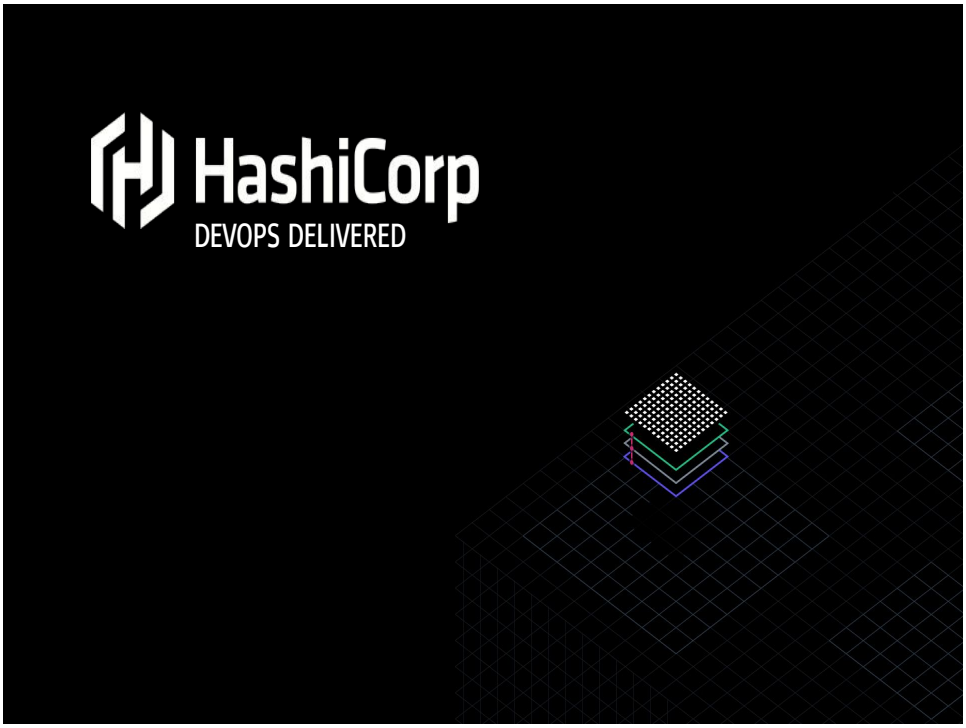
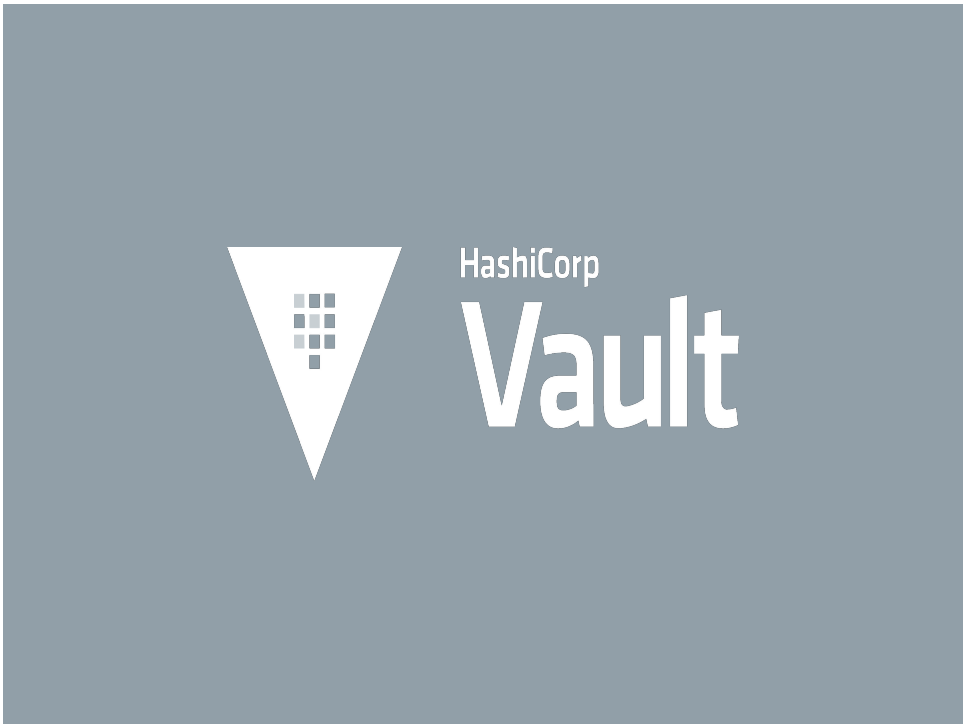

HashiCorp
DEVOPS DELIVERED



This section will deal entirely with Vault.

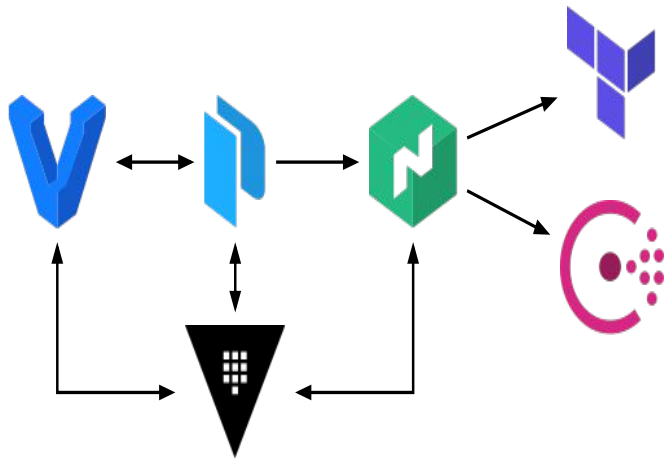
Agenda

1. Glossary and Architecture
2. Static and Generic Secrets
3. Policies and Policy Workflow
4. Dynamic Secrets
5. Authentication, Auditing, and Lease Model
6. Operationalizing Vault
7. HTTP API
8. Direct Application Integration

HashiCorp

Here is our tentative agenda for today.

Vault Manages Secure Information



Vault is the central place for storing and managing secret information in the HashiCorp-managed modern data center.

In order to talk about Vault, it is helpful to talk about the state of secret management before Vault.

Secret sprawl - there are many different secrets, api keys, credentials, and they are spread across many users systems. *Instructor note: I usually ask the room to raise their hand if they have a production credential on their laptop right now.*

Decentralized keys - closely related to secret sprawl, the lack of a centralized source and distribution center for keys makes it very challenging for organizations.

Limited Visibility - due to secret sprawl and limited visibility, it's challenging for organizations to understand the use and impact of a secret.

Break Glass Procedures - there was no clearly-defined "break-glass" procedure - what does the organization do when they detect an intrusion; how do you stop the bleeding?

Post-Vault World

Single source for secrets

Programatic access (automation)

Operation access (manual)

Practical security

Modern data-center friendly (no hardware reqs.)

HashiCorp

Vault attempts to address these problems by providing a single source for secrets with programatic access for machines and manual access for operators. It uses practical security that is modern data center-friendly.

Glossary

HashiCorp

Glossary

Storage backend

The storage backend is responsible for durable storage of encrypted data. There is only one storage backend per Vault cluster.

Data is encrypted in transit and at rest with 256bit AES.

Examples: *in-mem*, *file*, *consul*, and *postgresql*

Glossary

Barrier

The barrier is a cryptographic seal around the Vault. All data that flows between Vault and the storage backend passes through the barrier.

Glossary

Secret backend

A secret backend is responsible for managing secrets. Some secret backends behave like encrypted key-value stores, while others dynamically generate secrets when queried. There can be multiple secret backends in a Vault cluster.

Examples: *pki, generic, transit, postgresql*

Glossary

Secret backend

Secret backends can perform almost any function, not just return static data or hand out credentials.

PKI – Acts as a full CA, leveraging Vault's auth

Transit – Allows round-tripping data through Vault for "encryption as a service", without ever divulging the key

Glossary

Audit backend

An audit backend is responsible for managing audit logs. There can be multiple audit backends in a Vault cluster. Example audit logs include *file* and *syslog*.

Glossary

Auth backend

An auth backend is a credential-based backend that can be used as a way to authenticate humans or machines against Vault.

Machine-oriented: *approle*, *tls*, *tokens*
Operator-oriented: *github*, *ldap*, *userpass*

Glossary

Vault token

A vault token is a conceptually similar to a session cookie on a website. Once a user authenticates via an auth backend, Vault returns a token which is to be used for future requests.

Example: dc57a797-fc99-05d1-6878-f731206b1717

Glossary

Secret

A secret is anything stored or returned by Vault that contains confidential material.

A secret is anything that, if acquired by an unauthorized party, would cause political, financial, or appearance harm to an organization.

Glossary

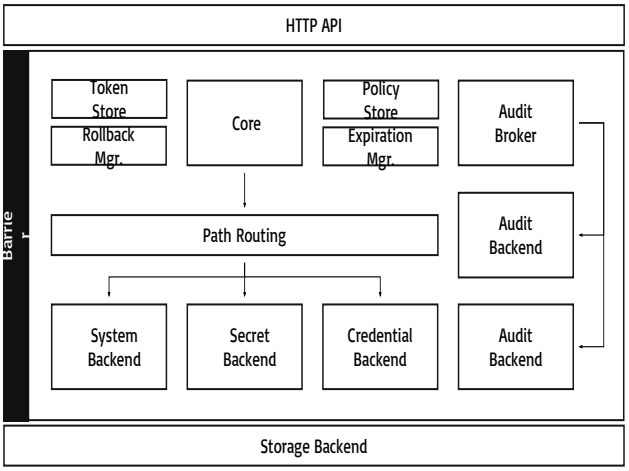
Server

The Vault server provides an HTTP API which clients interact with and manages the interaction between all the backends, ACL enforcement, and secret lease revocation.

Architecture

HashiCorp

Architecture

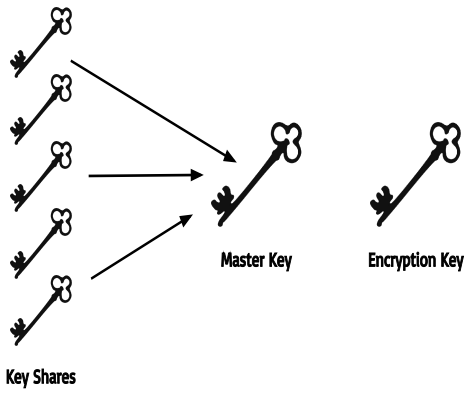


Let's begin to break down this picture. There is a clear separation of components that are inside or outside of the security barrier. Only the storage backend and the HTTP API are outside, all other components are inside the barrier.

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.

Shamir’s secret sharing



HashiCorp

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 to split the master key into 5 shares, any 3 of which are required to reconstruct the master key.

Summary

Solves the "secret sprawl" problem

Protects against external threats (cryptosystem)

Protects against internal threats (ACLs and secret sharing)

Using Generic Secrets

HashiCorp

Exercise: Launch Training Environment

Install Prerequisites:

- VirtualBox https://www.virtualbox.org/wiki/Downloads
- Vagrant https://www.vagrantup.com/downloads.html
- Git https://git-scm.com/downloads

Clone vault repository:

```
git clone https://github.com/chrismatteson/vault-101
```

Launch vagrant environment:

```
cd vault-101
vagrant up
```

HashiCorp

Exercise: Launch Training Environment

View Consul UI for Health Checks:

https://192.168.50.151:8500

View Vault UI:

https://192.168.50.151:8200

Connect to Vault:

```
export VAULT_ADDR=http://192.168.50.151:8200
vault status
```

HashiCorp

Here is a sample answer. Notice that the Vault is unsealed, has 1 keyshare, and 1 key threshold. These concepts will be discussed in more detail throughout the course.

Exercise: Initialize Vault

Initialize Vault:

```
vault init
```

HashiCorp

Here is a sample answer. Notice that the Vault is unsealed, has 1 keyshare, and 1 key threshold. These concepts will be discussed in more detail throughout the course.

Exercise: Unseal Vault

Unseal:

```
vault unseal
<enter one key>
vault unseal
<enter different key>
vault unseal
<enter different key>
```

View Status:

```
vault status
```

HashiCorp

Here is a sample answer. Notice that the Vault is unsealed, has 1 keyshare, and 1 key threshold. These concepts will be discussed in more detail throughout the course.

Generic Secret Backend

The **generic** secret backend is mounted by default and cannot be disabled.

Behaves like encrypted redis or memcached.

Lives at the `secret/` endpoint.

Exercise: Read Generic Secret

Attempt to read the secret at `secret/training`.

HINT: You can use Vault's help documentation

ANOTHER HINT: You'll get an error

HashiCorp

You should get a response like this - "missing client token". This means we have not properly authenticated to the Vault server. Let's fix that now.

Authentication

Most interactions with Vault require a token.

Tokens are generated via authentication.

Authentication is covered in more detail in a later section.

Information is persisted by the local client (you do not need to re-authenticate before each command).

Authenticating as Root

Authenticating as the root user is bad practice.

For the purpose of training, we will start slightly insecure and move to a more secure workflow.

The root token is usually used to setup policy and initial set of users, but then is discarded.

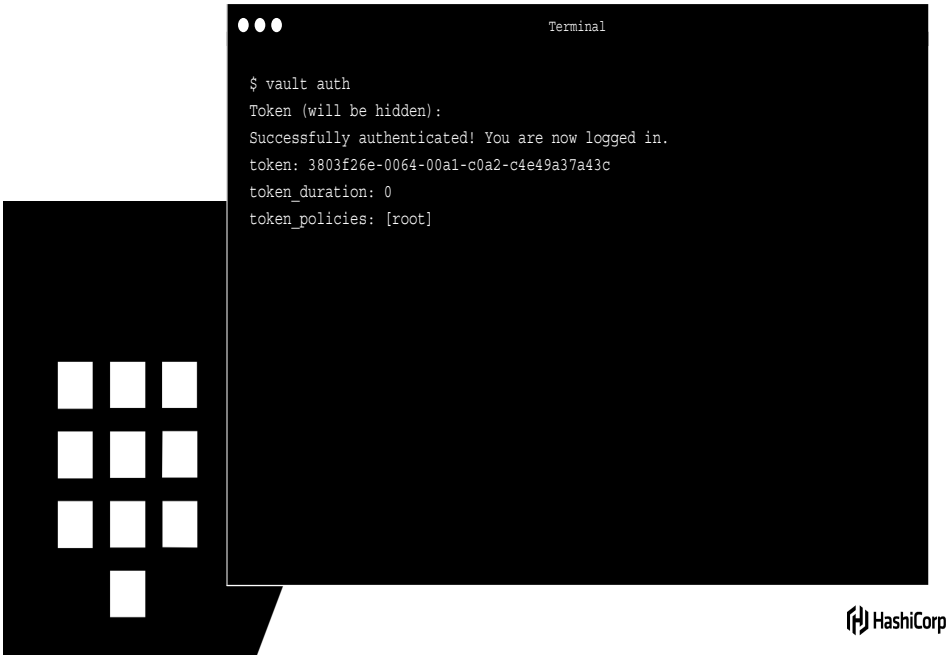
Authenticate as root to continue.

Exercise: Authenticate Vault

Authenticate:

```
vault auth
<enter root authentication token>
```

HashiCorp



Here is a sample answer. Notice that the Vault is unsealed, has 1 keyshare, and 1 key threshold. These concepts will be discussed in more detail throughout the course.

Vault Enterprise Licensing

Vault Enterprise uses a special binary and license file which is written to sys/license. If a license is not installed, vault will stop operating after 30 minutes. **If that happens** before the license is installed, connect into the vm and restart the service:

```
vagrant ssh vault1
sudo service vault restart
exit
```

HashiCorp

Exercise: Install License

Authenticate:

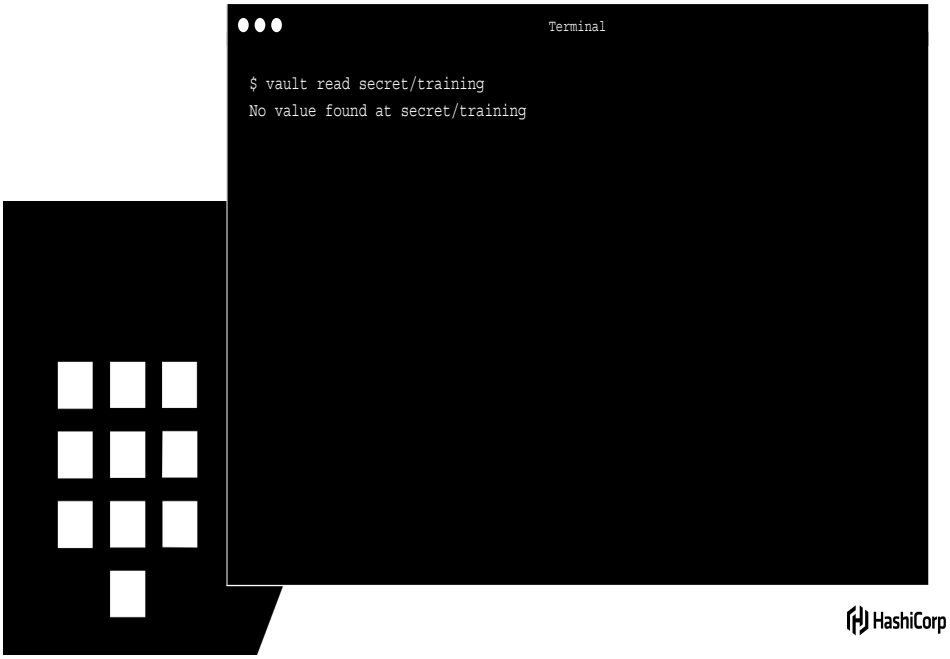
```
vault write sys/license text=`cat <license file>`
```

HashiCorp

Here is a sample answer. Notice that the Vault is unsealed, has 1 keyshare, and 1 key threshold. These concepts will be discussed in more detail throughout the course.

