Policies

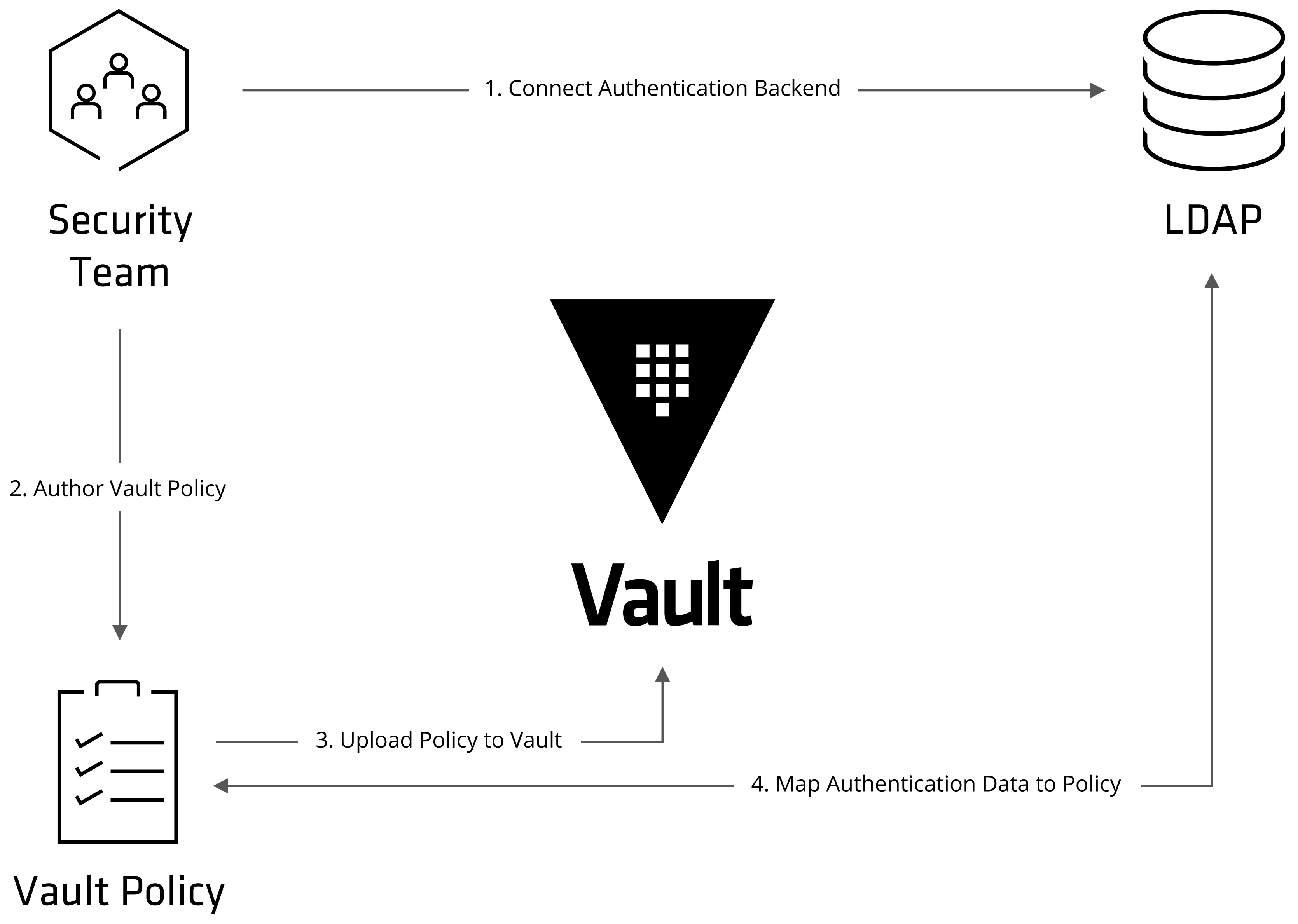
Everything in Vault is path based, and policies are no exception. Policies provide a declarative way to grant or forbid access to certain paths and operations in Vault. This section discusses policy workflows and syntaxes.

Policies are **deny by default**, so an empty policy grants no permission in the system.

