

Lab 1: Lab Setup

Duration: 10 minutes

Each student should have received the lab workstation log in information from the instructor. This lab ensures that everyone can connect to the workstation, and verify that a Vault server is running so that vault commands can run against it.

- Task 1: Connect to the Student Workstation
- Task 2: Getting Help
- Task 3: Enable Audit Logging

Task 1: Connect to the Student Workstation

Step 1.1.1

SSH into your workstation using the provided credentials.

```
$ ssh <username>@<workstation_IP_address>  
password: <password>
```

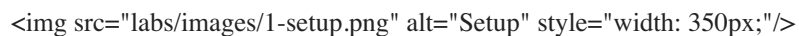
When you are prompted, enter "yes" to continue connecting.

On a Windows, use SSH client such as PuTTY. On a Linux or Mac, use the Terminal to SSH into your workstation.

Alternatively, launch a web browser and enter:

```
http://<workstation_IP_address>/wetty
```

When you are prompted, enter the username and password provided by your instructor.

A placeholder for an image with alt="Setup" and style="width: 350px;"/>

Step 1.1.2

Run the following command to check the Vault server status:

```
$ vault status
```

Key	Value
---	-----
Seal Type	shamir
Sealed	false
Total Shares	1
Threshold	1
Version	0.11.1
Cluster Name	vault-cluster-c78ba77b
Cluster ID	3e9a12c3-19f2-f20f-7f65-46bc02f0ac19
HA Enabled	false

Notice that the server has been unsealed.

Sealed	false
--------	-------

The server has been started in *dev* mode. When you start a Vault server in dev mode, it automatically unseals the server.

Step 1.1.3

Authenticate with Vault using the root token:

```
$ vault login root
```

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	root
token_accessor	7993db51-1c35-ecc7-6293-6c4279230299
token_duration	∞
token_renewable	false
token_policies	["root"]
identity_policies	[]
policies	["root"]

NOTE: For the purpose of training, we will start slightly insecure and login using the root token. Also, the Vault server is running in *dev* mode.

Task 2: Getting Help

Step 1.2.1

Execute the following command to display available commands:

```
$ vault help
```

Or, you can use short-hand:

```
$ vault -h
```

Step 1.2.2

Get help on vault server commands:

```
$ vault server -h
```

The help message explains how to start a server and its available options.

As you verified at Step 1.1.2, the Vault server is already running. The server was started using the command described in the help message: `vault server -dev -dev-root-token-id="root"`

Step 1.2.3

Get help on the `read` command:

```
$ vault read -h
```

This command reads a secret from a given path.

Step 1.2.4

To get help on the API, the help command becomes `path-help` instead:

```
$ vault path-help sys/policy
```

The key/value secret backend is mounted on `secret/` path.

Task 3: Enable Audit Logging

Audit backend keeps a detailed log of all requests and responses to Vault. Sensitive information is obfuscated by default (HMAC). Prioritizes safety over availability.

Step 1.3.1

Change directory into /workstation/vault

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

Step 1.3.2

Get help on the `audit enable` command:

```
$ vault audit enable -h
```

Step 1.3.3

Let's write audit log in current working directory so that you can inspect as you go through other labs.

Execute the following command to enable audit logging:

```
$ vault audit enable file \  
    file_path=/workstation/vault/audit.log
```

Expected output:

```
Success! Enabled the file audit device at: file/
```

Step 1.3.4

You can verify that the audit log file is generated:

```
$ sudo cat audit.log
```

However, at this point, its content is hard to read. You can pipe the output with `jq` tool.

```
$ sudo cat audit.log | jq  
...  
  "request": {  
    "id": "0f2fb5fd-6a74-f425-9537-2c6d4283b7b8",  
    "operation": "read",  
    "client_token": "hmac-sha256:85a4130cf4527b8bc5...",  
    "client_token_accessor": "hmac-sha256:7dcfaabb1c...",  
    "path": "secret/company",  
    "data": null,  
    "policy_override": false,  
  }  
...
```

Sensitive information such as client token is obfuscated by default (HMAC).