Exercise: Read Generic Secret (again)

Now that we have authenticated, attempt to read the secret at `secret/training` again.



We should get an error when trying to read this path. This makes sense because we have not written a value to that path. Let's do that now.

Exercise: Write Generic Secret

Write a value into `secret/training`.

HINT: Data is expressed as key=value pairs on the CLI

HashiCorp

Everything after the path is a key-value pair to write to the secret backend. You can specify multiple values. If the value has a space, you need to surround it with quotes. Having keys with spaces is permitted, but strongly discouraged because it can lead to unexpected client-side behavior.

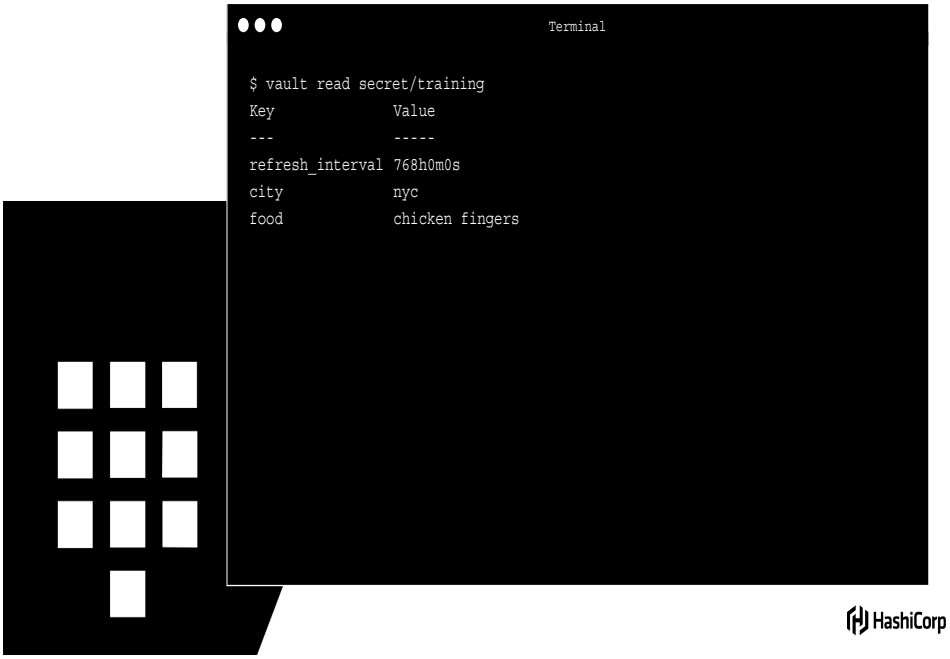
Assuming everything worked, you should see the success output. Let's try

reading again.

Exercise: Retrieve Secret

Read the value of the secret you just stored in `secret/training`.

HashiCorp



First, notice that response includes a refresh_interval key. If this was a dynamic lease, it would include a lease_id and lease_duration instead.

The refresh_interval is a hint to readers (clients) of this secret when to check for a new value. The generic secret backend does not enforce leases.

The output is optimized for grep, but it's also possible to query a specific value using the -field flag.

There is the sample result. You may notice that no newline is returned for the value (represented by the % above).

As a critical thinking exercise, what will the contents of the secret contain after we run this command?

Notice that the value of "name" has disappeared. This is very important to understand - the generic secret backend does not merge or add values. If you want to add/update a key, you must specify all the existing keys as well or data loss can occur!

Spend a few minutes creating more secrets on your own under different keys.

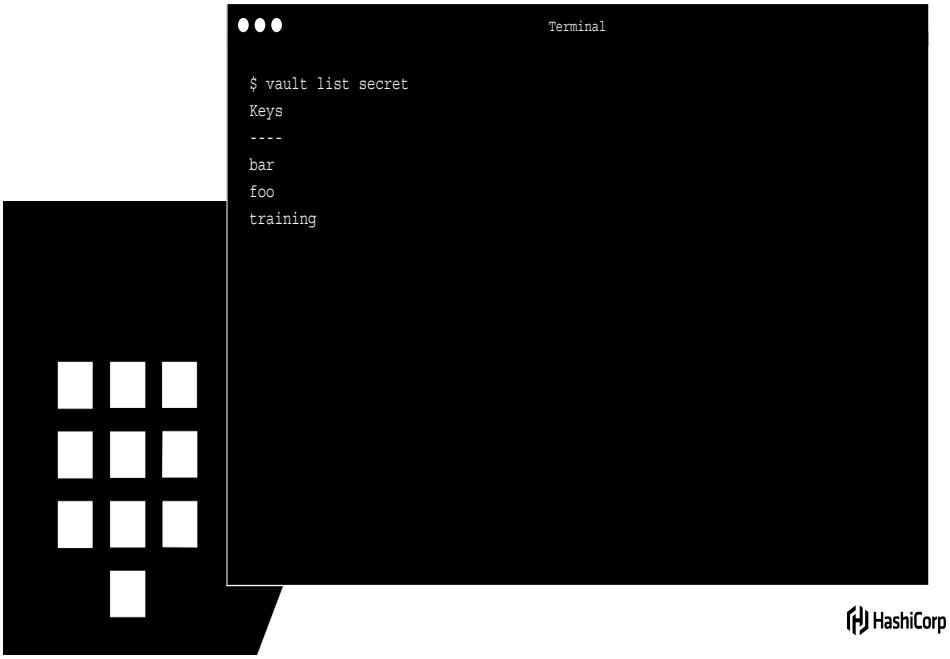
Instructor note: Spend 3-5 minutes. Ask questions.

Exercise: List Secrets

List all the secret keys stored in the generic secret backend.

HINT: Just the keys, not the values.

HashiCorp



Now that you have created a few keys, let's list them.

Notice that these keys always come back in alphabetical order, not insertion order. Additionally, the response object is only the name of the keys, not the values themselves.

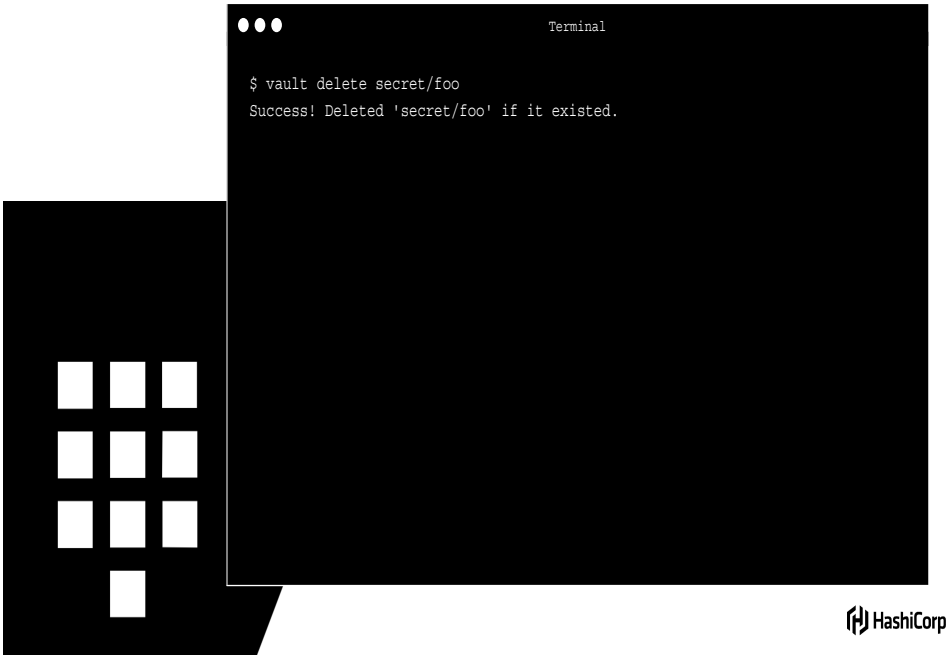
Listing all keys and their values is an N+1 operation.

Exercise: Delete Secret

Delete one of the secrets you just created.

Do **NOT** delete the training key.

HashiCorp



Now that we are done using our training secret, let's go ahead and delete it.

If you mistype a key, you'll still get a success - why?

Getting Help

HashiCorp

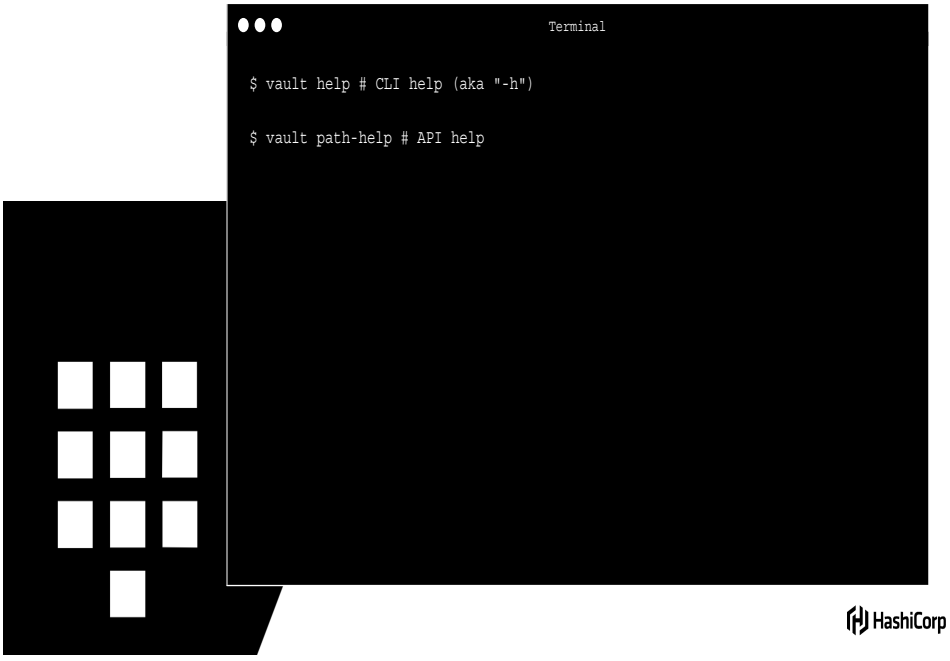
These exercises will gradually build on each other. While some may seem easy, you will need the knowledge from these exercises for future ones, so please pay close attention.

Getting Help

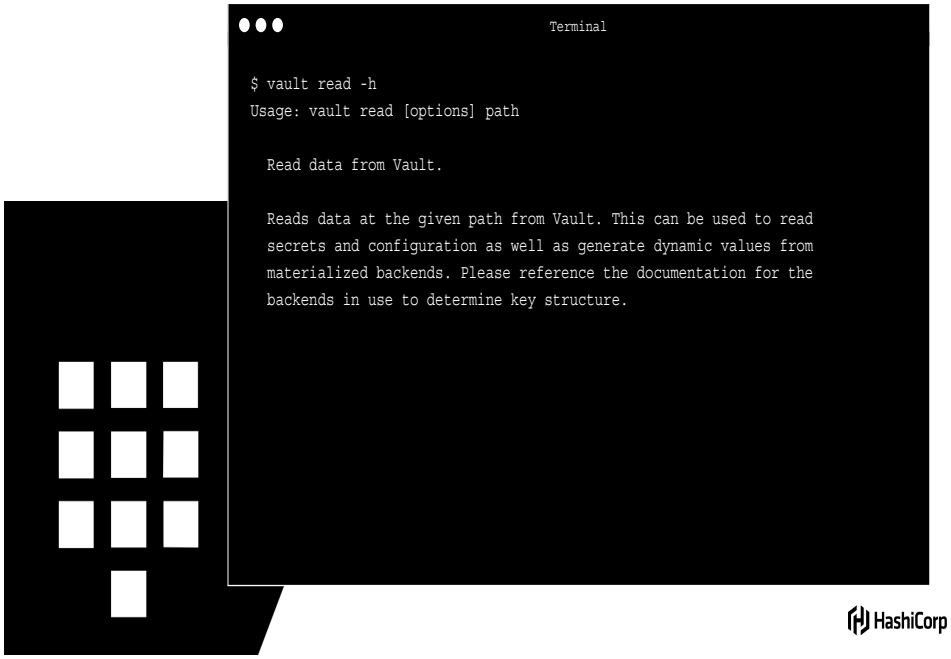
There are two primary ways to get help in Vault:

- CLI help (vault -h)
- API help (vault path-help)

HashiCorp



Here you can see those two options.

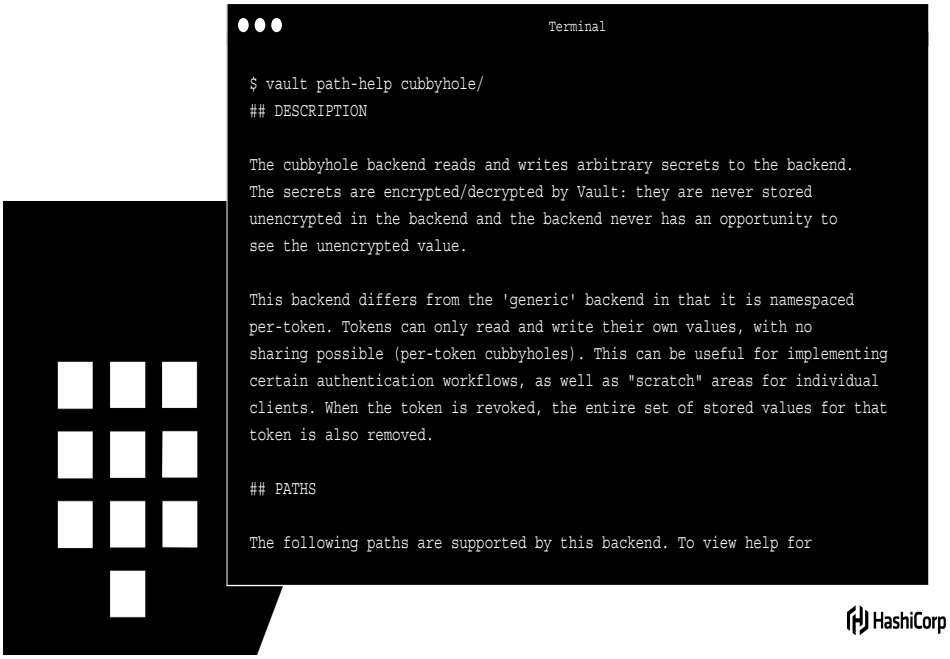

Terminal
$ vault read -h
Usage: vault read [options] path
Read data from Vault.
Reads data at the given path from Vault. This can be used to read
secrets and configuration as well as generate dynamic values from
materialized backends. Please reference the documentation for the
backends in use to determine key structure.
HashiCorp

Path-help provides specific help information about the given path. This is especially useful for configuring or mounting new paths.

Exercise: Using Help

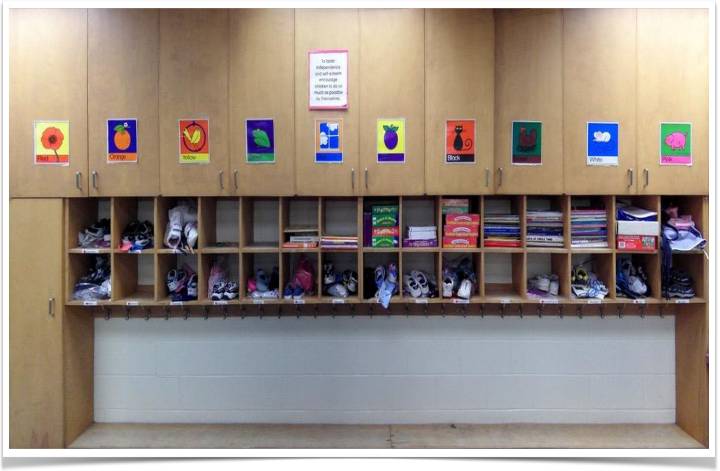
List help information for the HTTP API cubbyhole backend

HashiCorp



Path-help provides specific help information about the given path. This is especially useful for configuring or mounting new paths.

About: Cubbyhole



HashiCorp

The term cubbyhole comes from an Americanism where you get a "locker" or "safe place" to store your belongings/valuables. This is called a "cubbyhole".