Policy-Authorization Workflow

Before a human or machine can gain access, an administrator must configure Vault with an auth method (/docs/concepts/auth.html). Authentication is the process by which human or machine-supplied information is verified against an internal or external system.

Consider the following diagram, which illustrates the steps a security team would take to configure Vault to authenticate using a corporate LDAP or ActiveDirectory installation. Even though this example uses LDAP, the concept applies to all auth methods.


Security Team
1. Connect Authentication Backend
LDAP
2. Author Vault Policy
Vault
3. Upload Policy to Vault
4. Map Authentication Data to Policy
Vault Policy

(/assets/images/vault-policy-workflow-57ecfb04.svg)

1. The security team configures Vault to connect to an auth method. This configuration varies by auth method. In the case of LDAP, Vault needs to know the address of the LDAP server and whether to connect using TLS. It is important to note that Vault does not store a copy of the LDAP database - Vault will delegate the authentication to the auth method.

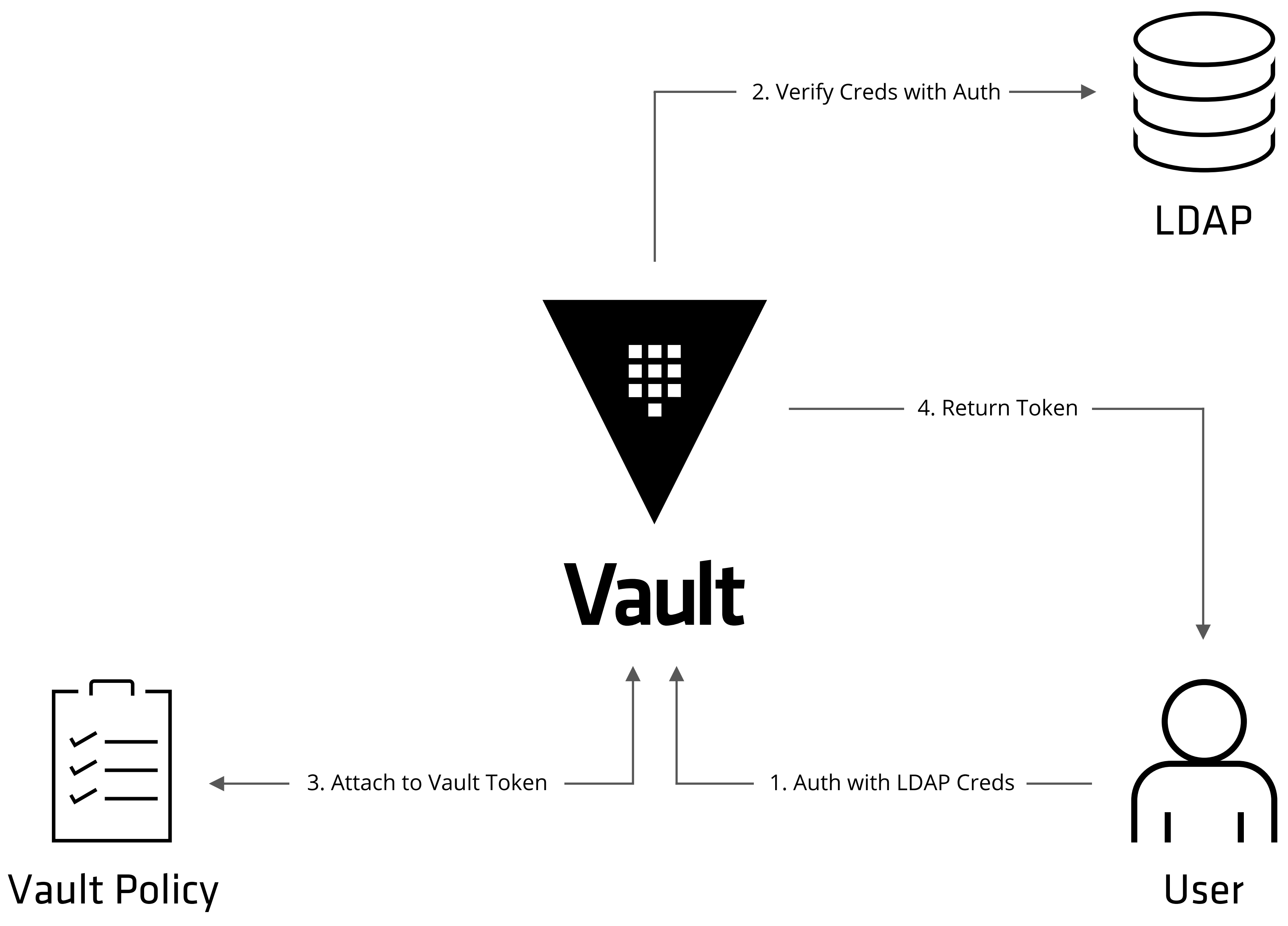
2. The security team authors a policy (or uses an existing policy) which grants access to paths in Vault. Policies are written in HCL in your editor of preference and saved to disk.
3. The policy's contents are uploaded and store in Vault and referenced by name. You can think of the policy's name as a pointer or symlink to its set of rules.
4. Most importantly, the security team maps data in the auth method to a policy. For example, the security team might create mappings like:

   > Members of the OU group "dev" map to the Vault policy named "readonly-dev".

   or

   > Members of the OU group "ops" map to the Vault policies "admin" and "auditor".

Now Vault has an internal mapping between a backend authentication system and internal policy. When a user authenticates to Vault, the actual authentication is delegated to the auth method. As a user, the flow looks like:



(/assets/images/vault-auth-workflow-ae1de5c3.svg)

1. A user attempts to authenticate to Vault using their LDAP credentials, providing Vault with their LDAP username and password.
2. Vault establishes a connection to LDAP and asks the LDAP server to verify the given credentials. Assuming this is successful, the LDAP server returns the information about the user, including the OU groups.

3. Vault maps the result from the LDAP server to policies inside Vault using the mapping configured by the security team in the previous section. Vault then generates a token and attaches the matching policies.
4. Vault returns the token to the user. This token has the correct policies assigned, as dictated by the mapping configuration that was setup by the security team in advance.

The user then uses this Vault token for future operations. If the user performs the authentication steps again, they will get a *new* token. The token will have the same permissions, but the actual token will be different. Authenticating a second time does not invalidate the original token.

Policy Syntax

Policies are written in HCL (https://github.com/hashicorp/hcl) or JSON and describe which paths in Vault a user or machine is allowed to access.

Here is a very simple policy which grants read capabilities to the path "secret/foo":

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

When this policy is assigned to a token, the token can read from `"secret/foo"`. However, the token could not update or delete `"secret/foo"`, since the capabilities do not allow it. Because policies are **deny by default**, the token would have no other access in Vault.

Here is a more detailed policy, and it is documented inline:

```
# This section grants all access on "secret/*". Further restrictions can be
# applied to this broad policy, as shown below.
path "secret/*" {
  capabilities = ["create", "read", "update", "delete", "list"]
}

# Even though we allowed secret/*, this line explicitly denies
# secret/super-secret. This takes precedence.
path "secret/super-secret" {
  capabilities = ["deny"]
}

# Policies can also specify allowed, disallowed, and required parameters. Here
# the key "secret/restricted" can only contain "foo" (any value) and "bar" (one
# of "zip" or "zap").
path "secret/restricted" {
  capabilities = ["create"]
  allowed_parameters = {
    "foo" = []
    "bar" = ["zip", "zap"]
  }
}
```

Policies use path-based matching to test the set of capabilities against a request. A policy `path` may specify an exact path to match, or it could specify a glob pattern which instructs Vault to use a prefix match:

```
# Permit reading only "secret/foo". An attached token cannot read "secret/food"
# or "secret/foo/bar".
path "secret/foo" {
  capabilities = ["read"]
}

# Permit reading everything under "secret/bar". An attached token could read
# "secret/bar/zip", "secret/bar/zip/zap", but not "secret/bars/zip".
path "secret/bar/*" {
  capabilities = ["read"]
}

# Permit reading everything prefixed with "zip-". An attached token could read
# "secret/zip-zap" or "secret/zip-zap/zong", but not "secret/zip/zap
path "secret/zip-*" {
  capabilities = ["read"]
}
```

Vault's architecture is similar to a filesystem. Every action in Vault has a corresponding path and capability - even Vault's internal core configuration endpoints live under the "sys/" path. Policies define access to these paths and capabilities, which controls a token's access to credentials in Vault.