Optional

Often times, the logged information can help you understand what is going on with each command. Invoke the following command to generate a raw log:

```
# Remove the old log
$ vault audit disable file
$ rm audit.log

$ vault audit enable file file_path=/workstation/vault/audit.log \
  log_raw=true
```

If you want to tail the log as you go through hands-on labs, you can open another terminal, and run the following command:

```
$ sudo tail -f audit.log | jq
```

End of Lab 1

Lab 2: Static Secrets

Duration: 25 minutes

This lab demonstrates both CLI commands and API to interact with key/value and cubbyhole secret engines.

- Task 1: Write Key/Value Secrets using CLI
- Task 2: List Secret Keys using CLI
- Task 3: Delete Secrets using CLI
- Task 4: Working with Key/Value Secret Engine using API
- Task 5: Hiding Secrets from History

Task 1: Write Key/Value Secrets using CLI

First, write your very first secrets in the key/value secret engine.

Step 2.1.1

First, check the current version of the key/value secret engine. Execute the following command:

```
$ vault secrets list -detailed
```

In the output, locate "secret/" and check its version under Options.

Path	Type	Accessor	... Options
----	----	-----	... -----
cubbyhole/	cubbyhole	cubbyhole_8f752112	... map[]
identity/	identity	identity_8fb35fba	... map[]
secret/	kv	kv_00c670a4	... map[version:2]
...			

Step 2.1.2

Execute the following command to read secrets at secret/training path:

```
$ vault kv get secret/training
```

Expected output:

```
No value found at secret/training"
```

Step 2.1.3

Write a secret into secret/training path:

```
$ vault kv put secret/training username="student01" password="pAssw0rd"
```

Expected output:

Key	Value
---	-----
created_time	2018-05-02T18:12:33.258249295Z
deletion_time	n/a
destroyed	false
version	1

Step 2.1.4

Now, read the secrets in secret/training path.

```
$ vault kv get secret/training
```

Expected output:

```
===== Metadata =====
Key          Value
---          -
created_time 2018-05-02T18:12:33.258249295Z
deletion_time n/a
destroyed    false
version      1

===== Data =====
Key          Value
---          -
password     pAssw0rd
username     student01
```

Step 2.1.5

Retrieve only the username value from secret/training.

```
$ vault kv get -field=username secret/training
```

Expected output:

```
student01
```

Question

What will happen to the contents of the secret when you execute the following command:

```
$ vault kv put secret/training password="another-password"
```

Answer

Creates another version of the secret.

Key	Value
---	----
created_time	2018-05-02T18:20:18.348234014Z
deletion_time	n/a
destroyed	false
version	2

When you read back the data, username no longer exists!

```
$ vault kv get secret/training

===== Metadata =====
Key          Value
---          -
created_time 2018-05-02T18:20:18.348234014Z
deletion_time n/a
destroyed    false
version      2

===== Data =====
Key          Value
---          -
password     another-password
```

This is very important to understand. The key/value secret engine does NOT merge or add values. If you want to add/update a key, you must specify all the existing keys as well; otherwise, *data loss* can occur!

Step 2.1.6

If you wish to partially update the value, use `patch`:

```
$ vault kv patch secret/training course="Vault 101"
```

This time, you should see that the `course` value is added to the existing key.


```
$ vault kv get secret/training
...
===== Data =====
Key          Value
---          -
course       Vault 101
password     another-password
```

Step 2.1.7

Review a file named, `data.json` in the `/workstation/vault` directory:

```
$ cat data.json
{
  "organization": "hashicorp",
  "region": "US-West",
  "zip_code": "94105"
}
```

Step 2.1.8

Now, let's upload the data from `data.json`:

```
$ vault kv put secret/company @data.json
```

Read the secret in the `secret/company` path:

```
$ vault kv get secret/company
===== Metadata =====
Key          Value
---          -
created_time  2018-05-02T18:24:52.03750902Z
deletion_time n/a
destroyed     false
version       1

===== Data =====
Key          Value
---          -
organization  hashicorp
region        US-West
zip_code      94105
```

Task 2: List Secret Keys using CLI

Step 2.2.1

Get help on the list command:

```
$ vault kv list -h
```

This command can be used to list keys in a given secret engine.