In Vault, cubbyhole is your "locker"; all secrets are namespaced under your token. If that token expires or is revoked, all the secrets in its cubbyhole are revoked as well.

It is not possible to reach into another token's cubbyhole, even as the root user.

Setting Policy

HashiCorp

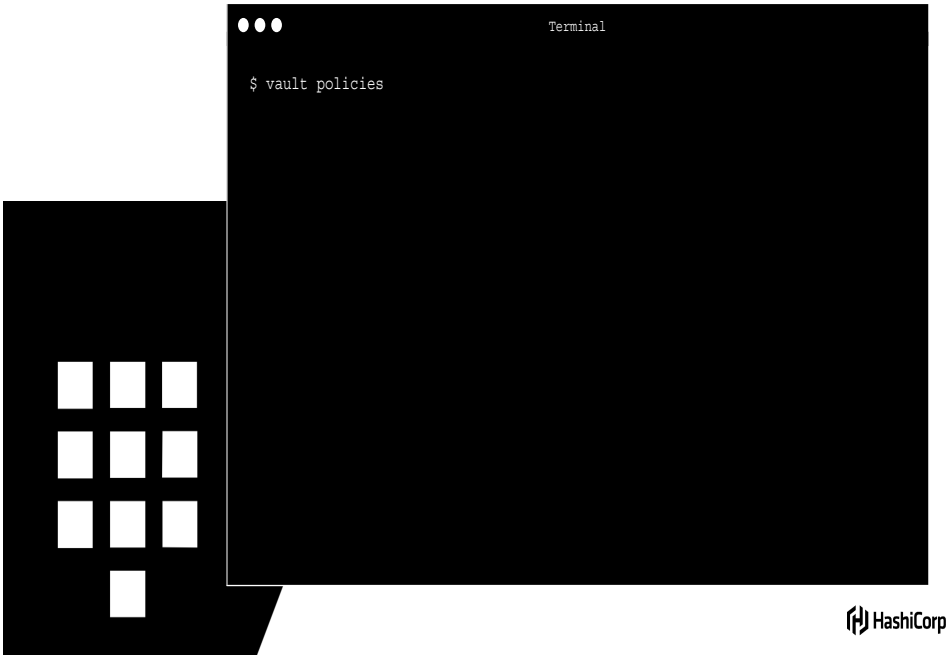
Access Control Policies (ACLs)

"root" policy is created by default – superuser with all permissions.

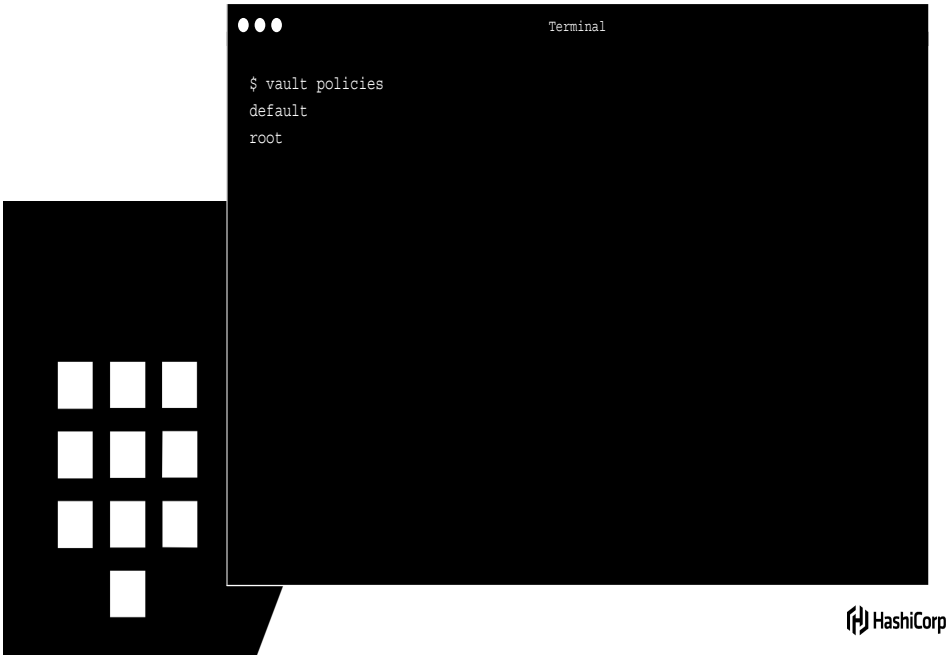
"default" policy is created by default - common permissions.

Policies are written in HashiCorp Configuration Language (HCL), which is a human-friendly config format.

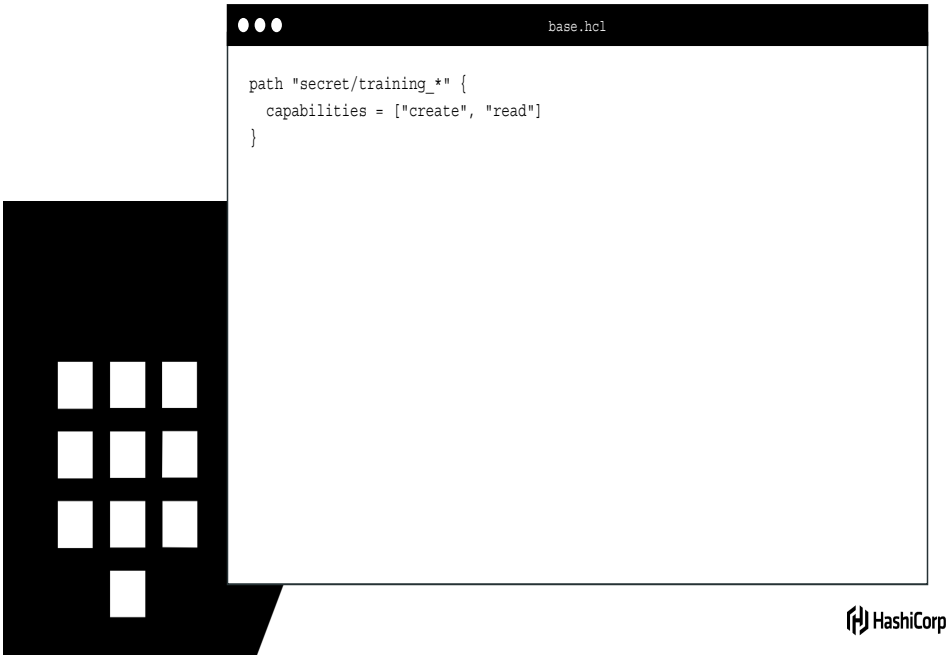
Deny by default (no policy = no authorization).



First, let's list the built-in policies.



Notice there are two built-in policies, default (which is deny all) and root (which is allow all).



Next, let's create a new policy. Here is an example policy that gives access to read and create secrets in the generic secret backend. All other operations are denied.

Notice the "splat" operator. This is a wildcard character that can be specified at the end of a path to allow for namespacing.

We are saving this as "base.hcl". Resources can have multiple policies.

There are a few ways to interact with the policy engine. There are top-level CLI commands (which are shown at the top of the terminal), but also API-level commands. **These commands are equivalent**. The CLI commands provide additional sugar and formatting, but they increase the barrier to entry later when we start authoring our own policies.

Let's write this policy to Vault. Note that no resources are currently tied to this policy.

You should see a success message. If you get an error, it is most likely a syntax error, so double check your syntax and try again.

Again, both options here are correct. This is just introducing students to the sys/ backend very subtly.

Now we can see this policy is returned in the list of policies. Again, it has not yet been applied to any resources.

We can inspect the default policy to see what the default permissions are.

Now we need to assign this policy to a token. First we are going to do this manually by specifying the policy when we create a token.

You should see a new token. This is similar to the root token from earlier, but it is attached to this new policy.

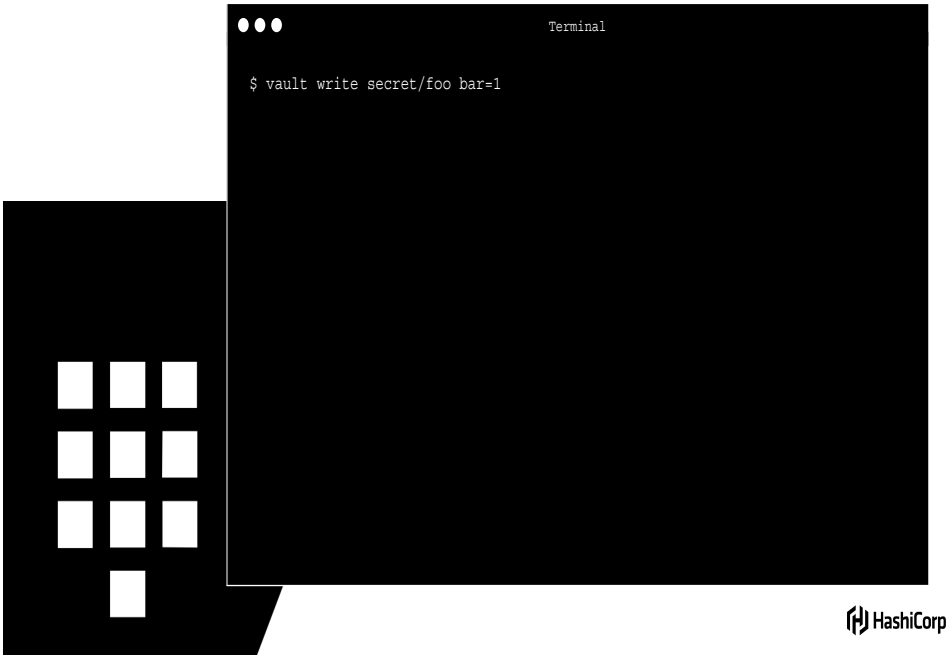
The token_accessor is like the pointer to the token - it can be used to revoke or lookup information about the token.

We are going to authenticate with this token temporarily to check our policy.

We can test the policy is working by "logging in" (authenticating) as this token and trying some commands.

First, let's try to list policies with this token

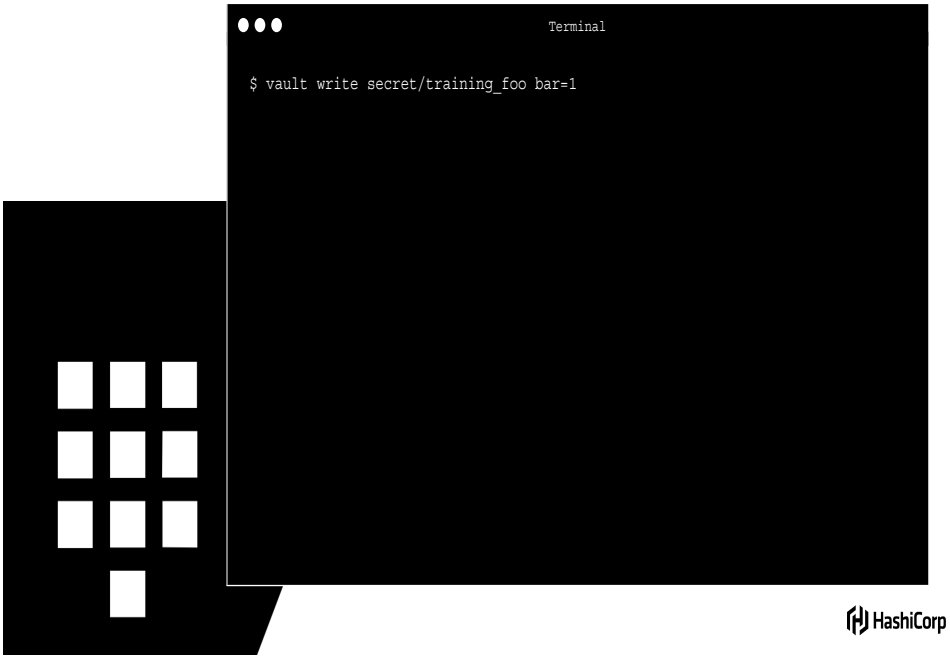
Notice that we get a 403 back from the API with an error "permission denied". This is expected because the policy does not grant access to list policies and everything is deny by default.



Now let's try writing data into the generic secret backend.

Again we get permission denied, why?

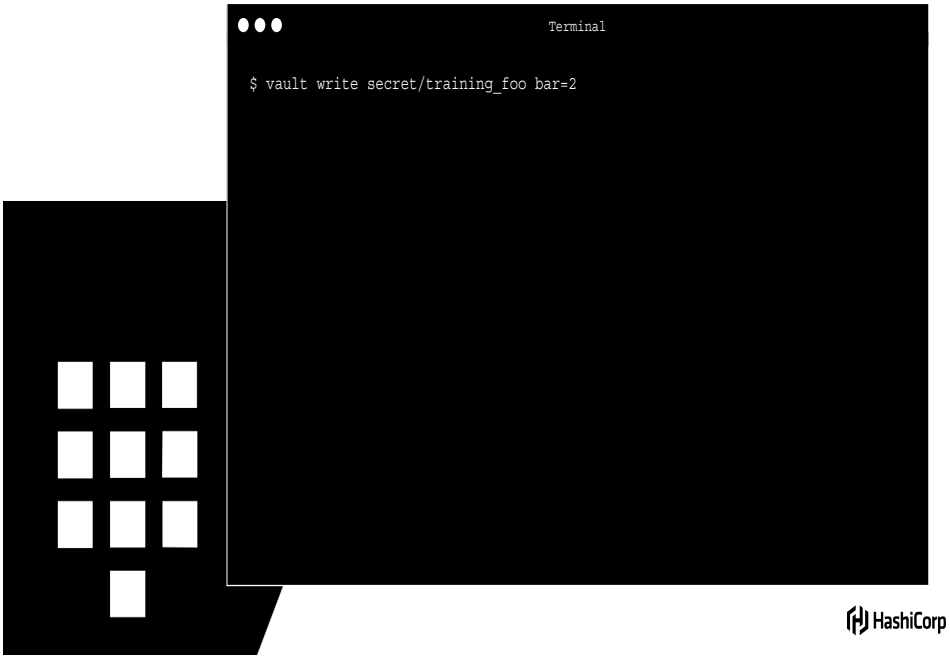
It's because we are only allowed to write keys that being with the certain prefix.



If we try to write to the proper path that matches the namespace...

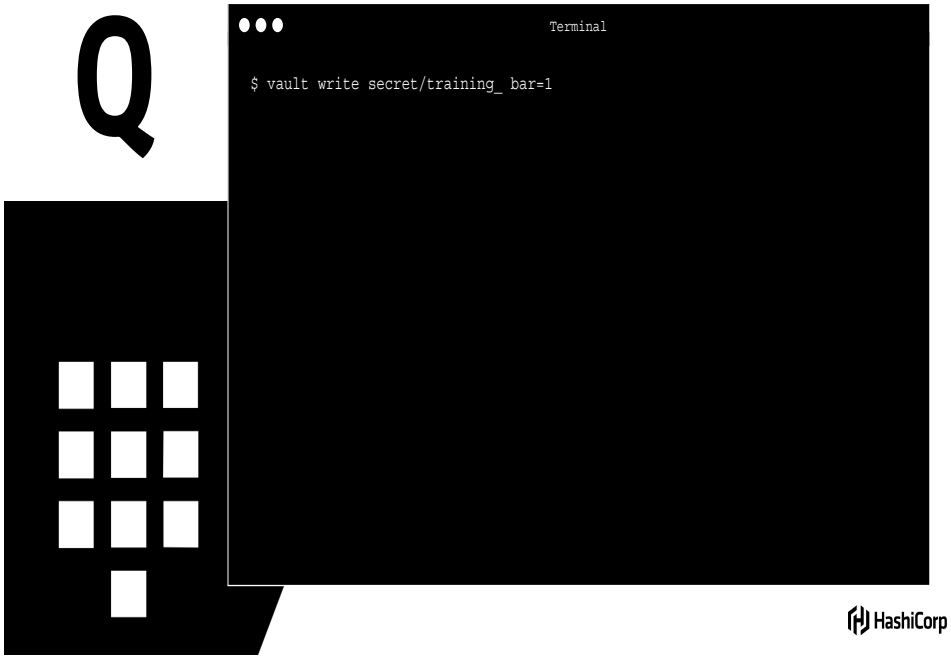

Terminal
$ vault write secret/training_foo bar=1
Success! Data written to: secret/training_foo
HashiCorp

It will work!



Now let's try to update that value to something different

Notice that we get an error. This is because we only gave the policy "create" and "read" capability on this path, but not "update".



