Lab 4: Working with Policies

Duration: 30 minutes

This lab demonstrates the policy authoring workflow.

- Task 1: Create a Policy
- Task 2: Test the "base" Policy
- Challenge: Create and Test Policies

Task 1: Create a policy

In reality, first, you gather policy requirements, and then author policies to meet the requirements. In this task, you are going to write an ACL policy (in HCL format), and then create a policy in Vault.

Step 4.1.1

Make sure that you are logged in with root token:

```
$ vault login root
```

Change directory into `/workstation/vault` if you have not done so already:

```
$ cd /workstation/vault
```

Let's review the policy file, `base.hcl`

```
$ cat base.hcl
```

Step 4.1.2

The policy defines the following rule:

```
path "secret/data/training_*" {
   capabilities = ["create", "read"]
}
```

Notice that the path has the "splat" operator (`training_*`). This is helpful in working with namespace patterns.

> **NOTE:** When you are working with *key/value secret engine v2*, the path to write policy would be **`secret/data/<path>`** even though the CLI command to the path is

`secret/<path>`. When you are working with **v1**, the policy should be written against **`secret/<path>`**. This is because the API endpoint to invoke key/value v2 is different from v1.

Step 4.1.3

Get help for the vault policy command:

```
$ vault policy -h
```

Step 4.1.4

Execute the following commands to create a policy:

```
$ vault policy write base base.hcl
```

Step 4.1.5

Execute the following CLI command to list existing policy names:

```
$ vault policy list
```

Expected output:

```
base
default
root
```

Step 4.1.6

Execute the following commands to read a policy:

```
$ vault policy read base
```

The output should display the `base` policy rule.

Step 4.1.7

To view the default policy, execute the following:

```
$ vault policy read default
```

Step 4.1.8

Create a token attached to the newly created base policy so that you can test it. Execute the following commands to create a new token:

```
$ vault token create -policy="base"
```

Expected output:

```
Key                    Value
---                    -----
token                  4a56ffd0-1a78-f32c-798f-62b94d14e731
token_accessor         f5009bbf-07a8-b99e-77d6-4b6ccebe481d
token_duration         768h
token_renewable        true
token_policies         ["base" "default"]
identity_policies      []
policies               ["base" "default"]]
```

NOTE: Every token automatically gets default policy attached.

Copy the generated token.

Step 4.1.9

Authenticate with Vault using the token generated at *Step 4.1.8*:

Example:

```
$ vault login ce3bd491-2533-7a32-9526-f0ea83c6a68a
```

Expected output:

```
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                    Value
---                    -----
token                  4a56ffd0-1a78-f32c-798f-62b94d14e731
token_accessor         f5009bbf-07a8-b99e-77d6-4b6ccebe481d
token_duration         767h56m35s
token_renewable        true
token_policies         ["base" "default"]
identity_policies      []
policies               ["base" "default"]
```

Question

What happens when you try to list existing policy names?

Task 2: Test the "base" Policy

Now that you have created a new policy, let's test to verify its effect on a token.

Step 4.2.1

Using the base token, you have a very limited permissions.

```
$ vault policy list
Error listing policies: Error making API request.

URL: GET http://127.0.0.1:8200/v1/sys/policy
Code: 403. Errors:

permission denied
```

The base policy does not have a rule on `sys/policy` path. Lack of policy means no permission on that path. Therefore, returning the *permission denied* error is the expected behavior.

Step 4.2.2

Now, try writing data into the key/value secret backend at `secret/dev` path.

```
$ vault kv put secret/dev password="p@ssw0rd"
Error writing data to secret/dev: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/secret/dev
Code: 403. Errors:

permission denied
```

Again, this should fail.

Step 4.2.3

Now, try writing data to a proper path that the base policy allows.

```
$ vault kv put secret/training_test password="p@ssw0rd"
Key              Value
---              -----
created_time     2018-06-14T17:09:25.888839614Z
deletion_time    n/a
destroyed        false
version          1
```

The policy was written for the `secret/training_*` path so that you can write on `secret/training_test`, `secret/training_dev`, `secret/training_prod`, etc.

Step 4.2.4

Read the data back:

```
$ vault kv get secret/training_test
====== Metadata ======
Key                 Value
---                 -----
created_time        2018-06-01T23:29:11.873255197Z
deletion_time       n/a
destroyed           false
version             1

====== Data ======
Key         Value
---         -----
password    p@ssw0rd
```

Step 4.2.5

Pass a different password value to update it.

```
$ vault kv put secret/training_test password="password1234"
Error writing data to secret/training_test: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/secret/training_test
Code: 403. Errors:

* permission denied
```

This should fail because the base policy only grants "create" and "read". With absence of "update" permission, this operation fails.

Question

What happens when you try to write data in `secret/training_` path?

```
$ vault kv put secret/training_ year="2018"
```

Will this work?

Answer

This is going to work.

```
$ vault kv put secret/training_ year="2018"
Success! Data written to: secret/training_
```

However, this is 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.

Task 3: Check the token capabilities

The `vault token capabilities` command fetches the capabilities of a token for a given path which can be used to troubleshoot an unexpected "permission denied" error. You can review the policy (e.g. "`vault policy read base`"), but if your token has multiple policies attached, you have to review all of the associated policies. If the policy is lengthy, it can get troublesome to find what you are looking for.

Step 4.3.1

Now, authenticate with root token again.

```
$ vault login root
```

Step 4.3.2

Let's view the help message for the `token capabilities` command:

```
$ vault token capabilities -h
```

Note that you can specify the token value to check its capabilities permitted by the attached policies. If no token value is provided, this command checks the capabilities of the locally authenticated token.

Step 4.3.3

Execute the capabilities command to check permissions on `secret/data/training_dev` path.

```
$ vault token capabilities <token> secret/data/training_dev
```

Where the `<token>` is the token you copied at Step 4.1.8.

Expected output:

```
create, read
```

This is because the `base` policy permits "create" and "read" operations on any path starting with secret/data/training_.

Step 4.3.4

Try another path that is **not** permitted by the base policy:

```
$ vault token capabilities <token> secret/data/test
```

Expected output:

```
deny
```

Step 4.3.5

Execute the command without a token:

```
$ vault token capabilities secret/data/training_dev
root
```

With absence of a token, the command checks the capabilities of current token.

Challenge

Author a policy named, `exercise` based on the given policy requirements.

Policy Requirements:

1. Permits create, read, and update anything in paths prefixed with `secret/data/exercise`
2. Forbids any operation against `secret/data/exercise/team-admin` (this is an exception to the requirement #1)
3. Forbids **deleting** anything in paths prefixed with `secret/data/exercise`
4. View existing policies (the endpoint is `sys/policy`)
5. View available auth methods (the endpoint is `sys/auth`)

NOTE: Practice *least privileged*, and don't grant more permissions than necessary.

Hint & Tips:

Refer to online documentation if necessary:

- https://www.vaultproject.io/docs/concepts/policies.html#capabilities
- https://www.vaultproject.io/api/system/policy.html#list-policies
- https://www.vaultproject.io/api/system/auth.html#list-auth-methods

The `audit.log` displays the API endpoint (`path`) and the request `operation` that was sent to Vault via CLI.