Step 2.2.2

List all the secret keys stored in the key/value secret backend.

```
$ vault kv list secret
```

Expected output:

```
Keys
----
company
training
```

The output displays only the keys and not the values.

Task 3: Delete Secrets using CLI

Step 2.3.1

Get help on the delete command:

```
$ vault kv delete -h
```

This command deletes secrets and configuration from Vault at the given path.

Step 2.3.2

Delete secret/company:

```
$ vault kv delete secret/company
```

Step 2.3.3

Try reading the `secret/company` path again.

Expected output includes the `deletion_time`:

```
===== Metadata =====
Key                        Value
---                      -
created_time              2018-05-02T18:24:52.03750902Z
deletion_time             2018-05-02T18:46:19.9948457Z
destroyed                 false
version                   1
```

NOTE: To permanently delete secret/company, use `vault kv destroy` or `vault kv metadata delete` commands instead.

Task 4: Working with Key/Value Secret Engine API

In this task, you are going to write, read, and delete secrets in key/value secret engine via API.

Step 2.4.1

To write secrets in the key/value secret engine via API using cURL:

```
$ curl --header "X-Vault-Token: <token>" --request POST \
  --data <data>
  <VAULT_ADDRESS>/v1/secret/data/<path>
```

Refer to the online API documentation for more detail: <https://www.vaultproject.io/api/secret/kv/kv-v2.html>

Check the vault address on your student workstation:

```
$ echo $VAULT_ADDR
```

Expected output:

```
http://127.0.0.1:8200
```

Step 2.4.2

Execute the following cURL command to write data in `secret/apikey/google` path:

```
$ curl --header "X-Vault-Token: root" --request POST \
  --data '{"data": {"apikey": "my-api-key"} }' \
  $VAULT_ADDR/v1/secret/data/apikey/google | jq
```

In this exercise, parsing the output using `jq` tool just for the readability of the JSON response message.

NOTE: If you are tailing the `audit.log` (optional step in Lab 1), you should see the trace log of this API call.

Step 2.4.3

Read the data in `secret/apikey/google` path:

```
$ curl --header "X-Vault-Token: root" \
      $VAULT_ADDR/v1/secret/data/apikey/google | jq
```

Expected output:

```
{
  "request_id": "dda623da-ff4f-7417-f354-4dcfa68cff5e",
  "lease_id": "",
  "renewable": false,
  "lease_duration": 0,
  "data": {
    "data": {
      "apikey": "my-api-key"
    },
    "metadata": {
      "created_time": "2018-05-02T18:59:24.293039655Z",
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  },
  "wrap_info": null,
  "warnings": null,
  "auth": null
}
```

Step 2.4.4

To retrieve the `apikey` value alone:

```
$ curl -s --header "X-Vault-Token: root" \
      $VAULT_ADDR/v1/secret/data/apikey/google | jq ".data.data.apikey"
```

Step 2.4.5

Delete the latest version of `secret/apikey/google` using API.

```
$ curl --header "X-Vault-Token: root" \  
      --request DELETE \  
      $VAULT_ADDR/v1/secret/data/apikey/google
```


Challenge

How can an organization protect the secrets in `secret/data/certificates` from being unintentionally overwritten?

Hint:

- *Check-and-Set* parameter:
<https://www.vaultproject.io/docs/secrets/kv/kv-v2.html#writing-reading-arbitrary-data>
- Check the command options: `vault kv put -h`

Lab 2: Static Secrets - Challenge Solution

Challenge: Protect secrets from unintentional overwrite

You have a couple of options:

- Option 1: Enable check-and-set at the `secret/data/certificates` level
- Option 2: Remind everyone to pass the `-cas` flag with every write operation

Option 1

Enable check-and-set at the `secret/data/certificates`:

```
$ vault kv metadata put -cas-required secret/certificates
```

This ensures that every write operation must pass the `-cas` flag.

