**[»](https://www.vaultproject.io/docs/internals/architecture.html" \l "architecture) Architecture**

Vault is a complex system that has many different pieces. To help both users and developers of Vault build a mental model of how it works, this page documents the system architecture.

**Advanced Topic!** This page covers technical details of Vault. You don't need to understand these details to effectively use Vault. The details are documented here for those who wish to learn about them without having to go spelunking through the source code. However, if you're an operator of Vault, we recommend learning about the architecture due to the importance of Vault in an environment.

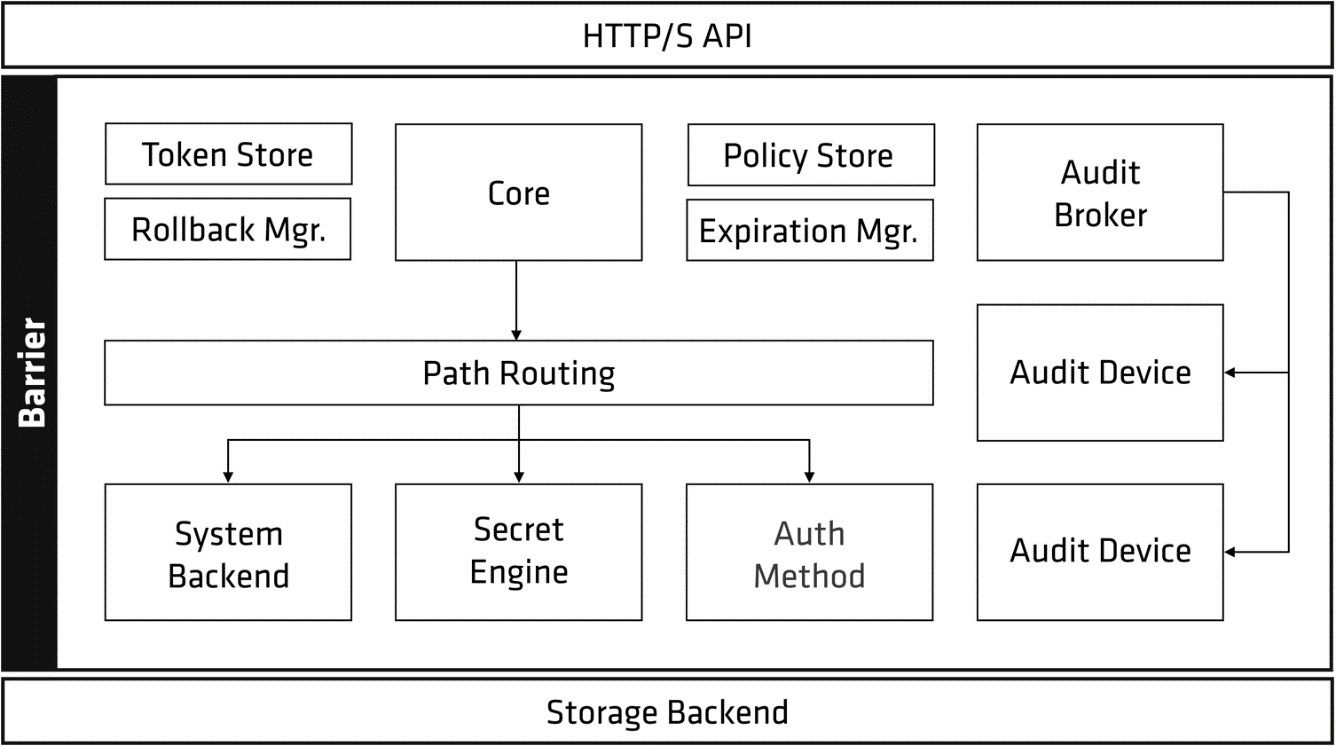
**[»](https://www.vaultproject.io/docs/internals/architecture.html" \l "glossary) Glossary**

Before describing the architecture, we provide a glossary of terms to help clarify what is being discussed:

* **Storage Backend** - A storage backend is responsible for durable storage of *encrypted* data. Backends are not trusted by Vault and are only expected to provide durability. The storage backend is configured when starting the Vault server.
* **Barrier** - The barrier is cryptographic steel and concrete around the Vault. All data that flows between Vault and the storage backend passes through the barrier. The barrier ensures that only encrypted data is written out, and that data is verified and decrypted on the way in. Much like a bank vault, the barrier must be "unsealed" before anything inside can be accessed.
* **Secrets Engine** - A secrets engine is responsible for managing secrets. Simple secrets engines like the "kv" secrets engine simply return the same secret when queried. Some secrets engines support using policies to dynamically generate a secret each time they are queried. This allows for unique secrets to be used which allows Vault to do fine-grained revocation and policy updates. As an example, a MySQL secrets engine could be configured with a "web" policy. When the "web" secret is read, a new MySQL user/password pair will be generated with a limited set of privileges for the web server.
* **Audit Device** - An audit device is responsible for managing audit logs. Every request to Vault and response from Vault goes through the configured audit devices. This provides a simple way to integrate Vault with multiple audit logging destinations of different types.
* **Auth Method** - An auth method is used to authenticate users or applications which are connecting to Vault. Once authenticated, the auth method returns the list of applicable policies which should be applied. Vault takes an authenticated user and returns a client token that can be used for future requests. As an example, the userpass auth method uses a username and password to authenticate the user. Alternatively, the github auth method allows users to authenticate via GitHub.
* **Client Token** - A client token (aka "Vault Token") is conceptually similar to a session cookie on a web site. Once a user authenticates, Vault returns a client token which is used for future requests. The token is used by Vault to verify the identity of the client and to enforce the applicable ACL policies. This token is passed via HTTP headers.
* **Secret** - A secret is the term for anything returned by Vault which contains confidential or cryptographic material. Not everything returned by Vault is a secret, for example system configuration, status information, or policies are not considered secrets. Secrets always have an associated lease. This means clients cannot assume that the secret contents can be used indefinitely. Vault will revoke a secret at the end of the lease, and an operator may intervene to revoke the secret before the lease is over. This contract between Vault and its clients is critical, as it allows for changes in keys and policies without manual intervention.
* **Server** - Vault depends on a long-running instance which operates as a server. The Vault server provides an API which clients interact with and manages the interaction between all the secrets engines, ACL enforcement, and secret lease revocation. Having a server based architecture decouples clients from the security keys and policies, enables centralized audit logging and simplifies administration for operators.

**[»](https://www.vaultproject.io/docs/internals/architecture.html" \l "high-level-overview) High-Level Overview**

A very high level overview of Vault looks like this:

[](https://www.vaultproject.io/img/layers.png)

Let's begin to break down this picture. There is a clear separation of components that are inside or outside of the security barrier. Only the storage backend and the HTTP API are outside, all other components are inside the barrier.

The storage backend is untrusted and is used to durably store encrypted data. When the Vault server is started, it must be provided with a storage backend so that data is available across restarts. The HTTP API similarly must be started by the Vault server on start so that clients can interact with it.

Once started, the Vault is in a *sealed* state. Before any operation can be performed on the Vault it must be unsealed. This is done by providing the unseal keys. When the Vault is initialized it generates an encryption key which is used to protect all the data. That key is protected by a master key. By default, Vault uses a technique known as [Shamir's secret sharing algorithm](https://en.wikipedia.org/wiki/Shamir's_Secret_Sharing) to split the master key into 5 shares, any 3 of which are required to reconstruct the master key.

The number of shares and the minimum threshold required can both be specified. Shamir's technique can be disabled, and the master key used directly for unsealing. Once Vault retrieves the encryption key, it is able to decrypt the data in the storage backend, and enters the *unsealed* state. Once unsealed, Vault loads all of the configured audit devices, auth methods, and secrets engines.

The configuration of those audit devices, auth methods, and secrets engines must be stored in Vault since they are security sensitive. Only users with the correct permissions should be able to modify them, meaning they cannot be specified outside of the barrier. By storing them in Vault, any changes to them are protected by the ACL system and tracked by audit logs.

