Issuer and Certificate section.

4. Install Jetstack's cert-manager's version 0.14.3 resources.

   ```
   $ kubectl apply --validate=false -f https://github.com/jetstack/cert-manager/
   ```

```
releases/download/v0.14.3/cert-manager.crds.yaml
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io created
```

5. Create a namespace named cert-manager to host the cert-manager.

```
$ kubectl create namespace cert-manager
namespace/cert-manager created
```

Jetstack's cert-manager Helm chart is available in a repository that they maintain. Helm can request and install Helm charts from these custom repositories.

6. Add the jetstack chart repository.

```
$ helm repo add jetstack https://charts.jetstack.io
“jetstack” has been added to your repositories
```

Helm maintains a cached list of charts for every repository that it maintains. This list needs to be updated periodically so that Helm knows about all available charts and their releases. A repository recently added needs to be updated before any chart is requested.

7. Update the local list of Helm charts.

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the “jetstack” chart repository
Update Complete. ⎈ Happy Helming!⎈
```

The results show that the jetstack chart repository has retrieved an update.

8. Install the cert-manager chart version 0.11 in the cert-manager namespace.

```
$ helm install cert-manager \
    --namespace cert-manager \
    --version v0.14.3 \
    jetstack/cert-manager
NAME: cert-manager
## …
```

The cert-manager chart deploys several pods within the cert-manager namespace.

9. Get all the pods within the cert-manager namespace.

```
$ kubectl get pods --namespace cert-manager
NAME                                                       READY      STATUS
RESTARTS   AGE
cert-manager-66958f45fc-pdf64              1/1    Running   0          27s
cert-manager-cainjector-755bbf9c6b-gpgtg   1/1    Running   0          27s
cert-manager-webhook-76954fcbcd-w4lll      1/1    Running   0          27s
```

Wait until the pods prefixed with cert-manager are running and ready (1/1).

These pods now require configuration to interface with Vault.

10. Create a service account named issuer within the default namespace.

```
$ kubectl create serviceaccount issuer
serviceaccount/issuer created
```

The service account generated a secret that is required by the Issuer.

11. Get all the secrets in the default namespace.

```
$ kubectl get secrets
default-token-mlm2n          kubernetes.io/service-account-token   3      13d
issuer-token-lmzpj           kubernetes.io/service-account-token   3      47s
sh.helm.release.v1.vault.v1  helm.sh/release.v1                    1      28m
vault-token-749nd            kubernetes.io/service-account-token   3      28m
```

12. The issuer secret is displayed here as the secret prefixed with issuer-token.

```
Create a variable named ISSUER_SECRET_REF to capture the secret name.
$ ISSUER_SECRET_REF=$(kubectl get serviceaccount issuer -o json | jq -r
“.secrets[].name”)
```

13. Create an Issuer, named vault-issuer, that defines Vault as a certificate issuer.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: Issuer
metadata:
  name: vault-issuer
  namespace: default
spec:
  vault:
    server: http://vault.default
    path: pki/sign/example-dot-com
    auth:
      kubernetes:
        mountPath: /v1/auth/kubernetes
        role: issuer
        secretRef:
          name: $ISSUER_SECRET_REF
          key: token
```

```
EOF
issuer.cert-manager.io/vault-issuer created
```

The specification defines the signing endpoint and the authentication endpoint and credentials.

- metadata.name sets the name of the Issuer to vault-issuer
- spec.vault.server sets the server address to the Kubernetes service created in the default namespace
- spec.vault.path is the signing endpoint created by Vault's PKI example-dot-com role
- spec.vault.auth.kubernetes.mountPath sets the Vault authentication endpoint
- spec.vault.auth.kubernetes.role sets the Vault Kubernetes role to issuer
- spec.vault.auth.kubernetes/secretRef.name sets the secret for the Kubernetes service account
- spec.vault.auth.kubernetes/secretRef.key sets the type to token.