Example:

```
$ vault kv put secret/certificates root="certificate.pem"
```

Error writing data to secret/data/certificates: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/secret/data/certificates
Code: 400. Errors:

* check-and-set parameter required for this call

In absence of the `-cas` flag, the write operation fails.

```
$ vault kv put -cas=0 secret/certificates root="certificate.pem"
```

Key	Value
---	----
created_time	2018-06-11T21:59:06.055765168Z
deletion_time	n/a
destroyed	false
version	1

If you re-run the same command:

```
$ vault kv put -cas=0 secret/certificates root="certificate.pem"
```

Error writing data to secret/data/certificates: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/secret/data/certificates

Code: 400. Errors:

* check-and-set parameter did not match the current version

Since `-cas=0` allows the write operation only if there is no secret already exists at `secret/certificates`.

Option 2

Make sure that everyone to pass the `-cas` flag with every write operation":

```
$ vault kv put -cas=1 secret/certificates root="certificate.pem"
```

The down side of this is that there will be no warning if one forgets to pass the `-cas` flag.

To learn more about the versioned key/value secret engine, refer to the *Versioned Key/Value Secret Engine* guide at <https://www.vaultproject.io/guides/secret-mgmt/versioned-kv.html>.

End of Lab 2

Lab 3: Cubbyhole Secret Engine

Duration: 10 minutes

This lab demonstrates both CLI commands and API to interact with key/value and cubbyhole secret engines.

- Task 1: Test the Cubbyhole Secret Engine using CLI
- Task 2: Trigger a response wrapping
- Task 3: Unwrap the Wrapped Secret

Task 1: Test the Cubbyhole Secret Engine using CLI

Step 3.1.1

To better demonstrate the cubbyhole secret engine, first create a non-privileged token.

```
$ vault token create -policy=default
```

Expected output look similar to:

Key	Value
---	-----
token	da773bfc-24bd-a364-4cce-46560c4fdcf1
token_accessor	4f536d51-5084-25f1-3bb6-9ae2e4ecfcf9
token_duration	768h
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Step 3.1.2

Log into Vault using the newly generated token:

```
$ vault login <token>
```

Example:


```
$ vault login 9c247c5d-c2be-2ba2-c450-34f33f668ecf
```

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	697d340d-1d78-7497-cb97-72bb6b3f42a5
token_accessor	aa2e40ff-30be-1810-4bf2-2c151e4f4782
token_duration	767h59m46s
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Step 3.1.3

Execute the following command to write secret in the cubbyhole/private path:

```
$ vault write cubbyhole/private mobile="123-456-7890"
```

Step 3.1.4

Read back the secret you just wrote. It should return the secret.

```
$ vault read cubbyhole/private
```

Key	Value
---	-----
mobile	123-456-7890

Step 3.1.5

Login with root token:

```
$ vault login root
```

Step 3.1.6

Now, try to read the cubbyhole/private path.

```
$ vault read cubbyhole/private
```

What response did you receive?