After the Vault is unsealed, requests can be processed from the HTTP API to the Core. The core is used to manage the flow of requests through the system, enforce ACLs, and ensure audit logging is done.

When a client first connects to Vault, it needs to authenticate. Vault provides configurable auth methods providing flexibility in the authentication mechanism used. Human friendly mechanisms such as username/password or GitHub might be used for operators, while applications may use public/private keys or tokens to authenticate. An authentication request flows through core and into an auth method, which determines if the request is valid and returns a list of associated policies.

Policies are just a named ACL rule. For example, the "root" policy is built-in and permits access to all resources. You can create any number of named policies with fine-grained control over paths. Vault operates exclusively in a whitelist mode, meaning that unless access is explicitly granted via a policy, the action is not allowed. Since a user may have multiple policies associated, an action is allowed if any policy permits it. Policies are stored and managed by an internal policy store. This internal store is manipulated through the system backend, which is always mounted at sys/.

Once authentication takes place and an auth method provides a set of applicable policies, a new client token is generated and managed by the token store. This client token is sent back to the client, and is used to make future requests. This is similar to a cookie sent by a website after a user logs in. The client token may have a lease associated with it depending on the auth method configuration. This means the client token may need to be periodically renewed to avoid invalidation.

Once authenticated, requests are made providing the client token. The token is used to verify the client is authorized and to load the relevant policies. The policies are used to authorize the client request. The request is then routed to the secrets engine, which is processed depending on its type. If the secrets engine returns a secret, the core registers it with the expiration manager and attaches a lease ID. The lease ID is used by clients to renew or revoke their secret. If a client allows the lease to expire, the expiration manager automatically revokes the secret.

The core handles logging of requests and responses to the audit broker, which fans the request out to all the configured audit devices. Outside of the request flow, the core performs certain background activity. Lease management is critical, as it allows expired client tokens or secrets to be revoked automatically. Additionally, Vault handles certain partial failure cases by using write ahead logging with a rollback manager. This is managed transparently within the core and is not user visible.

# Security Model

Due to the nature of Vault and the confidentiality of data it is managing, the Vault security model is very critical. The overall goal of Vault's security model is to provide [confidentiality, integrity, availability, accountability, authentication](https://en.wikipedia.org/wiki/Information_security).

This means that data at rest and in transit must be secure from eavesdropping or tampering. Clients must be appropriately authenticated and authorized to access data or modify policy. All interactions must be auditable and traced uniquely back to the origin entity. The system must be robust against intentional attempts to bypass any of its access controls.

# [»](https://www.vaultproject.io/docs/internals/security.html" \l "threat-model) Threat Model

The following are the various parts of the Vault threat model:

* Eavesdropping on any Vault communication. Client communication with Vault should be secure from eavesdropping as well as communication from Vault to its storage backend.
* Tampering with data at rest or in transit. Any tampering should be detectable and cause Vault to abort processing of the transaction.
* Access to data or controls without authentication or authorization. All requests must be proceeded by the applicable security policies.
* Access to data or controls without accountability. If audit logging is enabled, requests and responses must be logged before the client receives any secret material.
* Confidentiality of stored secrets. Any data that leaves Vault to rest in the storage backend must be safe from eavesdropping. In practice, this means all data at rest must be encrypted.
* Availability of secret material in the face of failure. Vault supports running in a highly available configuration to avoid loss of availability.

The following are not parts of the Vault threat model:

* Protecting against arbitrary control of the storage backend. An attacker that can perform arbitrary operations against the storage backend can undermine security in any number of ways that are difficult or impossible to protect against. As an example, an attacker could delete or corrupt all the contents of the storage backend causing total data loss for Vault. The ability to control reads would allow an attacker to snapshot in a well-known state and rollback state changes if that would be beneficial to them.
* Protecting against the leakage of the existence of secret material. An attacker that can read from the storage backend may observe that secret material exists and is stored, even if it is kept confidential.
* Protecting against memory analysis of a running Vault. If an attacker is able to inspect the memory state of a running Vault instance then the confidentiality of data may be compromised.