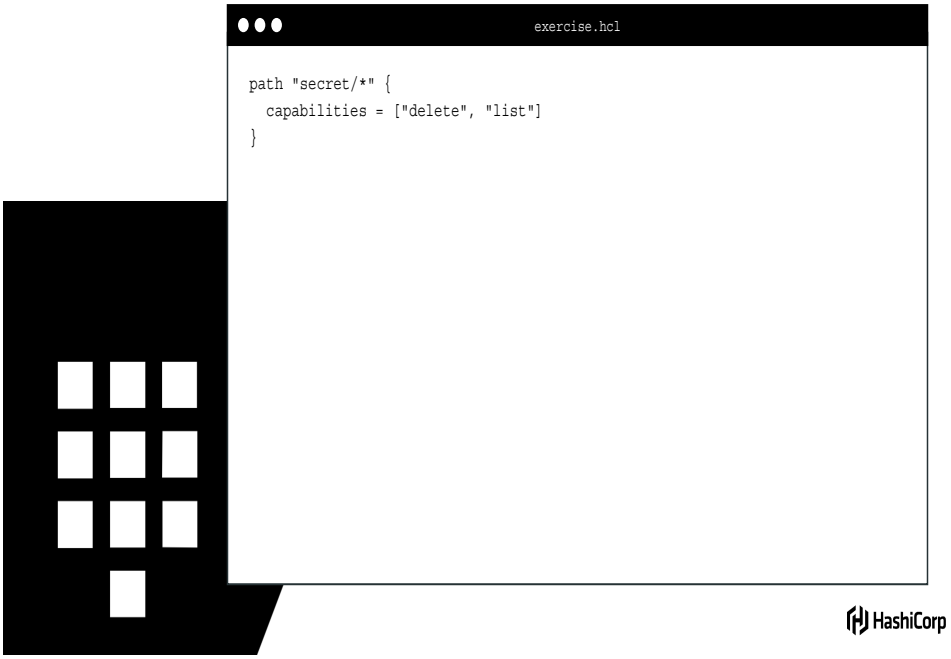
Don't do this yet - will this work?

Yes - this is going to work, but it's **not** because the path is a regular expression. Vault's paths use a radix tree, and that "*" can only come at the end. It matches zero or more characters, but not because of a regex.

Exercise: Create a Policy

Write a policy named "exercise" that permits listing and deleting anything in the generic secret backend, but forbids creating, reading, or updating a secret. **Do not upload the policy.**

```
$ vault list secret/      # ok
$ vault delete secret/foo # ok
$ vault read secret/foo   # 403
$ vault write secret/foo  # 403
```



Here is a possible solution to the problem. Because Vault is deny by default, we do not need to explicitly deny access to the read/write capabilities.

Exercise: Re-auth as root

Re-authenticate as root

(Our current user does not have enough permission)

HashiCorp

Here is a sample solution.

Dynamic Secrets

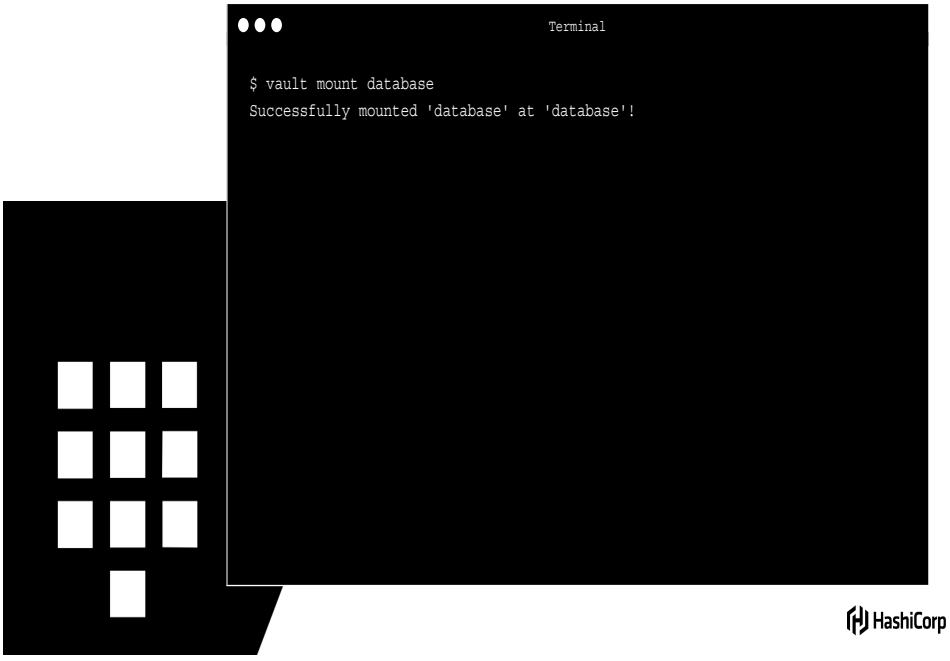
HashiCorp

Secret Backends

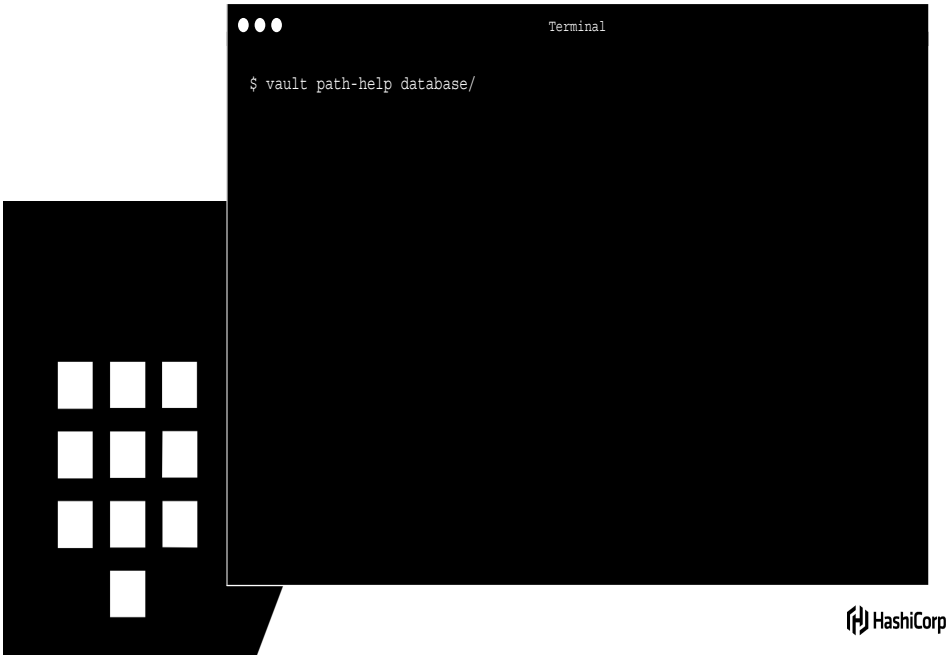
Most secret backends must be mounted before use.

Many secret backends require additional configuration before use.

HashiCorp



First, we need to mount the postgresql dynamic secret backend. Similar to filesystem mounts, dynamic secret backends must be mounted and configured before use. This makes sense, because we have to configure Vault to talk to the proper PostgreSQL instance with the proper credentials.



Now that we have mounted the backend, we can ask for help to configure this. Use the path-help command to get the help output.

Now that we have mounted the backend, we can ask for help to configure this. Use the path-help command to get the help output.


Terminal
$ vault write database/config/postgresql \
plugin_name=postgresql-database-plugin \
allowed_roles=readonly \
connection_url=postgresql://postgres@192.168.50.154/postgres?sslmode=disable

HashiCorp

First, we must configure the PostgreSQL backend with our root credentials. It is very common to give Vault the root credentials and let Vault manage the auditing and lifecycle credentials; it's much better than having one person manage the credentials.

If we ask for help on the configure endpoint, we see the list of configurable parameters. Let's use that to setup our

connection string.

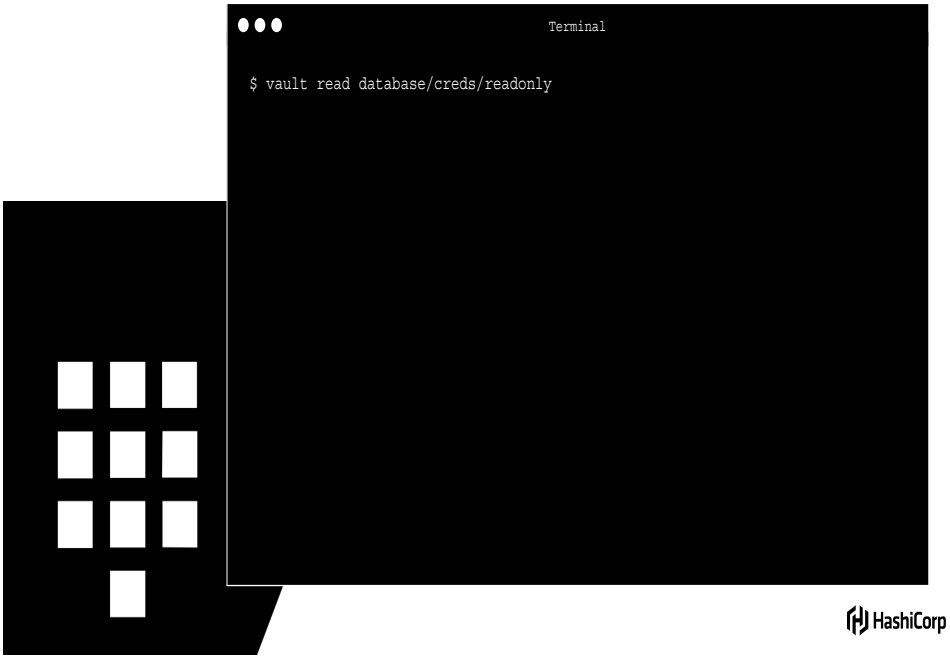
The next step is to configure a role. A role is a logical name that maps to a policy used to generate credentials. Here we are defining a readonly role.

Vault does not know what kind of postgres user you want to create, so you supply it with the SQL to run to create the user.

Since this is a Vault course, not a SQL course, we've added the SQL on the host. You can see the script here.

The values between the `{{ }}` will be filled in by Vault. Notice that we are using the VALID UNTIL clause. This tells PostgreSQL to revoke the credentials, even if Vault is offline or unable to communicate with it.

Upload the contents of this sql file using the `@` syntax.



To generate a new set of credentials, we simply read from the role endpoint.

The output should look something like this. Obviously your values will be different. The username and password values were dynamically generated by Vault with the readonly permissions.

Copy this lease_id to your clipboard - we will use it in a bit.

We can check that these users were created by logging in as the postgres user and listing all accounts.

Notice the VALID UNTIL clause in the Attributes section (should be 1h from now UTC time). Even if an attacker is able to DDoS Vault or take it offline, Postgres will still revoke the credential. Vault will ensure it's deleted. When backends support this expiration, Vault

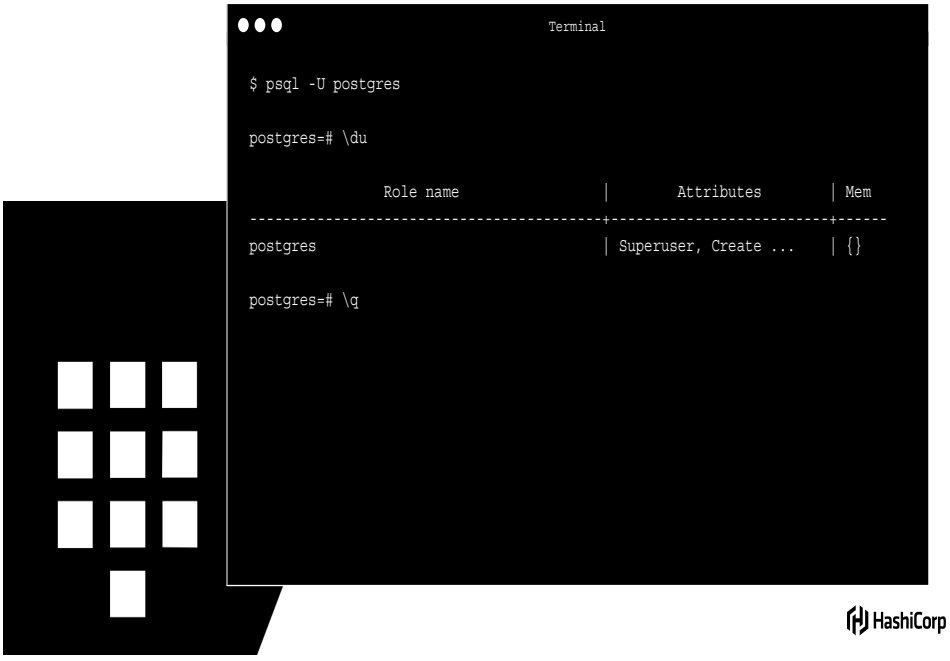
will take advantage of it.

Now let's renew the lease for this credential. A renew, if successful, resets the duration counter back to lease_duration.

You should see output like this, indicating the renewal was successful.

Now let's revoke this credential.

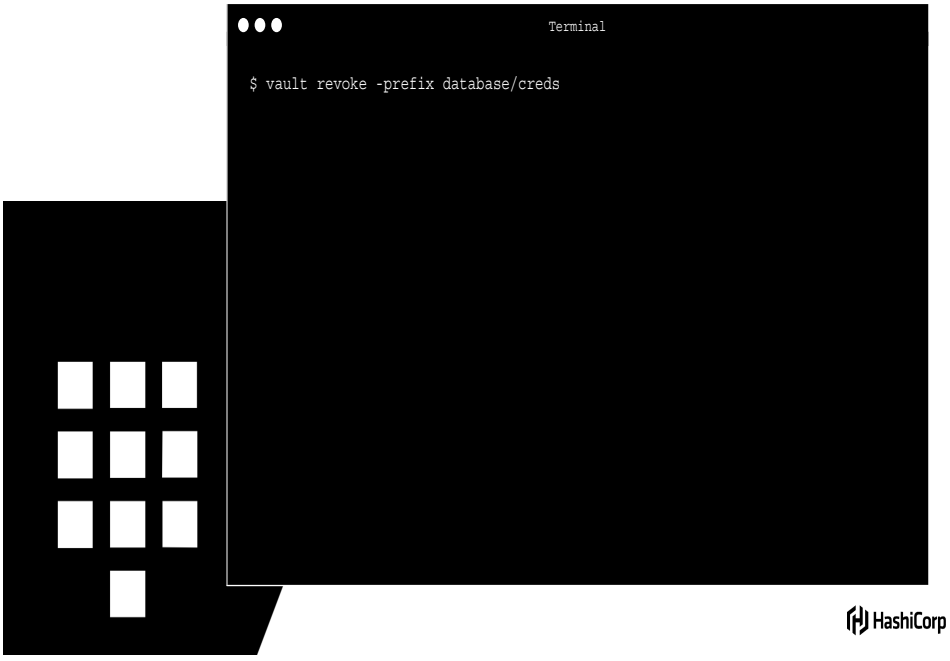
Now let's revoke this credential.



We can check that account was removed by logging in as the postgres user and listing all accounts.

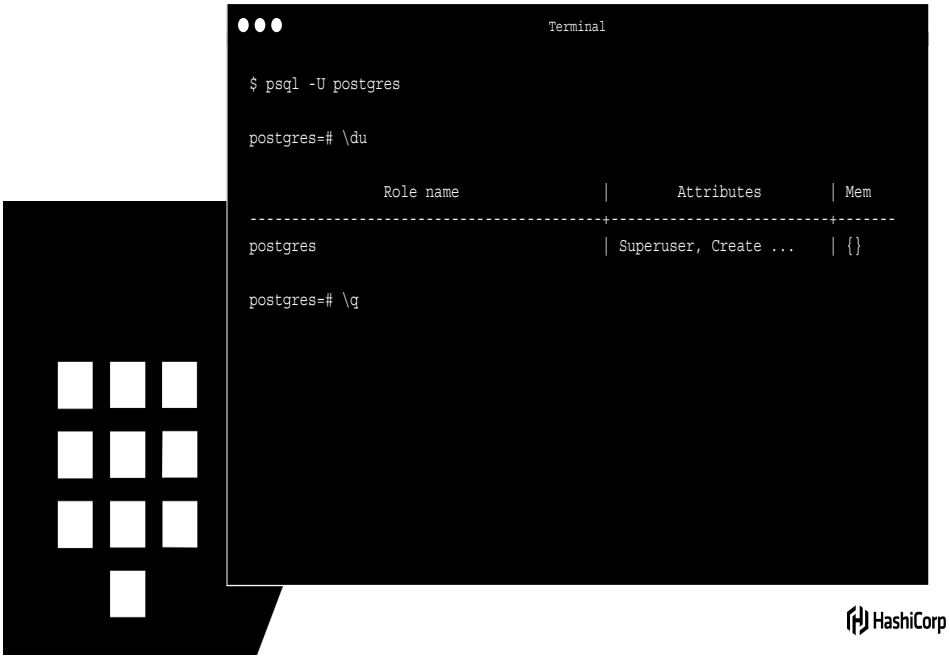
Let's read a few more credentials from the postgres secret backend. Here, we are simulating a real production scenario where multiple applications have requested database access.

Now let's imagine a scenario where we need to revoke all these secrets. Maybe we detected an anomaly in the postgres logs or the vault audit logs that indicates there may be a breach; we can revoke specific leases, but we may not know them (as in this case).