Policy paths are matched using the **most specific path match**. This may be an exact match or the longest-prefix match of a glob. This means if you define a policy for `"secret/foo*"`, the policy would also match `"secret/foobar"`.

The glob character is only supported as the **last character of the path**, and **is not a regular expression**!

Capabilities

Each path must define one or more capabilities which provide fine-grained control over permitted (or denied) operations. As shown in the examples above, capabilities are always specified as a list of strings, even if there is only one capability. The list of capabilities are:

In the list below, the associated HTTP verbs are shown in parenthesis next to the capability. When authoring policy, it is usually helpful to look at the HTTP API documentation for the paths and HTTP verbs and map them back onto capabilities. While the mapping is not strictly 1:1, they are often very similarly matched.

- `create` (`POST/PUT`) - Allows creating data at the given path. Very few parts of Vault distinguish between `create` and `update`, so most operations require both `create` and `update` capabilities. Parts of Vault that provide such a distinction are noted in documentation.
- `read` (`GET`) - Allows reading the data at the given path.
- `update` (`POST/PUT`) - Allows change the data at the given path. In most parts of Vault, this implicitly includes the ability to create the initial value at the path.
- `delete` (`DELETE`) - Allows deleting the data at the given path.
- `list` (`LIST`) - Allows listing values at the given path. Note that the keys returned by a `list` operation are *not* filtered by policies. Do not encode sensitive information in key names. Not all backends support listing.

In addition to the standard set, there are some capabilities that do not map to HTTP verbs.

- `sudo` - Allows access to paths that are *root-protected*. Tokens are not permitted to interact with these paths unless they are have the `sudo` capability (in addition to the other necessary capabilities for performing an operation against that path, such as `read` or `delete`).

  For example, modifying the audit log backends requires a token with `sudo` privileges.

- `deny` - Disallows access. This always takes precedence regardless of any other defined capabilities, including `sudo`.

Note that capabilities usually map to the HTTP verb, not the underlying action taken. This can be a common source of confusion. Generating database credentials *creates* database credentials, but the HTTP request is a GET which corresponds to a `read` capability. Thus, to grant access to generate database credentials, the policy would grant `read` access on the appropriate path.

Fine-Grained Control

In addition to the standard set of capabilities, Vault offers finer-grained control over permissions at a given path. The capabilities associated with a path take precedence over permissions on parameters.

Parameter Constraints

In Vault, data is represented as `key=value` pairs. Vault policies can optionally further restrict paths based on the keys and data at those keys when evaluating the permissions for a path. The optional finer-grained control options are:

- `required_parameters` - A list of parameters that must be specified.

```
# This requires the user to create "secret/foo" with a parameter named
# "bar" and "baz".
path "secret/foo" {
  capabilities = ["create"]
  required_parameters = ["bar", "baz"]
}
```

- `allowed_parameters` - Whitelists a list of keys and values that are permitted on the given path.
  - Setting a parameter with a value of the empty list allows the parameter to contain any value.

```
# This allows the user to create "secret/foo" with a parameter named
# "bar". It cannot contain any other parameters, but "bar" can contain
# any value.
path "secret/foo" {
  capabilities = ["create"]
  allowed_parameters = {
    "bar" = []
  }
}
```

  - Setting a parameter with a value of a populated list allows the parameter to contain only those values.

```
# This allows the user to create "secret/foo" with a parameter named
# "bar". It cannot contain any other parameters, and "bar" can only
# contain the values "zip" or "zap".
path "secret/foo" {
  capabilities = ["create"]
  allowed_parameters = {
    "bar" = ["zip", "zap"]
  }
}
```