14. Generate a certificate named example-com.

```
$ cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: vault-issuer
  commonName: www.example.com
  dnsNames:
  - www.example.com
EOF
certificate.cert-manager.io/example-com created
```

The Certificate, named example-com, requests from Vault the certificate through the Issuer, named vault-issuer. The common name and DNS names are names within the allowed domains for the configured Vault endpoint.

15. View the details of the example-com certificate.

```
$ kubectl describe certificate.cert-manager example-com
```

```
Name:         example-com
Namespace:    default
## ...
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  GeneratedKey  10m   cert-manager  Generated a new private key
  Normal  Requested     10m   cert-manager  Created new CertificateRequest resource
"example-com-1072521490"
  Normal  Issued        70s   cert-manager  Certificate issued successfully
```

The certificate reports that it has been issued successfully.

How-To

For this section, we assume that you have configured your Vault instance/cluster using the steps outlined in the Configuration section, and you have integrated your additional solutions outlined in the Integrations section.

Request a Certificate

Overview

Request signed leaf or endpoint certificates to use in your environment.

References

- PKI Secrets Engine
- Build Your Own Certificate Authority (CA)

Steps

CLI

1. Execute the following command to request a new certificate for the test.example.com domain based on the example-dot-com role.

   ```
   $ vault write pki_int/issue/example-dot-com common_name=”test.example.com”
   ttl=”24h”
   ```

REST API

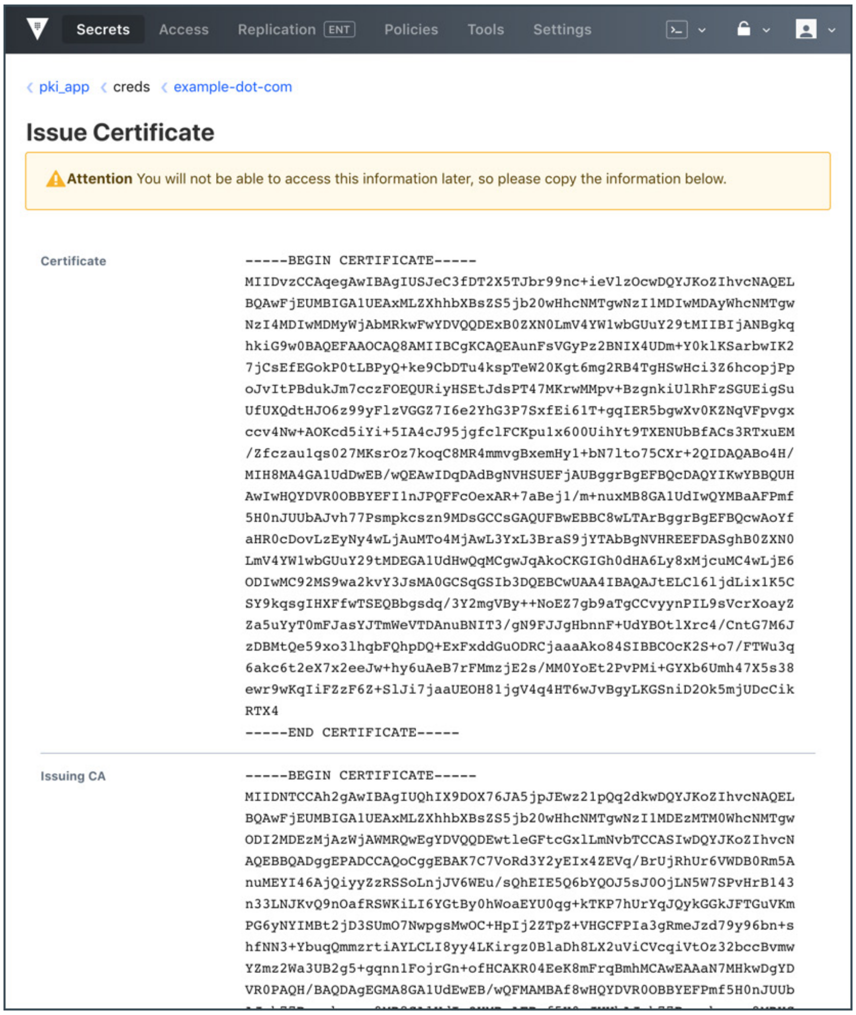
2. Invoke the /pki_int/issue/<role_name> endpoint to request a new certificate.