We can specify the path of the credentials we want to revoke. In this case, we want to revoke all the postgres credentials.

You should see output like this.

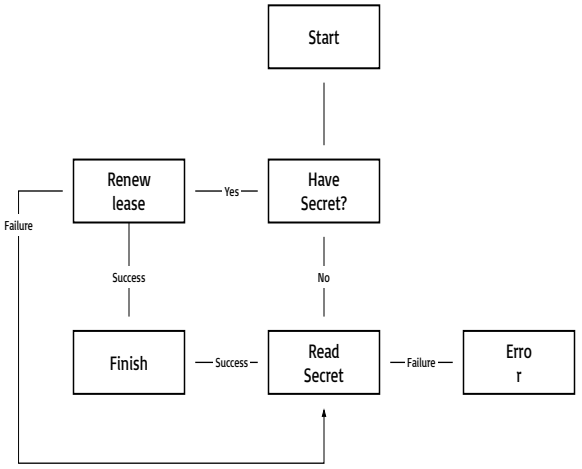


If we login to postgres and list all users, you'll notice that Vault revoked all those accounts.

Working with Leases

HashiCorp

Leasing, renewal, and revocation



If you recall from the earlier section, we discussed the lease renewal and revocation workflow. Almost everything in Vault has an associated lease, and when that lease is expired, the secret is revoked. This actually includes tokens as well. A token, if not renewed, with automatically expire.

Lease Hierarchy and Revocations

b519c6aa... (3h)
6a2cf3e7... (4h)
1d3fd4b2... (1h)
794b6f2f... (2h)

HashiCorp

With every secret and authentication token, Vault creates a lease: metadata containing information such as a time duration, renewability, and more. Vault promises that the data will be valid for the given duration, or Time To Live (TTL). Once the lease is expired, Vault can automatically revoke the data, and the consumer of the secret can no longer be certain that it is valid.

The benefit should be clear: consumers of secrets need to check in with Vault routinely to either renew the lease (if allowed) or request a replacement secret. This makes the Vault audit logs more valuable and also makes key rolling a lot easier.

All secrets in Vault are required to have a lease. Even if the data is meant to be valid for eternity, a lease is required to force the consumer to check in routinely.

Suppose an architecture exists like this. When a new token or secret is created, it's actually a child of the creator. If the parent is revoked or expires, so too do all the children, **regardless of their own leases**.

Exercise: Predicting Behavior

List the order in which the leases would expire.

b519c6aa... (3h)

6a2cf3e7... (4h)

1d3fd4b2... (1h)

794b6f2f... (2h)

HashiCorp

Exercise: Predicting Behavior

List the order in which the leases would expire.

HashiCorp

First, 1d3fd... would be revoked, but it would also revoke it's child 794b6. Then, 2 hours later, b519c would revoke and take 6a2cf with it.

Token and Lease Renewals

If a token or secret with a lease is not renewed before the lease expires, it and all children will be revoked by the Vault server.

A child is a token, secret, or authentication created by a parent. A parent is almost always a token.

Exercise: Understanding Revocations

1. Create a new token with a 30s lease (hint: use help output)
2. Auth as this token
3. After 30s, try to read a value using token

HashiCorp


Terminal
$ vault token-create -ttl=30s
token 5cf4e7b5-7f88-0769-e406-686e2a90c471
token_duration 30
$ vault auth 5cf4e7b5-7f88-0769-e406-686e2a90c471
Successfully authenticated!
$ vault read secret/training
...
$ vault read secret/training
Error!

HashiCorp

Exercise: Re-auth as root

Re-authenticate as root

HashiCorp

Here is a sample solution.

Lease Best Practices

Renew leases at half the lease duration value – e.g. 10m lease should renew every 5m.

Attempt a re-read if renewal fails (generates new credentials).

Notable Exception: Orphan Token

Root/sudo users have the ability to generate "orphan" tokens.

Orphan tokens are not children of their parent, therefore do not expire when their parent does.

Orphan tokens still expire when their own Max TTL is reached.

Notable Exception: Periodic Token

Root/sudo users have the ability to generate "periodic" tokens.

Periodic tokens have a TTL, but no max TTL.

Periodic tokens may live for an infinite amount of time, so long as they are renewed within their TTL.

This is useful for long-running services that cannot handle regenerating a token.

Notable Exception: Use Limits

In addition to TTL and Max TTL, tokens may be limited to a number of uses.

Use limit tokens expire at the end of their last use, regardless of their remaining TTLS.

Use limit tokens expire at the end of their TTLs, regardless of remaining uses.

Authentication

HashiCorp

Understanding Authentication

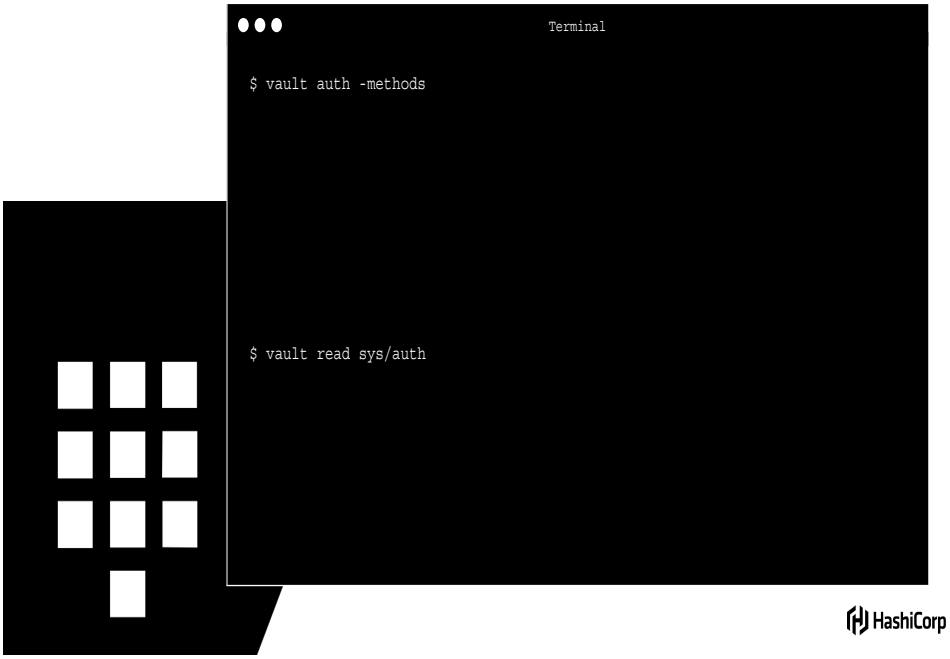
Authentication is a process in Vault by which user or machine-supplied information is verified to create a token with pre-configured policy.

Future requests are made using the token.

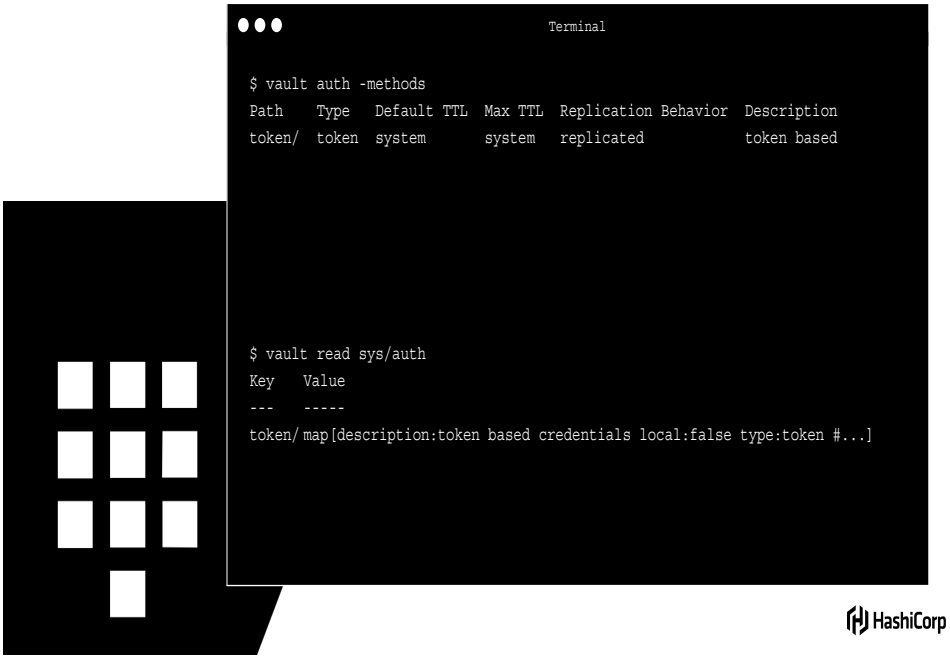
Authentication Setup

1. Activate the authentication using the auth-enable command
2. Configure the authentication (varies)
3. Map the authentication to a set of policies

HashiCorp



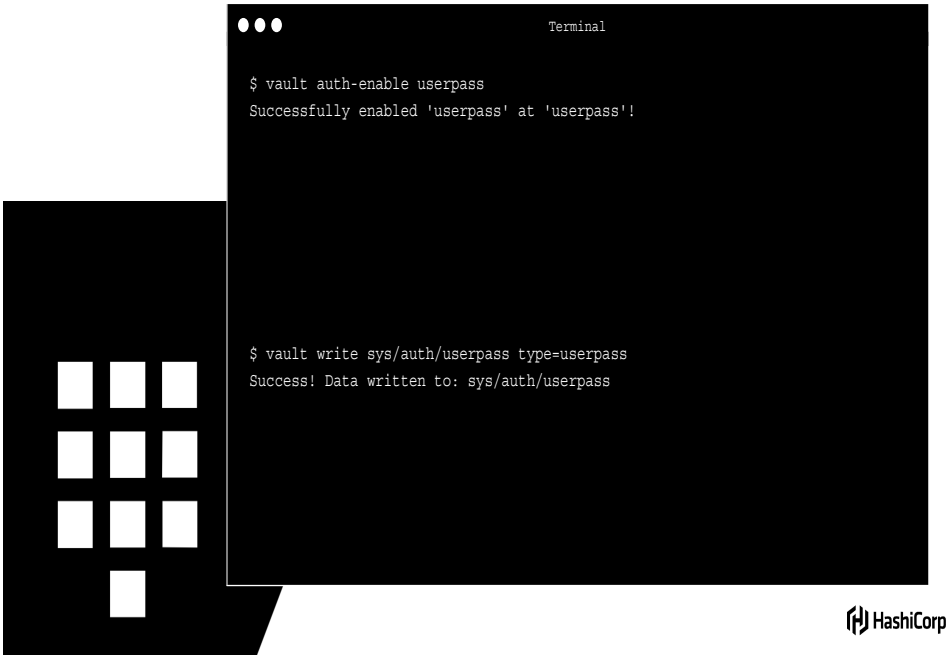
Let's list all the currently configured authentication backends in Vault. We can do this using the command `vault auth -methods`.



You should see something like this.

Token auth is always enabled.

Let's enable the username-password authentication backend. This allows users to supply a username and password to retrieve a vault token, similar to how a website login works.



You should see a message like this.

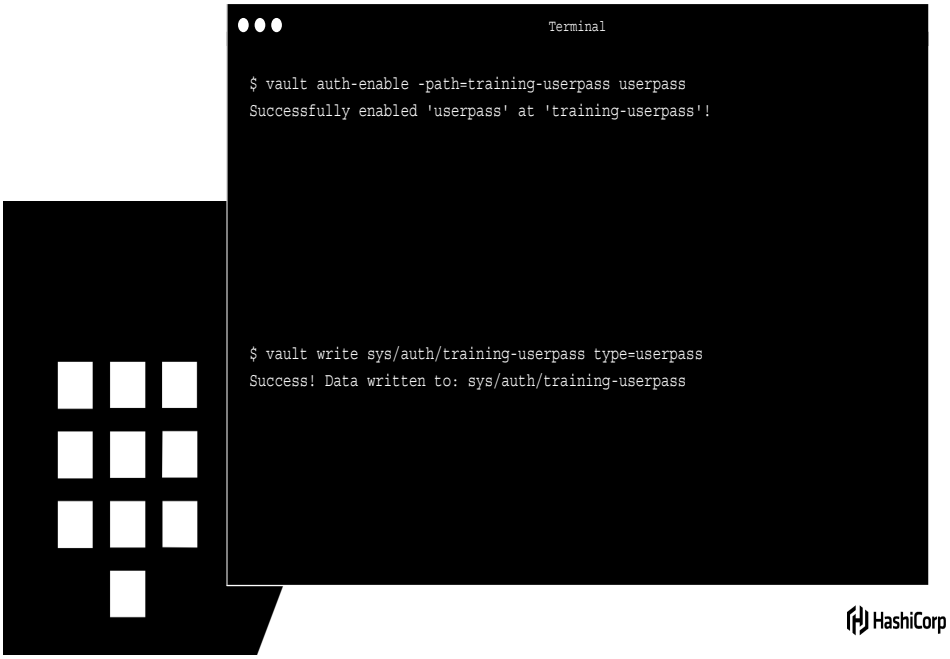
We can verify the auth method was successfully enabled by listing all the methods.

Exercise: Mount at Path

Mount the userpass backend at the path "training-userpass".

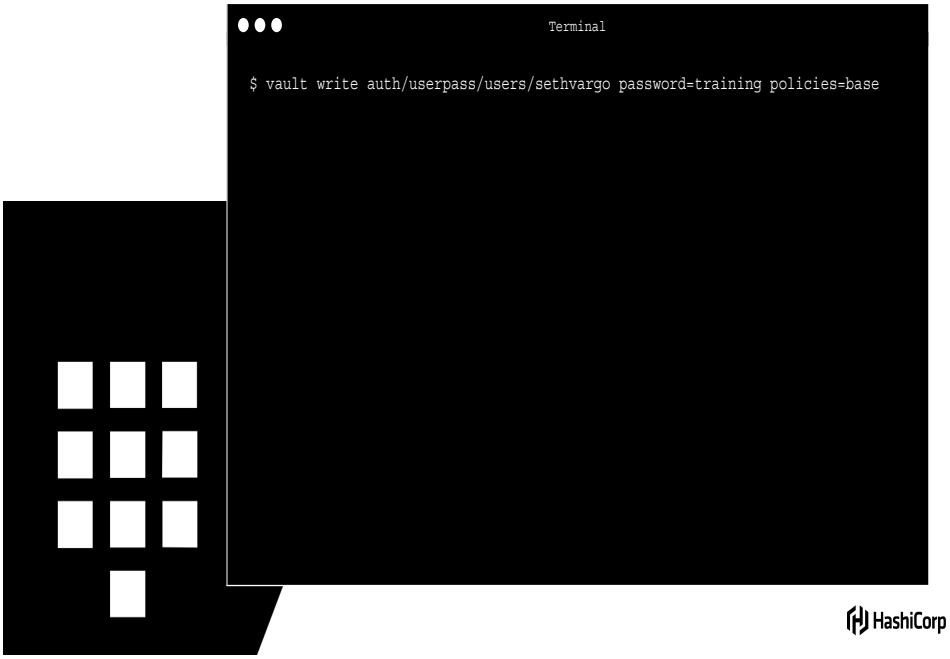
HashiCorp

Everything in Vault is path based, and we can mount the same backend at multiple paths. Data between backends is not shared between paths (they are isolated and separate instances, much like an instance of an object in object-oriented programming).



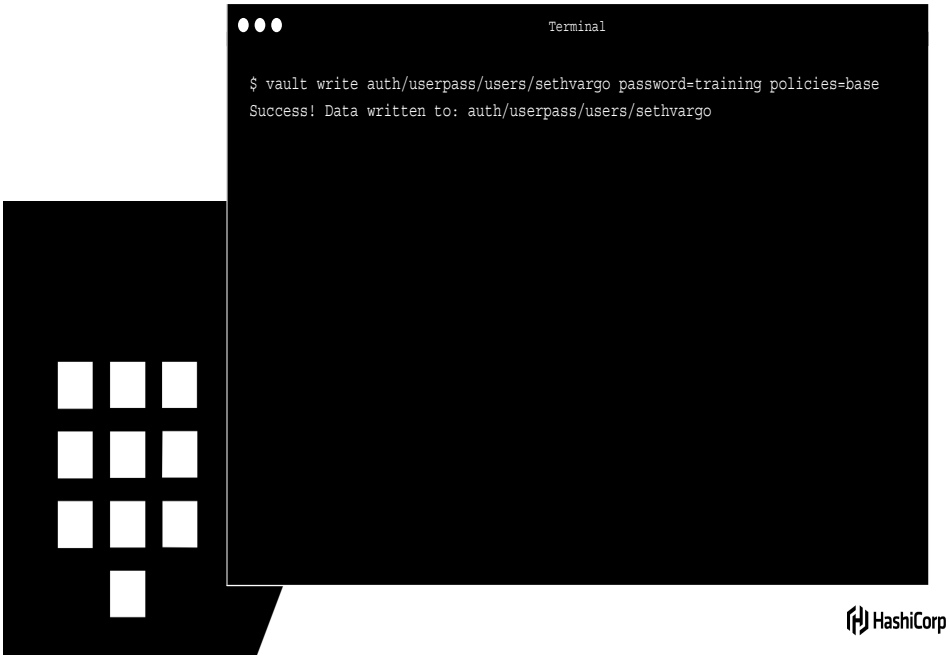
Here you can see the subtle differences between the CLI and the API interactions. Notice that the API makes it a bit clearer.