  - If any keys are specified, all non-specified parameters will be denied unless the parameter `"*"` is set to an empty array, which will allow all other parameters to be modified. Parameters with specific values will still be restricted to those values.

```
# This allows the user to create "secret/foo" with a parameter named
# "bar". The parameter "bar" can only contain the values "zip" or "zap",
# but any other parameters may be created with any value.
path "secret/foo" {
  capabilities = ["create"]
  allowed_parameters = {
    "bar" = ["zip", "zap"]
    "*"   = []
  }
}
```

- `denied_parameters` - Blacklists a list of parameter and values. Any values specified here take precedence over `allowed_parameters`.
  - Setting a parameter with a value of the empty list denies any changes to that parameter.

```
# This allows the user to create "secret/foo" with any parameters not
# named "bar".
path "secret/foo" {
  capabilities = ["create"]
  denied_parameters = {
    "bar" = []
  }
}
```

  - Setting a parameter with a value of a populated list denies any parameter containing those values.

```
# This allows the user to create "secret/foo" with a parameter named
# "bar". It can contain any other parameters, but "bar" cannot contain
# the values "zip" or "zap".
path "secret/foo" {
  capabilities = ["create"]
  denied_parameters = {
    "bar" = ["zip", "zap"]
  }
}
```

  - Setting to `"*"` will deny any parameter.

```
# This allows the user to create "secret/foo", but it cannot have any
# parameters.
path "secret/foo" {
  capabilities = ["create"]
  denied_parameters = {
    "*" = []
  }
}
```

  - If any parameters are specified, all non-specified parameters are allowed, unless `allowed_parameters` is also set, in which case normal rules apply.

Parameter values also support prefix/suffix globbing. Globbing is enabled by prepending or appending or prepending a splat (`*`) to the value:

```
# Only allow a parameter named "bar" with a value starting with "foo-*".
path "secret/foo" {
  capabilities = ["create"]
  allowed_parameters = {
    "bar" = ["foo-*"]
  }
}
```

Note: the only value that can be used with the `*` parameter is `[]`.

Required Response Wrapping TTLs

These parameters can be used to set minimums/maximums on TTLs set by clients when requesting that a response be wrapped (/docs/concepts/response-wrapping.html), with a granularity of a second. These can either be specified as a number of seconds or a string with a `s`, `m`, or `h` suffix indicating seconds, minutes, and hours respectively.

In practice, setting a minimum TTL of one second effectively makes response wrapping mandatory for a particular path.

- `min_wrapping_ttl` - The minimum allowed TTL that clients can specify for a wrapped response. In practice, setting a minimum TTL of one second effectively makes response wrapping mandatory for a particular path. It can also be used to ensure that the TTL is not too low, leading to end targets being unable to unwrap before the token expires.

- `max_wrapping_ttl` - The maximum allowed TTL that clients can specify for a wrapped response.

If both are specified, the minimum value must be less than the maximum. In addition, if paths are merged from different stanzas, the lowest value specified for each is the value that will result, in line with the idea of keeping token lifetimes as short as possible.

Builtin Policies

Vault has two built-in policies: `default` and `root`. This section describes the two builtin policies.

Default Policy

The `default` policy is a builtin Vault policy that cannot be removed. By default, it is attached to all tokens, but may be explicitly excluded at token creation time by supporting authentication methods.

The policy contains basic functionality such as the ability for the token to look up data about itself and to use its cubbyhole data. However, Vault is not proscriptive about its contents. It can be modified to suit your needs; Vault will never overwrite your modifications. If you want to stay up-to-date with the latest upstream version of the `default` policy, simply read the contents of the policy from an up-to-date `dev` server, and write those contents into your Vault's `default` policy.

To view all permissions granted by the default policy on your Vault installation, run:

```
$ vault read sys/policy/default
```

To disable attachment of the default policy:

```
$ vault token create -no-default-policy
```

or via the API:

```
$ curl \
  --request POST \
  --header "X-Vault-Token: ..." \
  --data '{"no_default_policy": "true"}' \
  https://vault.hashicorp.rocks/v1/auth/token/create
```

Root Policy

The `root` policy is a builtin Vault policy that can not be modified or removed. Any user associated with this policy becomes a root user. A root user can do *anything* within Vault. As such, it is **highly recommended** that you revoke any root tokens before running Vault in production.