   Request a certificate for the test.example.com domain based on the example-dot-com role.

   ```
   curl --header “X-Vault-Token: <TOKEN>” \
       --request POST \
       --data ‘{“common_name”: “test.example.com”, “ttl”: “24h”}’ \
       http://127.0.0.1:8200/v1/pki_int/issue/example-dot-com | jq
   ```

Web UI

1. Select **Secrets**.
2. Select **pki_int** from the **Secrets Engines** list.
3. Select **example-dot-com** under **Roles**.
4. Enter **test.example.com** in the **Common Name** field.
5. Expand **Options** and then set the **TTL** to **24 hours**.
6. Click **Generate**.



Issue Certificate

Attention You will not be able to access this information later, so please copy the information below.

Certificate

```
-----BEGIN CERTIFICATE-----
MIIDvzCCAqegAwIBAgIUSJeC3fDT2X5TJbr99nc+ieVlzOcwDQYJKoZIhvcNAQEL
BQAwFjEUMBIGA1UEAxMLZXhhbXBsZS5jb20wHhcNMTgwNzI1MDIwMDAyWhcNMTgw
NzI4MDIwMDMyWjAbMRkwFwYDVQQDExB0ZXN0LmV4YW1wbGUuY29tMIIBIjANBgkq
hkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAunFsVGyPz2BNIX4UDm+Y0klKSarbwIK2
7jCsEfEGokP0tLBPyQ+ke9CbDTu4kspTeW20Kgt6mg2RB4TgHSwHci3Z6hcopjPp
oJvItPBdukJm7cczFOEQURiyHSEtJdsPT47MKrwMMpv+BzgnkiUlRhFzSGUEigSu
UfUXQdtHJO6z99yFlzVGGZ7I6e2YhG3P7SxfEi61T+gqIER5bgwXv0KZNqVFpvgx
ccv4Nw+AOKcd5iYi+5IA4cJ95jgfclFCKpu1x600UihYt9TXENUbBfACs3RTxuEM
/Zfczau1qs027MKsrOz7koqC8MR4mmvgBxemHy1+bN7lto75CXr+2QIDAQABo4H/
MIH8MA4GA1UdDwEB/wQEAwIDqDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUH
AwIwHQYDVR0OBBYEFI1nJPQFFcOexAR+7aBej1/m+nuxMB8GA1UdIwQYMBaAFPmf
5H0nJUUbAJvh77Psmpkcszn9MDsGCCsGAQUFBwEBBC8wLTArBggrBgEFBQcwAoYf
aHR0cDovLzEyNy4wLjAuMTo4MjAwL3YxL3BraS9jYTAbBgNVHREEFDASghB0ZXN0
LmV4YW1wbGUuY29tMDEGA1UdHwQqMCgwJqAkoCKGIGh0dHA6Ly8xMjcuMC4wLjE6
ODIwMC92MS9wa2kvY3JsMA0GCSqGSIb3DQEBCwUAA4IBAQAJtELCl6ljdLix1K5C
SY9kqsgIHXFfwTSEQBbgsdq/3Y2mgVBy++NoEZ7gb9aTgCCvyynPIL9sVcrXoayZ
Za5uYyT0mFJasYJTmWeVTDAnuBNIT3/gN9FJJgHbnnF+UdYBOtlXrc4/CntG7M6J
zDBMtQe59xo3lhqbFQhpDQ+ExFxddGuODRCjaaaAko84SIBBCOcK2S+o7/FTWu3q
6akc6t2eX7x2eeJw+hy6uAeB7rFMmzjE2s/MM0YoEt2PvPMi+GYXb6Umh47X5s38
ewr9wKqIiFZzF6Z+SlJi7jaaUEOH81jgV4q4HT6wJvBgyLKGSniD2Ok5mjUDcCik
RTX4
-----END CERTIFICATE-----
```