Again, these two commands are functionally the same.



Let's create our first user. Replace my information with yours. Notice that the "username" is part of the path and the two parameters are the password (in plain text) and the list of policies as comma-separated values.

Here, we are granting access to our "base" policy from earlier.



Assuming all was successful, you should see a message like this.

We can also verify the setup is correct by reading from the path. Notice that the password is not included in the response.

Exercise: Create Auth with Custom Policy

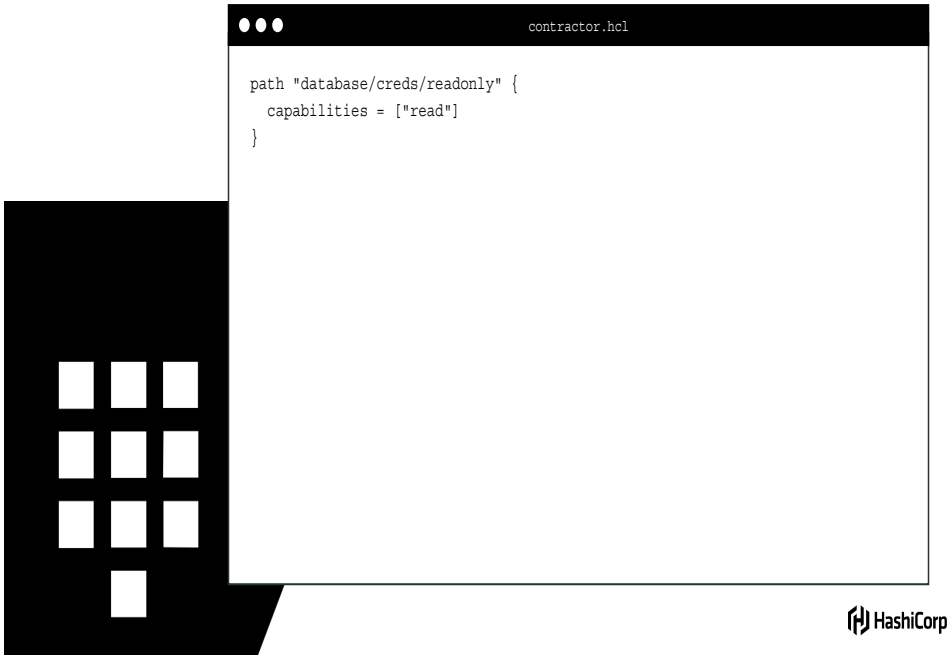
Create a new policy named "contractor" that grants only the ability to generate readonly credentials from the database backend.

Create a new userpass authentication that attaches the above policy. Use the username "sandy" and the password "training".

Authenticate as this user and generate a postgresql credential (HINT: vault auth -h)

HashiCorp

Instructor note: casually remind students that exercises build on each other, and that the Internet and online documentation are fair resources.



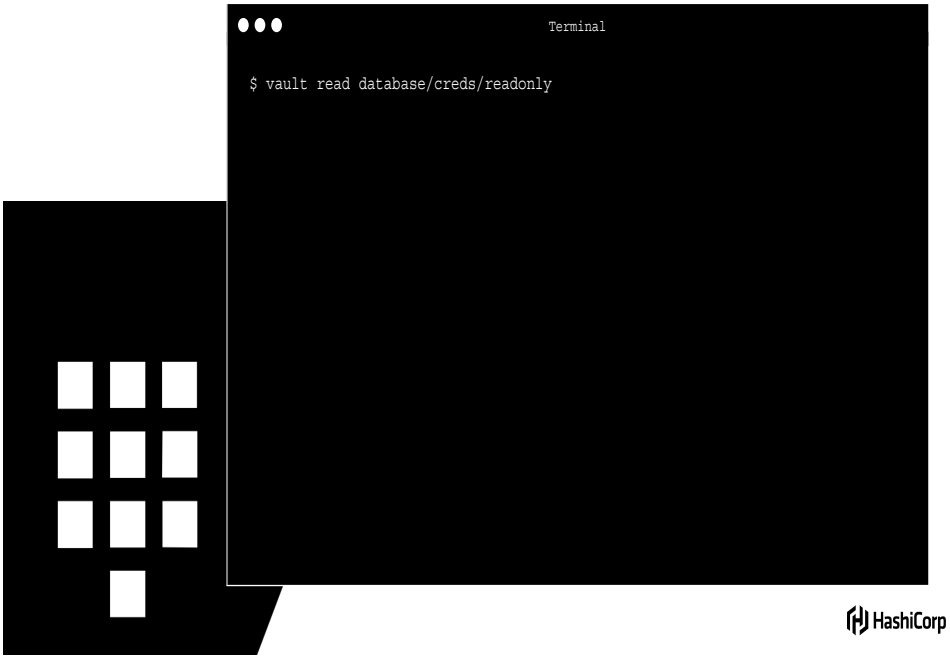
Here is a sample policy

Next we have to write that policy using the policy-write command.

Next, we need to setup the contractor user and map to the appropriate policy.

Finally, authenticate as this user.

BIG IMPORTANT NOTE - In this example, the CLI and API calls behave differently. The `vault auth` command provides some additional sugar with saves the generated token to the local keyring. The API is stateless, so the second command generates the token, but does not save it locally.



Now let's try to read some postgres credentials

You should see something like this. Notice how the username includes the authentication method and the username. This can be helpful in identifying where a secret came from.

Try to perform some other operations in vault and notice they are denied.

Exercise: Auth as yourself

Authenticate as root token.

HashiCorp

Here is a sample solution.

Auditing

HashiCorp

About Audit Backends

Audit backends keep a detailed log of all requests and responses to Vault.

Sensitive information is obfuscated by default (HMAC).

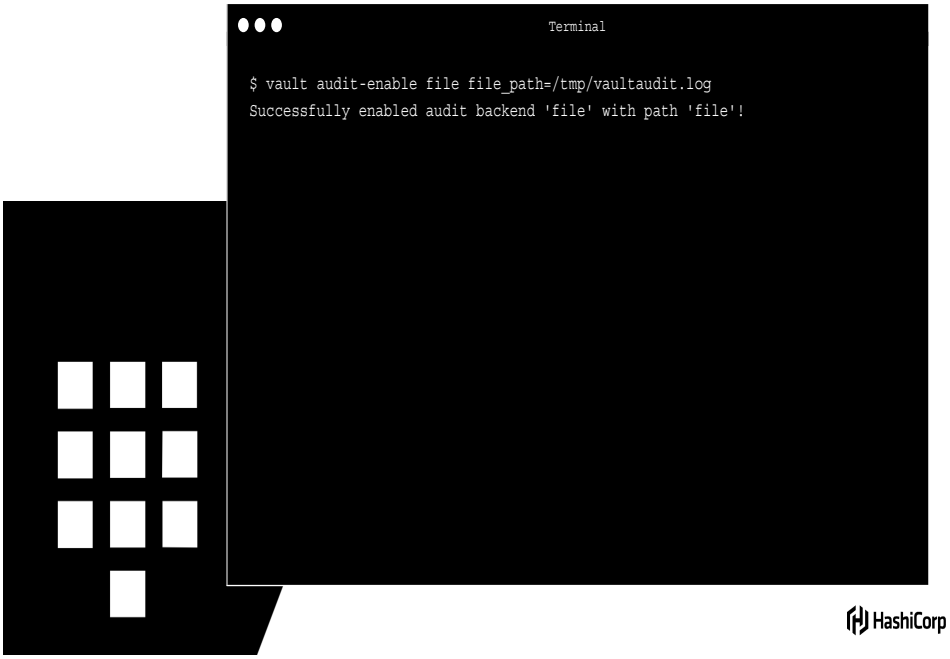
Prioritizes safety over availability.

Exercise: Enable Audit Backend

Enable the "file" audit backend to write to the path `/workstation/vault/audit.log`

HINT: there are two "paths" - the URL path and the path on disk

HashiCorp



Here is how we enable the audit backend.

If we cat the audit log, you can see there is already one entry. It's a bit meta, but the audit log logged itself.

Auditing Additional Fields

In addition to the standard fields, Vault can optionally audit user-defined headers

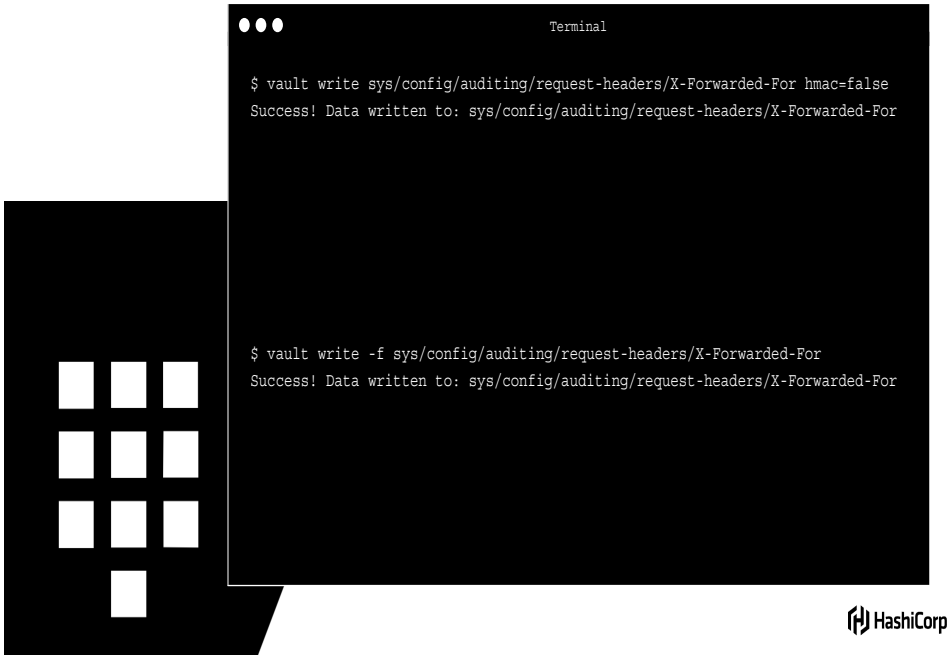
Useful for logging things like `X-Forwarded-For`

Exercise: Audit X-Forwarded-For

Configure Vault to audit the `X-Forwarded-For` header.

HINT: API docs for `sys/config`

HashiCorp



Here is a sample solution.

Now if we make a request (it doesn't have to succeed) to Vault with the X-Forwarded-For header, it'll appear in the audit logs.

High Availability

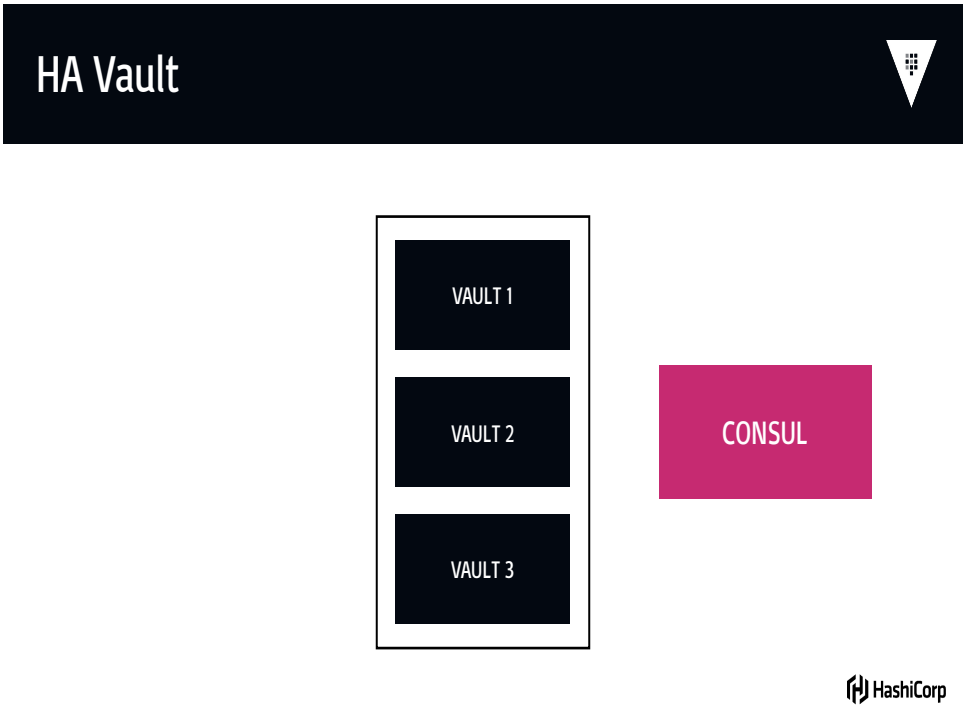
HashiCorp

Deploying Vault HA

1. Deploy one Vault with an HA storage backend configured
2. Run *vault init* to generate unseal keys and token on first Vault
3. Unseal the Vault
4. Repeat the above steps on the second Vault, except `init`

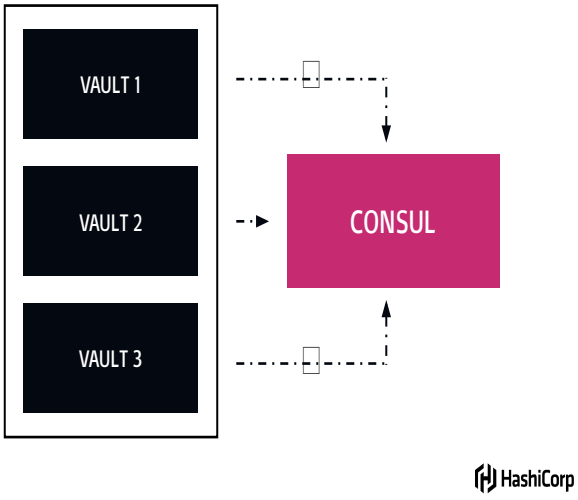
HashiCorp

Vault is primarily used in production environments to manage secrets. As a result, any downtime of the Vault service can affect downstream clients. Vault is designed to support a highly available deploy to ensure a machine or process failure is minimally disruptive.



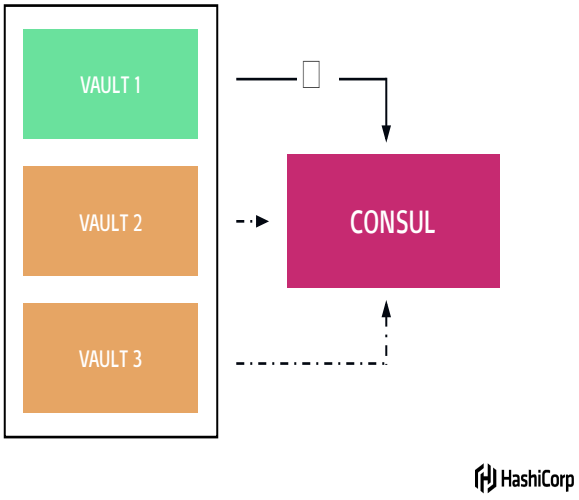
In an HA setup, all the Vault servers are running.

HA Vault

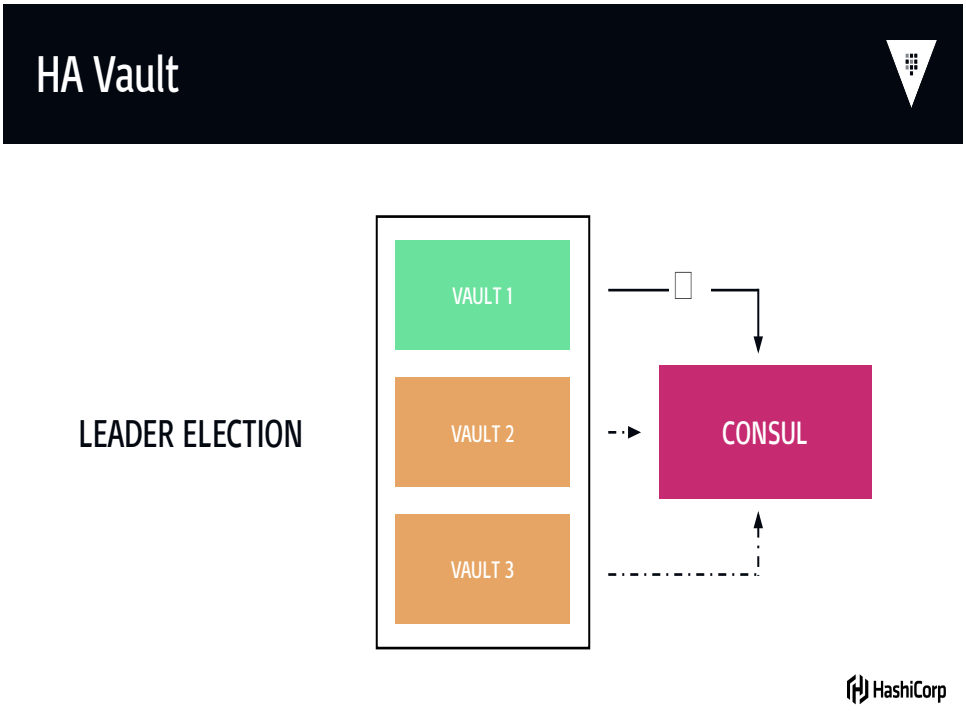


At startup, they all try to acquire a "lock" on a shared key in the storage backend.