# [»](https://www.vaultproject.io/docs/internals/security.html" \l "external-threat-overview) External Threat Overview

Given the architecture of Vault, there are 3 distinct systems we are concerned with for Vault. There is the client, which is speaking to Vault over an API. There is Vault or the server more accurately, which is providing an API and serving requests. Lastly, there is the storage backend, which the server is utilizing to read and write data.

There is no mutual trust between the Vault client and server. Clients use [TLS](https://en.wikipedia.org/wiki/Transport_Layer_Security) to verify the identity of the server and to establish a secure communication channel. Servers require that a client provides a client token for every request which is used to identify the client. A client that does not provide their token is only permitted to make login requests.

The storage backends used by Vault are also untrusted by design. Vault uses a security barrier for all requests made to the backend. The security barrier automatically encrypts all data leaving Vault using a 256-bit [Advanced Encryption Standard (AES)](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) cipher in the [Galois Counter Mode (GCM)](https://en.wikipedia.org/wiki/Galois/Counter_Mode) with 96-bit nonces. The nonce is randomly generated for every encrypted object. When data is read from the security barrier the GCM authentication tag is verified during the decryption process to detect any tampering.

Depending on the backend used, Vault may communicate with the backend over TLS to provide an added layer of security. In some cases, such as a file backend this is not applicable. Because storage backends are untrusted, an eavesdropper would only gain access to encrypted data even if communication with the backend was intercepted.

# [»](https://www.vaultproject.io/docs/internals/security.html" \l "internal-threat-overview) Internal Threat Overview

Within the Vault system, a critical security concern is an attacker attempting to gain access to secret material they are not authorized to. This is an internal threat if the attacker is already permitted some level of access to Vault and is able to authenticate.

When a client first authenticates with Vault, an auth method is used to verify the identity of the client and to return a list of associated ACL policies. This association is configured by operators of Vault ahead of time. For example, GitHub users in the "engineering" team may be mapped to the "engineering" and "ops" Vault policies. Vault then generates a client token which is a randomly generated, serialized value and maps it to the policy list. This client token is then returned to the client.

On each request a client provides this token. Vault then uses it to check that the token is valid and has not been revoked or expired, and generates an ACL based on the associated policies. Vault uses a strict default deny or whitelist enforcement. This means unless an associated policy allows for a given action, it will be denied. Each policy specifies a level of access granted to a path in Vault. When the policies are merged (if multiple policies are associated with a client), the highest access level permitted is used. For example, if the "engineering" policy permits read/update access to the "eng/" path, and the "ops" policy permits read access to the "ops/" path, then the user gets the union of those. Policy is matched using the most specific defined policy, which may be an exact match or the longest-prefix match glob pattern.

Certain operations are only permitted by "root" users, which is a distinguished policy built into Vault. This is similar to the concept of a root user on a Unix system or an Administrator on Windows. Although clients could be provided with root tokens or associated with the root policy, instead Vault supports the notion of "sudo" privilege. As part of a policy, users may be granted "sudo" privileges to certain paths, so that they can still perform security sensitive operations without being granted global root access to Vault.

Lastly, Vault supports using a [Two-man rule](https://en.wikipedia.org/wiki/Two-man_rule) for unsealing using [Shamir's Secret Sharing technique](https://en.wikipedia.org/wiki/Shamir's_Secret_Sharing). When Vault is started, it starts in a sealed state. This means that the encryption key needed to read and write from the storage backend is not yet known. The process of unsealing requires providing the master key so that the encryption key can be retrieved. The risk of distributing the master key is that a single malicious actor with access to it can decrypt the entire Vault. Instead, Shamir's technique allows us to split the master key into multiple shares or parts. The number of shares and the threshold needed is configurable, but by default Vault generates 5 shares, any 3 of which must be provided to reconstruct the master key.

By using a secret sharing technique, we avoid the need to place absolute trust in the holder of the master key, and avoid storing the master key at all. The master key is only retrievable by reconstructing the shares. The shares are not useful for making any requests to Vault, and can only be used for unsealing. Once unsealed the standard ACL mechanisms are used for all requests.