Issuing CA

```
-----BEGIN CERTIFICATE-----
MIIDNTCCAh2gAwIBAgIUQhIX9DOX76JA5jpJEwz21pQq2dkwDQYJKoZIhvcNAQEL
BQAwFjEUMBIGA1UEAxMLZXhhbXBsZS5jb20wHhcNMTgwNzI1MDEzMTM0WhcNMTgw
ODI2MDEzMjAzWjAWMRQwEgYDVQQDEwtleGFtcGxlLmNvbTCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAK7C7VoRd3Y2yEIx4ZEVq/BrUjRhUr6VWDB0Rm5A
nuMEYI46AjQiyyZzRSSoLnjJV6WEu/sQhEIE5Q6bYQOJ5sJ0OjLN5W7SPvHrB143
n33LNJKvQ9nOafRSWKiLI6YGtBy0hWoaEYU0qg+kTKP7hUrYqJQykGGkJFTGuVKm
PG6yNYIMBt2jD3SUmO7NwpgsMwOC+HpIj2ZTpZ+VHGCFPIa3gRmeJzd79y96bn+s
hfNN3+YbuqQmmzrtiAYLCLI8yy4LKirgz0BlaDh8LX2uViCVcqiVtOz32bccBvmw
YZmz2Wa3UB2g5+gqnn1FojrGn+ofHCAKR04EeK8mFrqBmhMCAwEAAaN7MHkwDgYD
VR0PAQH/BAQDAgEGMA8GA1UdEwEB/wQFMAMBAf8wHQYDVR0OBBYEFPmf5H0nJUUb
```

The response contains the PEM-encoded private key, key type and certificate serial number.

7. Click **Copy Credentials** and save it to a file.

Revoke Certificates

Overview

If a certificate must be revoked, you can easily perform the revocation action, which causes the CRL to be regenerated. When the CRL is regenerated, any expired certificates are removed from the CRL.

References

- PKI Secrets Engine
- Build Your Own Certificate Authority (CA)

Steps

CLI

1. To revoke a certificate, execute the following command.

```
$ vault write pki_int/revoke serial_number=<serial_number>
```

REST API

1. Invoke the /pki_int/revoke endpoint to invoke a certificate using its serial number.

```
curl --header “X-Vault-Token: <TOKEN>” \
       --request POST \
       --data ‘{“serial_number”:
“48:97:82:dd:f0:d3:d9:7e:53:25:ba:fd:f6:77:3e:89:e5:65:cc:e7”}’ \
       http://127.0.0.1:8200/v1/pki_int/revoke
```

Web UI

1. Select **Secrets**.

2. Select **pki_int** from the **Secrets Engines** list.

3. Select the **Certificates** tab.

4. Select the serial number for the certificate you wish to revoke.

5. Click **Revoke**. At the confirmation, click **Revoke** again.

Remove Expired Certificates

Overview

Keep the storage backend and CRL by periodically removing certificates that have expired and are past a certain buffer period beyond their expiration time.

References

- PKI Secrets Engine
- Build Your Own Certificate Authority (CA)

Steps

CLI

1. To remove revoked certificates and clean the CRL.

```
$ vault write pki_int/tidy tidy_cert_store=true tidy_revoked_certs=true
```

REST API

2. Invoke the /pki_int/tidy endpoint to remove revoked certificate and clean the CRL

```
$ curl --header “X-Vault-Token: <TOKEN>” \
       --request POST \
       --data ‘{“tidy_cert_store”: true, “tidy_revoked_certs”: true}’ \
       http://127.0.0.1:8200/v1/pki_int/tidy
```

Web UI

1. Select **Secrets**.
1. Select **pki_int** from the **Secrets Engines** list.
1. Select **Configure**.
1. Select the **Tidy** tab.
1. Select the check-box for **Tidy the Certificate Store** and **Tidy the Revocation List (CRL)**.
1. Click **Save**.