HA Vault

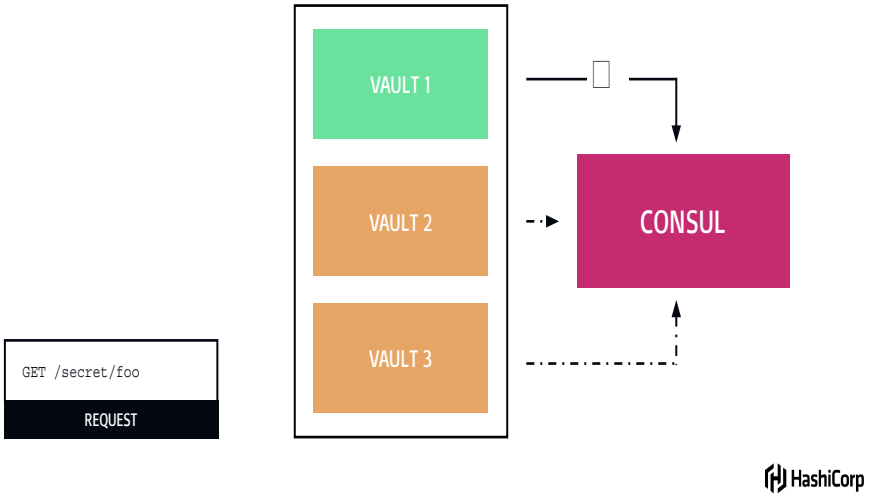


One of them succeeds, and goes into leadership mode, while the others remain alive but in standby.

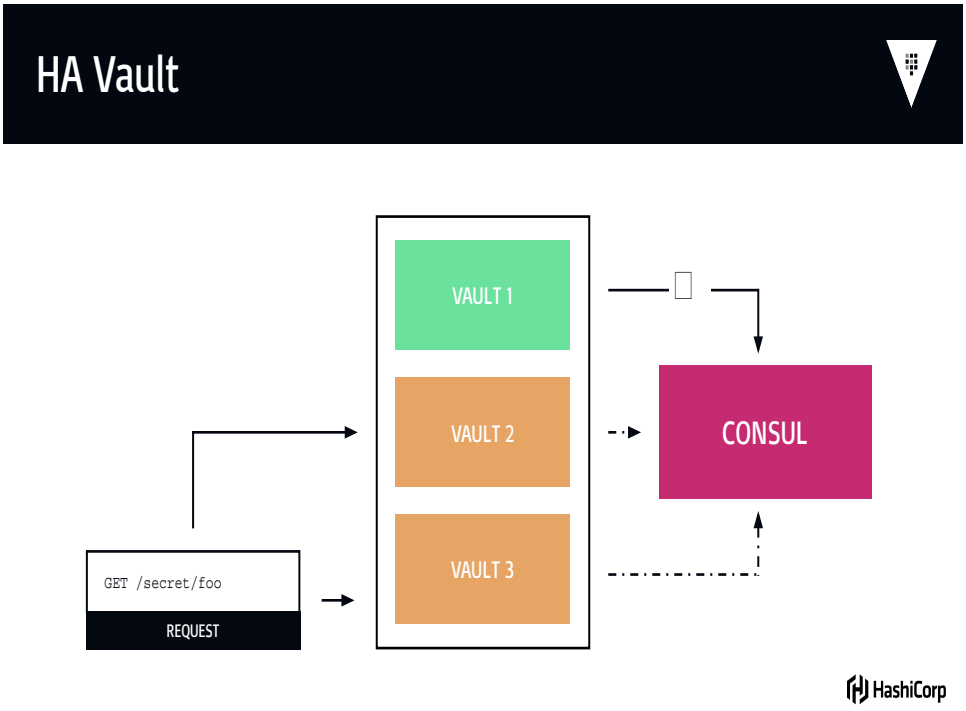


This is basic leader election in a distributed system.

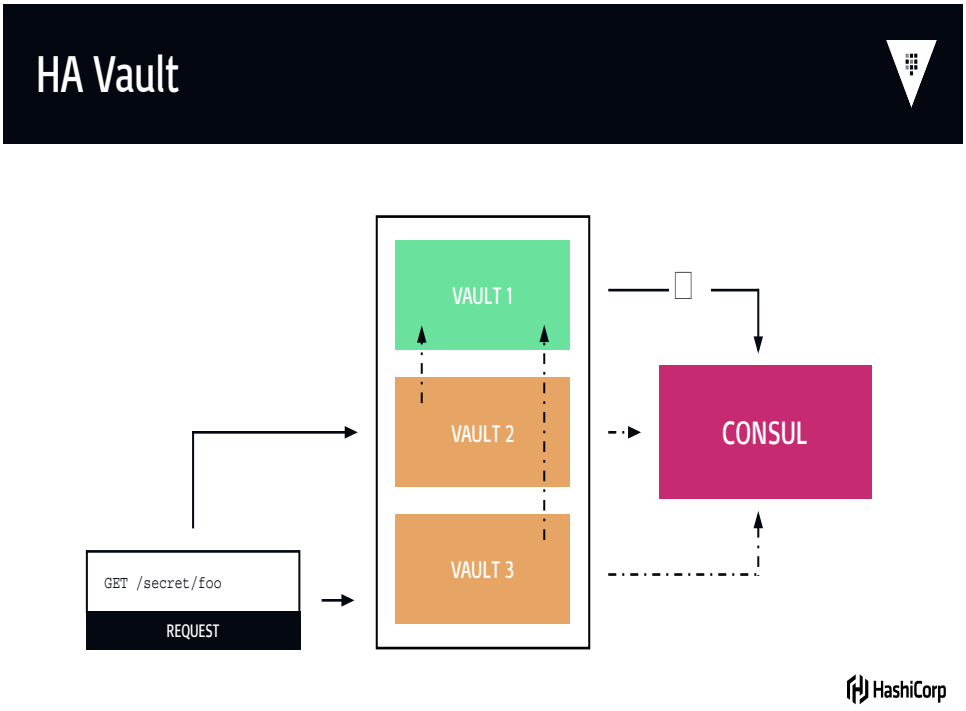
HA Vault



When a request comes in to the vault servers...



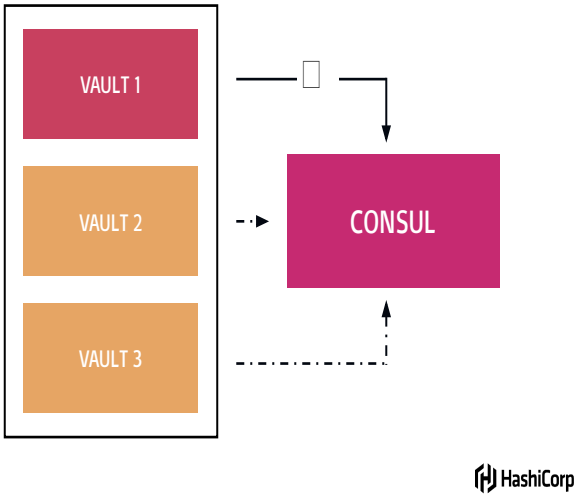
If it requests any of the standby nodes...



The standby nodes forward the request to the leader. The standbys know the location of the leader because the HA storage backend informed them.

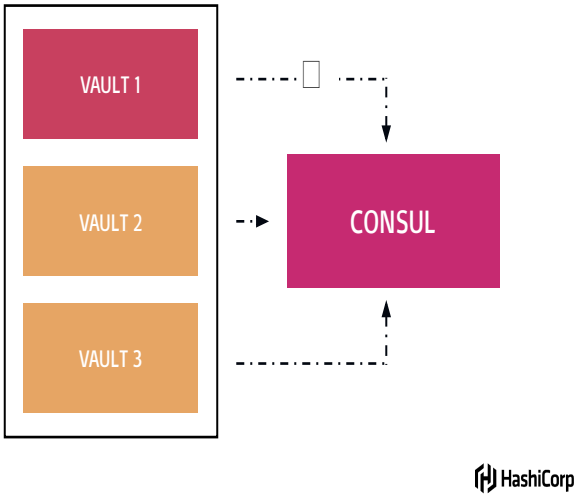
Instructor note: This is true as of Vault 0.7. Previous versions of Vault returned a client-side redirect.

HA Vault



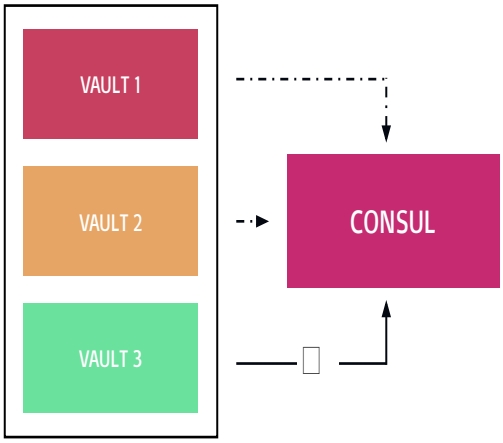
If the leader ever goes down

HA Vault

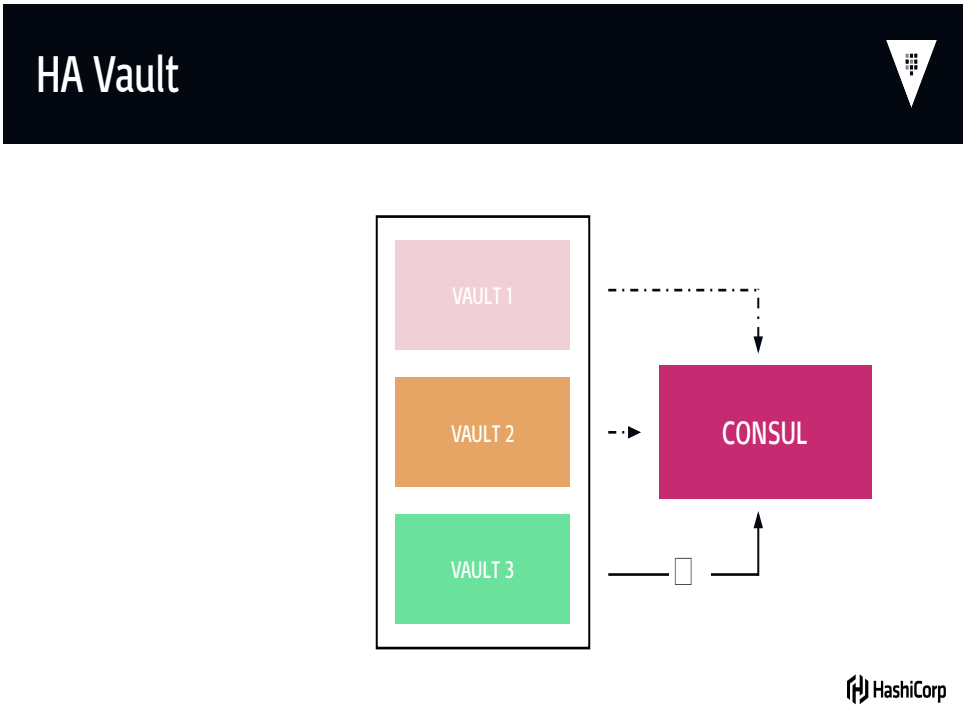


It loses it's lock either gracefully or after the minimal TTL.

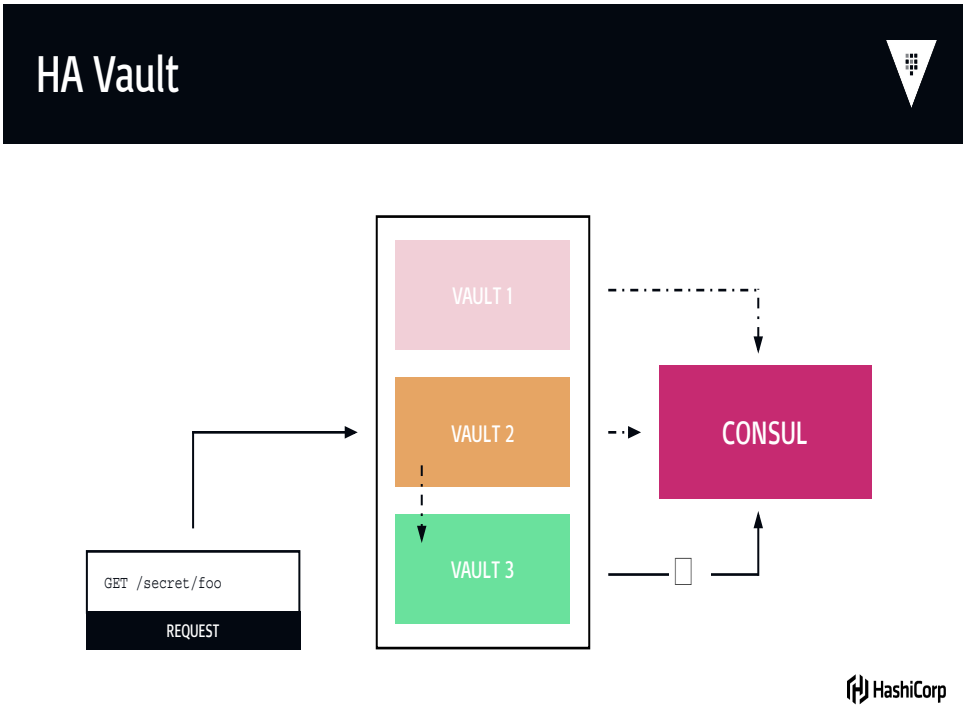
HA Vault



HashiCorp



Vault 1 is either decommissioned, but in the meantime one of the other vault's takes leadership.



The request process is the same, except with one less vault.

Exercise: Enable HA

Unseal vault on vault2:

```
export VAULT_ADDR=http://192.168.50.152:8200
vault unseal
```

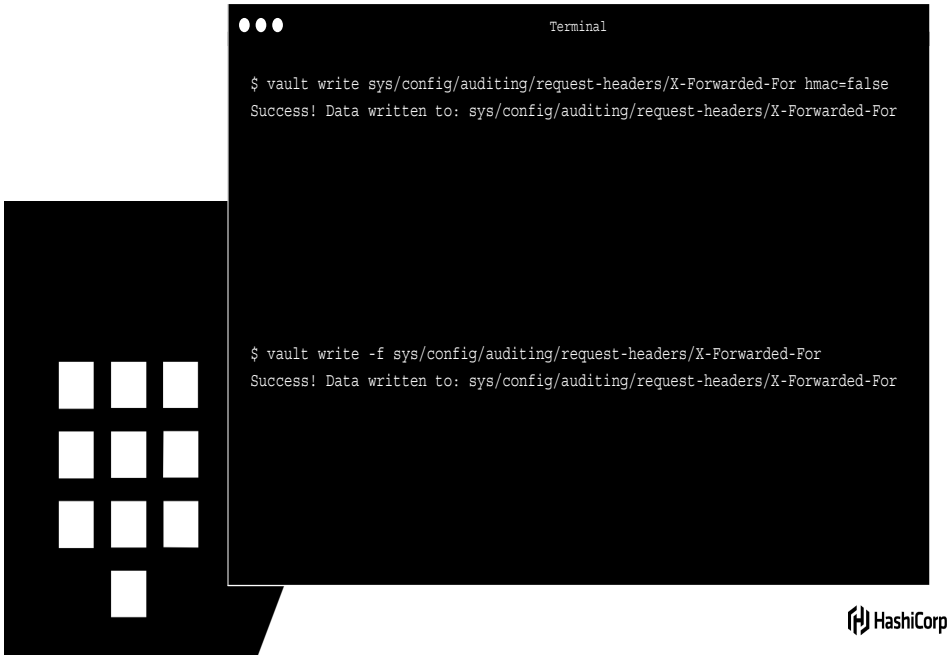
Unseal vault on vault3:

```
export VAULT_ADDR=http://192.168.50.153:8200
vault unseal
```

View Health Checks on Consul:

```
http://192.168.50.151:8500
```

HashiCorp



Here is a sample solution.

(Re)generating Root

HashiCorp

Regenerating the Root Token

In a production Vault installation, the initial root token should only be used for initial configuration.

After a subset of administrators have `sudo` access, almost all operations can be performed.

But for some system-critical operations, a root token may still be required.

Regenerating the Root Token

A quorum of unseal key holders can generate a new root token.