When a Vault server is first initialized, there always exists one root user. This user is used to do the initial configuration and setup of Vault. After configured, the initial root token should be revoked and more strictly controlled users and authentication should be used.

To revoke a root token, run:

```
$ vault token revoke "<token>"
```

or via the API:

```
$ curl \
  --request POST \
  --header "X-Vault-Token: ..." \
  --data '{"token": "<token>"}' \
  https://vault.hashicorp.rocks/v1/auth/token/revoke
```

For more information, please read:

- Production Hardening (/guides/operations/production.html)
- Generating a Root Token (/guides/operations/generate-root.html)

Managing Policies

Policies are authored (written) in your editor of choice. They can be authored in HCL or JSON, and the syntax is described in detail above. Once saved, policies must be uploaded to Vault before they can be used.

Listing Policies

To list all registered policies in Vault:

```
$ vault read sys/policy
```

or via the API:

```
$ curl \
  --header "X-Vault-Token: ..." \
  https://vault.hashicorp.rocks/v1/sys/policy
```

You may also see the CLI command `vault policies`. This is a convenience wrapper around reading the sys endpoint directly. It provides the same functionality but formats the output in a special manner.

Creating Policies

Policies may be created (uploaded) via the CLI or via the API. To create a new policy in Vault:

```
$ vault write sys/policy/my-policy policy=@my-policy.hcl
```

The `@` tells Vault to read from a file on disk. In the example above, Vault -will read the contents of `my-policy.hcl` in the current working directory into -the value for that parameter.

or via the API:

```
$ curl \
  --request POST \
  --header "X-Vault-Token: ..." \
  --data '{"policy":"path \"...\" {...} "}' \
  https://vault.hashicorp.rocks/v1/sys/policy/my-policy
```

In both examples, the name of the policy is "my-policy". You can think of this name as a pointer or symlink to the policy ACLs. Tokens are attached policies by name, which are then mapped to the set of rules corresponding to that name.

Updating Policies

Existing policies may be updated to change permissions via the CLI or via the API. To update an existing policy in Vault, follow the same steps as creating a policy, but use an existing policy name:

```
$ vault write sys/policy/my-existing-policy policy=@updated-policy.json
```

or via the API:

```
$ curl \
  --request POST \
  --header "X-Vault-Token: ..." \
  --data '{"policy":"path \"...\" {...} "}' \
  https://vault.hashicorp.rocks/v1/sys/policy/my-existing-policy
```

Deleting Policies

Existing policies may be deleted via the CLI or API. To delete a policy:

```
$ vault delete sys/policy/my-policy
```

or via the API:

```
$ curl \
  --request DELETE \
  --header "X-Vault-Token: ..." \
  https://vault.hashicorp.rocks/v1/sys/policy/my-policy
```

This is an idempotent operation. Vault will not return an error when deleting a policy that does not exist.

Associating Policies

Vault can automatically associate a set of policies to a token based on an authorization. This configuration varies significantly between authentication backends. For simplicity, this example will use Vault's built-in userpass auth method.

A Vault administrator or someone from the security team would create the user in Vault with a list of associated policies:

```
$ vault write auth/userpass/users/sethvargo \
    password="s3cr3t!" \
    policies="dev-readonly,logs"
```

This creates an authentication mapping to the policy such that, when the user authenticates successfully to Vault, they will be given a token which has the list of policies attached.

The user wishing to authenticate would run

```
$ vault login -method="userpass" username="sethvargo"
Password (will be hidden): ...
```

If the provided information is correct, Vault will generate a token, assign the list of configured policies to the token, and return that token to the authenticated user.

Tokens

Tokens are associated with their policies at creation time. For example:

```
$ vault token create -policy=dev-readonly -policy=logs
```

Child tokens can be associated with a subset of a parent's policies. Root users can assign any policies.

There is no way to modify the policies associated with a token once the token has been issued. The token must be revoked and a new one acquired to receive a new set of policies.

However, the *contents* of policies are parsed in real-time whenever the token is used. As a result, if a policy is modified, the modified rules will be in force the next time a token, with that policy attached, is used to make a call to Vault.