SSH Login

Overview

The signed SSH certificate is the simplest and most powerful mechanism in terms of setup complexity and being platform agnostic. By leveraging Vault's powerful CA capabilities and functionality built into OpenSSH, clients can SSH into target hosts using their own local SSH keys.

In this section, the term "client" refers to the person or machine performing the SSH operation. The "host" refers to the target machine. If this is confusing, substitute "client" with "user."

Troubleshooting can be found at the end of the Signed SSH Certificates web page.

References

- SSH Secrets Engine
- Signed SSH Certificates
- Manage SSH with HashiCorp Vault

Steps

Configure the SSH Secrets Engine

1. Mount the secrets engine. Like all Secrets Engines in Vault, the SSH secrets engine must be mounted before use.

   ```
   $ vault secrets enable -path=ssh-client-signer ssh
   ```

   This command enables the SSH secrets engine at the path "ssh-client-signer." It is possible to mount the same secrets engine multiple times using different -path arguments. The name "ssh-client-signer" is not special - it can be any name, but this documentation assumes "ssh-client-signer."

2. Configure Vault with a CA for signing client keys using the /config/ca endpoint. If you do not have an internal CA, Vault can generate a keypair for you.

   ```
   $ vault write ssh-client-signer/config/ca generate_signing_key=true
   ```

   If you already have a keypair, specify the public and private key parts as part of the payload:

   ```
   $ vault write ssh-client-signer/config/ca \
       private_key="..." \
       public_key="..."
   ```

   Regardless of whether it is generated or uploaded, the client signer public key is accessible via the API at the /public_key endpoint.

3. Add the public key to all target host's SSH configuration. This process can be manual or automated using a configuration management tool. The public key is accessible via the API and does not require authentication.

   ```
   $ curl -o /etc/ssh/trusted-user-ca-keys.pem http://127.0.0.1:8200/v1/ssh-client-
   signer/public_key
   $ vault read -field=public_key ssh-client-signer/config/ca > /etc/ssh/trusted-user-
   ca-keys.pem
   ```

   Add the path where the public key contents are stored to the SSH configuration file as the

TrustedUserCAKeys option.

```
# /etc/ssh/sshd_config
# ...
TrustedUserCAKeys /etc/ssh/trusted-user-ca-keys.pem
```

Restart the SSH service to pick up the changes.

4. Create a named Vault role for signing client keys.

   Because of the way some SSH certificate features are implemented, options are passed as a map. The following example adds the permit-pty extension to the certificate.

```
$ vault write ssh-client-signer/roles/my-role -<<"EOH"
{
  "allow_user_certificates": true,
  "allowed_users": "*",
  "allowed_extensions": "permit-pty,permit-port-forwarding",
  "default_extensions": [
    {
      "permit-pty": ""
    }
  ],
  "key_type": "ca",
  "default_user": "ubuntu",
  "ttl": "30m0s"
}
EOH
```

Client SSH Authentication

The following steps are performed by the client (user) that wants to authenticate to machines managed by Vault. These commands are usually run from the client's local workstation.

1. Locate or generate the SSH public key. Usually, this is `~/.ssh/id_rsa.pub`. If you do not have an

SSH keypair, generate one:

```
$ ssh-keygen -t rsa -C “user@example.com”
```

2. Ask Vault to sign your public key. This file usually ends in .pub and the contents begin with ssh-rsa...

```
$ vault write ssh-client-signer/sign/my-role \
    public_key=@$HOME/.ssh/id_rsa.pub
```

The result includes the serial and the signed key. This signed key is another public key.