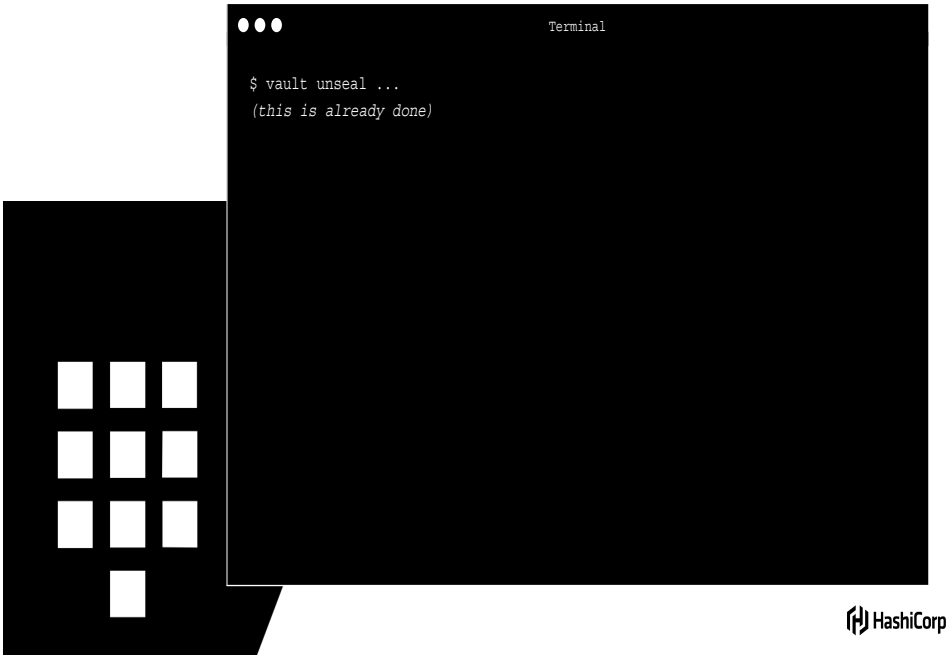
Enforces the "no one person has complete access to the system".

HashiCorp

Steps to Regenerate Root

1. Make sure the Vault is unsealed
2. Generate a one-time-password to share
3. Each key-holder runs `generate-root` with the OTP
4. Decode the root token

HashiCorp



First the vault must be unsealed. This is already done.

Next we need to generate the OTP. This OTP ensures that all keyholders are taking place in the same operation and prevents "trickery".

Next we need to generate the OTP. This OTP ensures that all keyholders are taking place in the same operation and prevents "trickery".

After you type in the last key, you'll an option like this.

Then finally decode the root token by supplying the otp and value.

Exercise: Generate New Root Token

Generate a new root token for the initial Vault server (not vault-2).

HINT: Find the unseal key in `sudo journalctl -u vault-2`

HashiCorp

Instructor note: you'll probably be asked to go back a few slides. DON'T. Reinforce that Google and the Vault documentation is the best place to get help, since they won't have slides in front of them next time they need to do this.

HTTP API

HashiCorp

About the HTTP API

All interactions with Vault happen via the HTTP API

Even the CLI uses the HTTP API – there is nothing special

Auth is passed via the X-Vault-Token header unless authing

Multiple client libraries exist (Go, Ruby, Python, Node, etc)

HashiCorp

Class poll: Why do we use a header instead of a query string parameters?
Answer : Because it won't show up in access logs.

HTTP API Status Codes

- 200/204 - Success (no data)
- 400 - Invalid request
- 403 - Forbidden
- 404 - Invalid path
- 429 - Rate limit exceeded
- 500 - Internal server error
- 503 - Vault is sealed or in maintenance

HashiCorp

Here is an example HTTP API request to retrieve the secret named "training" in the generic secret backend. Notice that the API returns more information than the CLI, including any warnings. The actual data for the secret is stored as a map in the "data" key.

To perform a list operation, use the LIST HTTP verb instead of GET.

To perform a write operation, use the POST or PUT HTTP verb instead of GET.

Exercise: Use HTTP API

Retrieve a new set of readonly credentials from the postgres backend using the HTTP API.

HashiCorp

Here is the answer and sample response.

Exercise: Use HTTP API

Renew the lease you just created, using the HTTP API (HINT: `sys`).

HashiCorp

Here is the answer and sample response.

Exercise: Authenticate Using the HTTP API

Authenticate as a contractor with userpass using the HTTP API.

Recall that the credentials are:

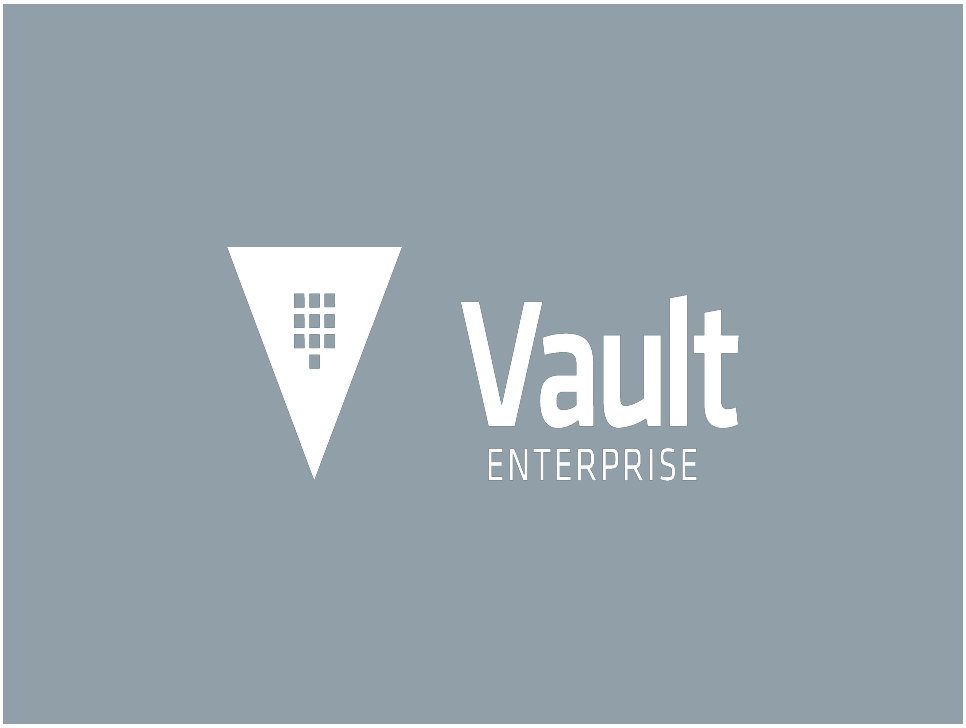
Username: `sandy`
Password: `training`

HINT: You may need to look at the API documentation

Here is the answer.

Here is a sample response. The important part is the "client_token", which is the new token that we would auth with.

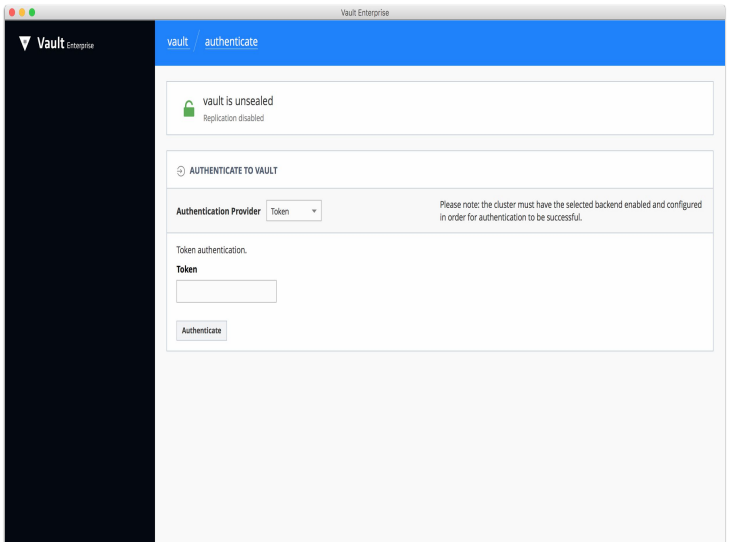
The difference here vs the CLI is that the CLI will take that token and store it for us. In the case of curl, because HTTP is stateless, we would need to copy-paste this token for future requests.



Depending on time/comfort, you can either show screenshots or work with the UI directly.

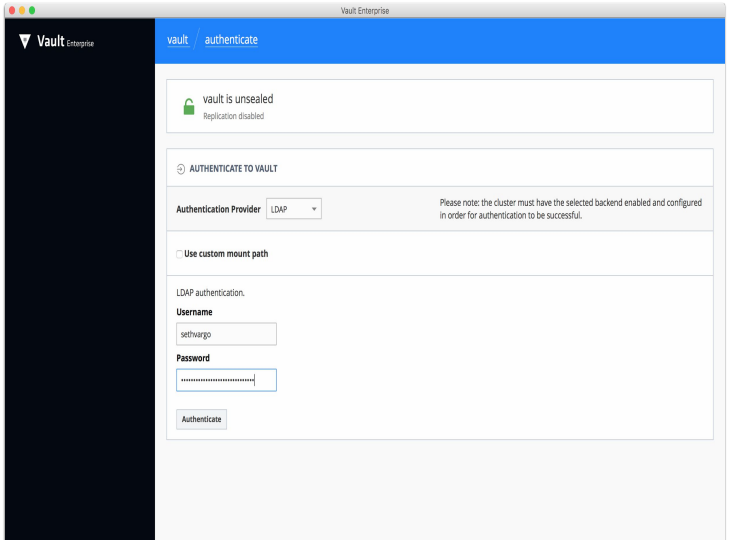
Instructor note: the load balancer address and initial root token are shown in the terraform output when you created the lab. You can use `terraform output` to see it again.

Vault Enterprise



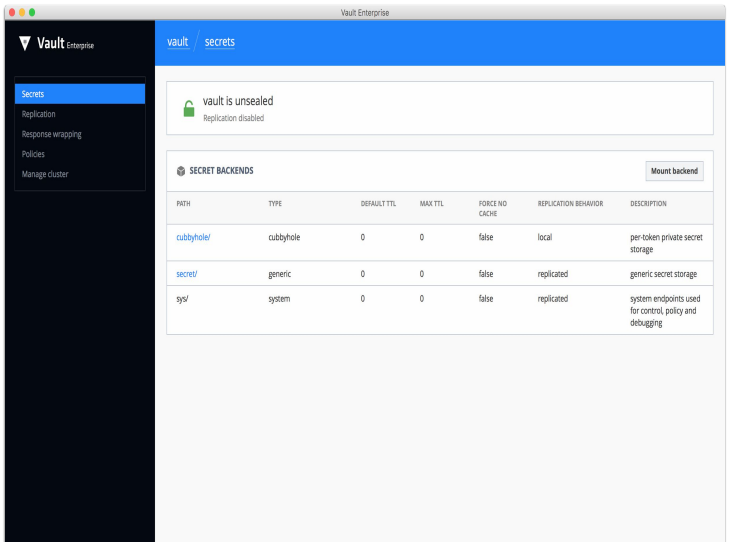
When first visited, the browser will show you a login page. You can login with a token...

Vault Enterprise



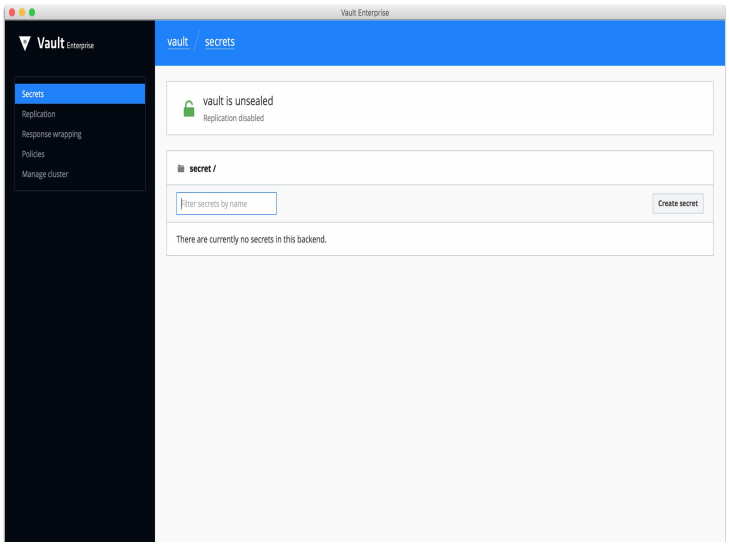
Or a username-password connected to LDAP.

Vault Enterprise



Once authenticated, you'll see a page like this. All the mount points are displayed here.

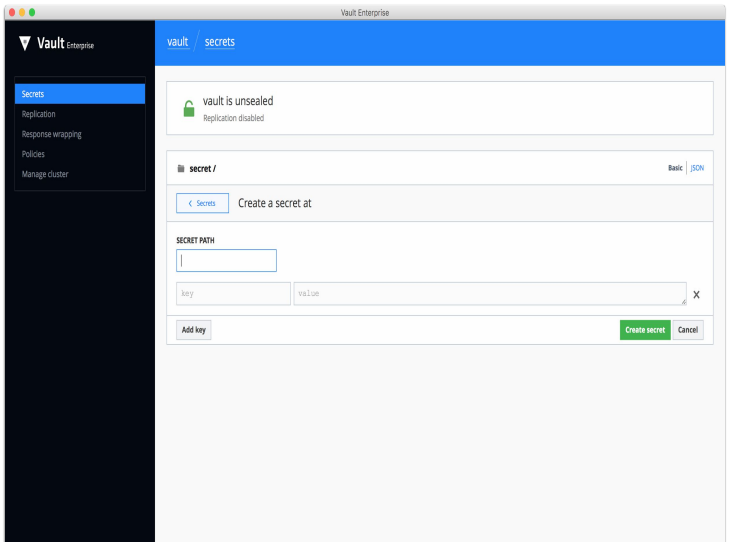
Vault Enterprise



HashiCorp

You can list and filter secrets

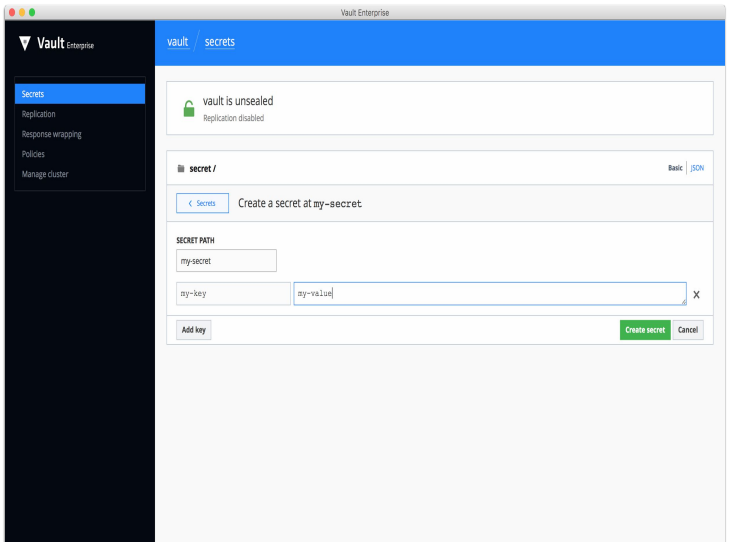
Vault Enterprise



HashiCorp

create new secrets

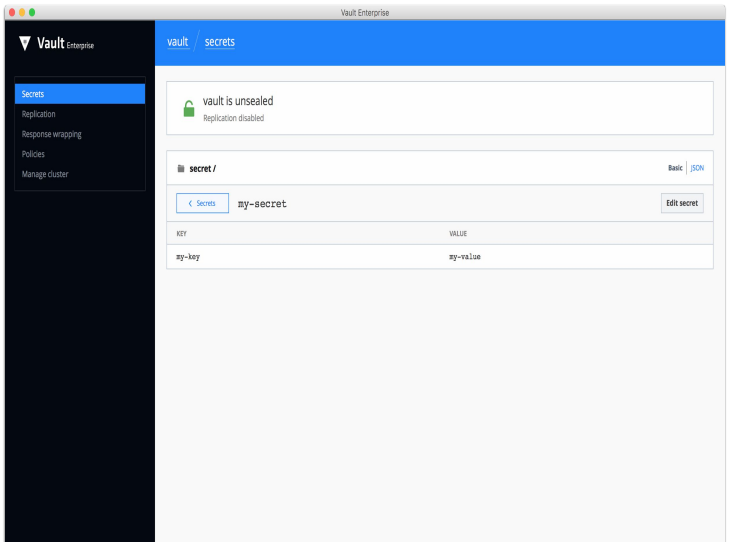
Vault Enterprise



HashiCorp

(still create)

Vault Enterprise



Browse all secrets and delete secrets, and much more

Further Reading

HashiCorp

Further Reading

CLI

vaultproject.io/docs/commands

HTTP API

vaultproject.io/docs/http

Internals

vaultproject.io/docs/internals