To make an analogy, a bank puts security deposit boxes inside of a vault. Each security deposit box has a key, while the vault door has both a combination and a key. The vault is encased in steel and concrete so that the door is the only practical entrance. The analogy to Vault, is that the cryptosystem is the steel and concrete protecting the data. While you could tunnel through the concrete or brute force the encryption keys, it would be prohibitively time consuming. Opening the bank vault requires two-factors: the key and the combination. Similarly, Vault requires multiple shares be provided to reconstruct the master key. Once unsealed, each security deposit boxes still requires the owner provide a key, and similarly the Vault ACL system protects all the secrets stored.

# [»](https://www.vaultproject.io/docs/internals/telemetry.html" \l "telemetry) Telemetry

<https://www.vaultproject.io/docs/internals/telemetry.html>

The Vault server process collects various runtime metrics about the performance of different libraries and subsystems. These metrics are aggregated on a ten second interval and are retained for one minute.

To view the raw data, you must send a signal to the Vault process: on Unix-style operating systems, this is USR1 while on Windows it is BREAK. When the Vault process receives this signal it will dump the current telemetry information to the process's stderr.

This telemetry information can be used for debugging or otherwise getting a better view of what Vault is doing.

Telemetry information can also be streamed directly from Vault to a range of metrics aggregation solutions as described in the [telemetry Stanza documentation](https://www.vaultproject.io/docs/configuration/telemetry.html).

# [»](https://www.vaultproject.io/docs/internals/rotation.html" \l "key-rotation) Key Rotation

Vault has multiple encryption keys that are used for various purposes. These keys support rotation so that they can be periodically changed or in response to a potential leak or compromise. It is useful to first understand the [high-level architecture](https://www.vaultproject.io/docs/internals/architecture.html) before learning about key rotation.

As a review, Vault starts in a sealed state. Vault is unsealed by providing the unseal keys. By default, Vault uses a technique known as [Shamir's secret sharing algorithm](https://en.wikipedia.org/wiki/Shamir's_Secret_Sharing) to split the master key into 5 shares, any 3 of which are required to reconstruct the master key. The master key is used to protect the encryption key, which is ultimately used to protect data written to the storage backend.

To support key rotation, we need to support changing the unseal keys, master key, and the backend encryption key. We split this into two separate operations, rekey and rotate.

The rekey operation is used to generate a **new master key**. When this is being done, it is possible to change the parameters of the key splitting, so that the number of shares and the threshold required to unseal can be changed. To perform a rekey a threshold of the current unseal keys must be provided. This is to prevent a single malicious operator from performing a rekey and invalidating the existing master key.

Performing a rekey is fairly straightforward. The rekey operation must be initialized with the new parameters for the split and threshold. Once initialized, the current unseal keys must be provided until the threshold is met. Once met, Vault will generate the new master key, perform the splitting, and re-encrypt the encryption key with the new master key. The new unseal keys are then provided to the operator, and the old unseal keys are no longer usable.

The rotate operation is used to **change the encryption key** used to protect data written to the storage backend. This key is never provided or visible to operators, who only have unseal keys. This simplifies the rotation, as it does not require the current key holders unlike the rekey operation. When rotate is triggered, a new encryption key is generated and added to a keyring. All new values written to the storage backend are encrypted with the new key. Old values written with previous encryption keys can still be decrypted since older keys are saved in the keyring. This allows key rotation to be done online, without an expensive re-encryption process.

Both the rekey and rotate operations can be done online and in a highly available configuration. Only the active Vault instance can perform either of the operations but standby instances can still assume an active role after either operation. This is done by providing an online upgrade path for standby instances. If the current encryption key is N and a rotation installs N+1, Vault creates a special "upgrade" key, which provides the N+1 encryption key protected by the N key. This upgrade key is only available for a few minutes enabling standby instances to do a periodic check for upgrades. This allows standby instances to update their keys and stay in-sync with the active Vault without requiring operators to perform another unseal.