To customize the signing options, use a JSON payload:

```
$ vault write ssh-client-signer/sign/my-role -<<”EOH”
{
  “public_key”: “ssh-rsa AAA...”,
  “valid_principals”: “my-user”,
  “key_id”: “custom-prefix”,
  “extensions”: {
    “permit-pty”: “”,
    “permit-port-forwarding”: “”
  }
}
EOH
```

3. Save the resulting signed public key to disk. Limit permissions as needed.
4. SSH into the host machine using the signed key. You must supply both the signed public key from Vault and the corresponding private key as authentication to the SSH call.

```
$ ssh -i signed-cert.pub -i ~/.ssh/id_rsa username@10.0.23.5
```

Host Key Signing

For added layers of security, we recommend enabling host key signing. This method is used in conjunction with client key signing to provide an additional integrity layer. When enabled, the SSH agent verifies the target host is valid and trusted before attempting to SSH. This reduces the probability of a user accidentally SSHing into an unmanaged or malicious machine.

1. Mount the secrets engine. For the most security, mount at a different path from the client signer.

   ```
   $ vault secrets enable -path=ssh-host-signer ssh
   ```

2. Configure Vault with a CA for signing host keys using the /config/ca endpoint. If you do not have an internal CA, Vault can generate a keypair for you.

   ```
   $ vault write ssh-host-signer/config/ca generate_signing_key=true
   ```

3. If you already have a keypair, specify the public and private key parts as part of the payload:

   ```
   $ vault write ssh-host-signer/config/ca \
       private_key=”...” \
       public_key=”...”
   ```

   Regardless of whether it is generated or uploaded, the host signer public key is accessible via the API at the /public_key endpoint.

4. Extend host key certificate TTLs

   ```
   $ vault secrets tune -max-lease-ttl=87600h ssh-host-signer
   Create a role for signing host keys. Be sure to fill in the list of allowed
   domains, set allow_bare_domains, or both.
   $ vault write ssh-host-signer/roles/hostrole \
       key_type=ca \
       ttl=87600h \
       allow_host_certificates=true \
   ```

```
    allowed_domains=”localdomain,example.com” \
    allow_subdomains=true
```

5. Sign the host’s SSH public key.

```
vault write ssh-host-signer/sign/hostrole \
    cert_type=host \
    public_key=@/etc/ssh/ssh_host_rsa_key.pub
```

6. Set the resulting signed certificate as HostCertificate in the SSH configuration on the host machine.

```
$ vault write -field=signed_key ssh-host-signer/sign/hostrole \
    cert_type=host \
    public_key=@/etc/ssh/ssh_host_rsa_key.pub > /etc/ssh/ssh_host_rsa_key-cert.pub
```

Set permissions on the certificate to be 0640

```
$ chmod 0640 /etc/ssh/ssh_host_rsa_key-cert.pub
```

Add host key and host certificate to the SSH configuration file.

```
# /etc/ssh/sshd_config
# ...

# For client keys
TrustedUserCAKeys /etc/ssh/trusted-user-ca-keys.pem

# For host keys
HostKey /etc/ssh/ssh_host_rsa_key
HostCertificate /etc/ssh/ssh_host_rsa_key-cert.pub
```

Restart the SSH service to pick up the changes.