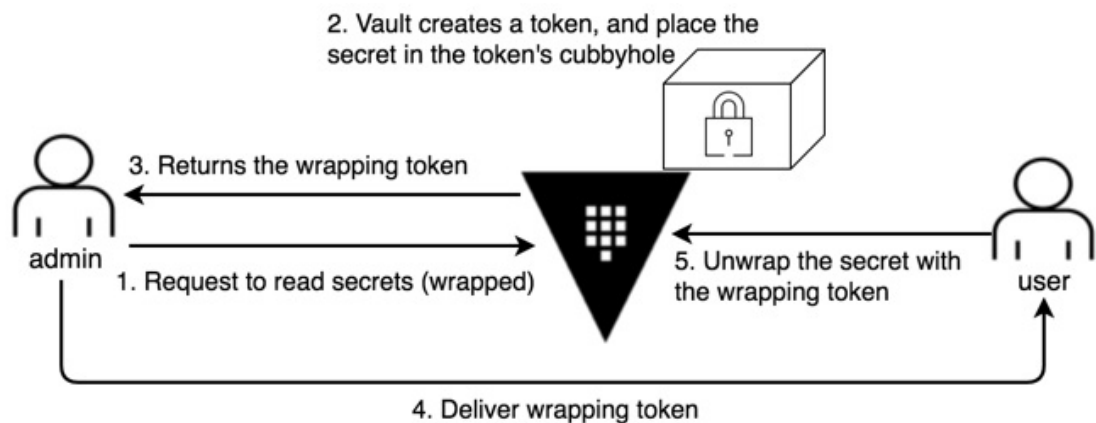
Cubbyhole secret backend provide an isolated secret storage area for an individual token where no other token can violate.

Cubbyhole Wrapping Token

Think of a scenario where a user does not have a permission to read secrets from the `secret/data/training` path. As a privileged user (admin), you have a permission to read the secret in `secret/data/training`.

You can use response wrapping to pass the secret to the non-privileged user.

1. Admin user reads the secret in `secret/data/training` with response wrapping enabled
2. Vault creates a temporal token (wrapping token) and place the requested secret in the wrapping token's cubbyhole
3. Vault returns the wrapping token to the admin
4. Admin delivers the wrapping token to the non-privileged user
5. User uses the wrapping token to read the secret placed in its cubbyhole



Remember that cubbyhole is tied to its token that even the root cannot read it if the cubbyhole does not belong to the root token.

NOTE: A common usage of the response wrapping is to wrap an initial token for a trusted entity to use. For example, an admin generates a Vault token for a Jenkins server to use. Instead of transmitting the token value over the wire, response wrap the token, and let the Jenkins server to unwrap it.

Task 2: Trigger Response Wrapping

Step 3.2.1

Execute the following commands to read secrets using response wrapping with TTL of 360 seconds.

```
$ vault kv get -wrap-ttl=360 secret/training
```

Output should look similar to:

Key	Value
---	-----
wrapping_token:	0a728b26-7db7-3b2b-5c6a-9c09ac073c2e
wrapping_accessor:	e0731c2d-5c5e-fe16-d645-f10b80375bd3
wrapping_token_ttl:	6m
wrapping_token_creation_time:	2018-09-13 23:04:19.856332048 +0000 UTC
wrapping_token_creation_path:	secret/data/training

The response is the wrapping token; therefore, the admin user does not even see the secrets.

Make a note of this `wrapping_token`. You will use it later to unwrap the secret.

Task 3: Unwrap the Wrapped Secret

Since you are currently logged in as a root, you are going to perform the following to demonstrate the apps operations:

1. Create a token with default policy (non-privileged token)
2. Authenticate with Vault using this default token
3. Unwrap the secret to obtain the apps token
4. Verify that you can read `secret/data/dev` using the apps token

Step 3.3.1

Generate a token with default policy:

```
$ vault token create -policy=default
```

Key	Value
---	-----
token	be17b1d0-ca70-d9a6-f44c-c0dfacf3ce37
token_accessor	2f9bda50-8c33-e0c1-ee9a-c2917b3fe8f0
token_duration	768h
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Step 3.3.2

Login with the generated token.

Example:

```
$ vault login be17b1d0-ca70-d9a6-f44c-c0dfacf3ce37
```

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	be17b1d0-ca70-d9a6-f44c-c0dfacf3ce37
token_accessor	2f9bda50-8c33-e0c1-ee9a-c2917b3fe8f0
token_duration	767h58m3s
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Step 3.3.3

Test to make sure that you cannot read the `secret/data/training` path with default token.

```
$ vault kv get secret/training
```

Error making API request.