Client-Side Host Verification

1. Retrieve the host signing CA public key to validate the host signature of target machines.

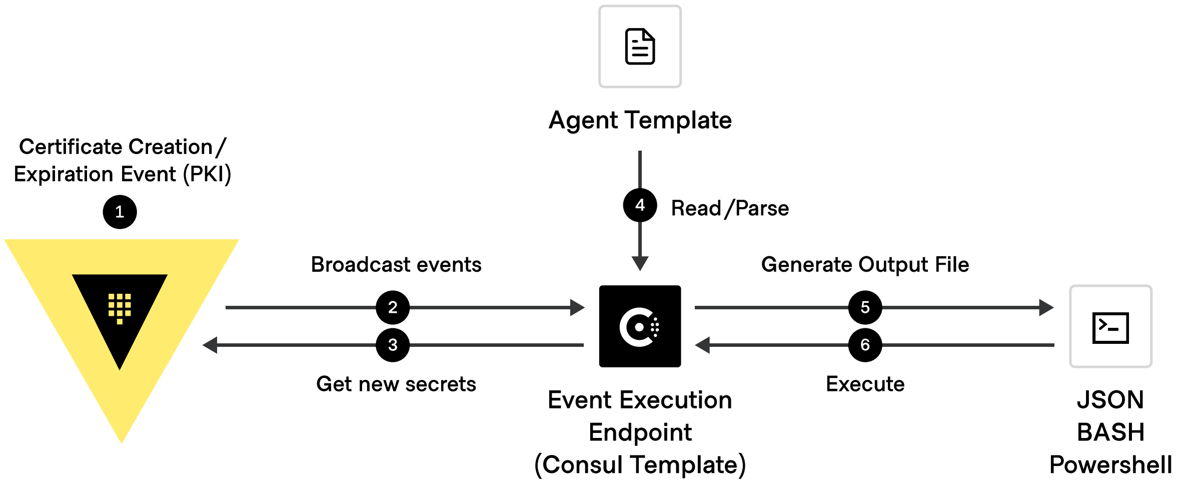
```
$ curl http://127.0.0.1:8200/v1/ssh-host-signer/public_key
$ vault read -field=public_key ssh-host-signer/config/ca
```

2. Add the resulting public key to the known_hosts file with authority.

```
# ~/.ssh/known_hosts
@cert-authority *.example.com ssh-rsa AAAAB3NzaC1yc2EAAA…
```

3. SSH into target machines as usual.

Automate Endpoint Certificate Rotation



Overview

Already you can see how generating PKI certificates with Vault saves operators time. A single call to the Vault API replaces the tedious process of generating a private key, generating a CSR, submitting to a CA, and then waiting for a verification and signing process to complete.

To automate the process further, use a template rendering tool such as Consul Template.

Consul Template is a daemon that queries a Consul or Vault cluster and updates any number of specified templates on the file system. Rendering templates requires both a template file and a template configuration. Template files are written in the Go Template format, and the configuration files are in HCL. (See Consul Template's README.md for further documentation.)

- Some of the most common Vault use cases include:
- Continuously read values from Vault and store them locally
- Continuously renewing Vault auth tokens
- Executing arbitrary commands
- Continuously renewing Dynamic Secrets (PKI, AWS, Database)

References

- X.509 Certificate Management with Vault
- Automating Certificate Lifecycle Management with HashiCorp Vault
- Templating Language
- Certificates Automation with Vault and Consul Template
- F5-certificate-rotate repo
- Consul Template Download

Steps

Deploy Consul Template

1. You can download the latest release from Consul Template here on the target host that requires a certificate.

2. We will use a simple configuration for our Consul Template covering only the Vault integration scope, to create a config.hcl file with the following:

```
vault {
  address = “http://vault-server:8200”
  token=”XXXXXXXX” .  #also can be under VAULT_TOKEN env variable
  grace = “1s”
  unwrap_token = false
  renew_token = false
}
syslog {
  enabled = true
  facility = “LOCAL5”
}
template {
contents=”{{ with secret \”pki/issue/CA\” \”common_name=example.com\” }}{{ .Data.
certificate }}{{ end }}”
destination=”~/example.com.cert”
#Optional Command after certificate renewal
command = “restart service foo”
```

3. Start Consul Template with the -config option set to the configuration file. Consider configuring it as a registered service to execute on host startup if needed.

```
consul-template -config config.hcl
```

4. Check the time validity on the certificate by issuing the following command against the certificate.

```
openssl x509 -text -noout -in example.com.cert
```

HashiCorp

www.hashicorp.com