URL: GET http://127.0.0.1:8200/v1/sys/internal/ui/mounts/secret/training
Code: 403. Errors:

* Preflight capability check returned 403, please ensure client's policies grant access to path "secret/training/"

Step 3.3.4

Now, execute the following commands to unwrap the secret.

```
$ vault unwrap <WRAPPING_TOKEN>
```

Where `<WRAPPING_TOKEN>` is the `wrapping_token` obtained at *Step 3.2.1*.

For example:

```
$ vault unwrap 0a728b26-7db7-3b2b-5c6a-9c09ac073c2e
```

Key	Value
---	-----
data	map[course:Vault 101 password:another-password]
metadata	map[destroyed:false version:4 created_time:2018-09-13T21:29:23.489065342Z deletion_time:]

Since the wrapping token is a single-use token, you will receive an error if you re-run the command.

Step 3.3.5

Log back in as root:

```
$ vault login root
```

Question

What happens to the token if no one unwrap its containing secrets within 360 seconds?

Answer

To test this, generate a new token with short TTL (e.g. 15 seconds):

```
$ vault token create -wrap-ttl=15 -format=json \
  | jq -r ".wrap_info.token" > wrapping-token.txt
```

The above command stores the generated `wrapping_token` in a file.

Wait for 15 seconds and try to unwrap the containing secret:

```
$ vault unwrap $(cat wrapping_token.txt)

Error unwrapping: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/sys/wrapping/unwrap
Code: 400. Errors:

* wrapping token is not valid or does not exist
```

NOTE: The TTL of the wrapping token is separate from the wrapped secret's TTL (in this case, a new token you generated).

Additional Exercises

To learn more about Cubbyhole secret engine, try additional hands-on exercises:

- Cubbyhole Response Wrapping guide:
<https://www.vaultproject.io/guides/secret-mgmt/cubbyhole.html>
- Katacoda scenarios authored by HashiCorp: <https://www.katacoda.com/hashicorp/scenarios/vault-cubbyhole>

End of Lab 3

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

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.

```
{
  "request": {
    "id": "94787d8f-13e7-43f9-7248-9ebf1f2fb318",
    "operation": "create",
    "client_token": "hmac-sha256:a6367110afe4ceb40934c3f946aca339a2f70dc178c2ecc80c08bcf6e371da11",
    "client_token_accessor": "hmac-sha256:a56c1529f524b23e412ce8a547c6b7573a7093bc6d32fbab9b8535058250e54",
    "path": "secret/data/training_",
    "data": {
      "data": {
        "year": "hmac-sha256:a65d8f46e2a4299c05315c748dda3d21f1b012c905f0cfaf60698c8f526ebc7e"
      },
      "options": {}
    },
    "policy_override": false,
    "remote_address": "127.0.0.1",
    "wrap_ttl": 0
  },
  "response": {
    "key": "secret/data/training_dev",
    "value": "root"
  }
}
```

Lab 4: Working with Policies - Challenge Solution

Challenge: Create and Test Policies - Sample Solution

Requirement 3 was a trick question. Vault uses deny-by-default model that no policy means no permission. So, the lack of "delete" in the capability list fulfills this requirement.

exercise.hcl

```
# Requirement 1 and 3
path "secret/data/exercise/*" {
    capabilities = [ "create", "read", "update" ]
}

# Requirement 2
path "secret/data/exercise/team-admin" {
    capabilities = [ "deny" ]
}

# Requirement 4
path "sys/policy" {
    capabilities = [ "read" ]
}

# Requirement 5
path "sys/auth" {
    capabilities = [ "read" ]
}
```

Test the policy for both happy path and failure path.

Example:

```
# Create policy
$ vault policy write exercise ./exercise.hcl

# Generate a new token
$ vault token create -policy=exercise

# Login with the new token
$ vault login <token>

# Test requirement 1
$ vault kv put secret/exercise/test date="today"
$ vault kv get secret/exercise/test
$ vault token capabilities secret/data/exercise/test

# Test requirement 2
$ vault kv put secret/exercise/team-admin status="active"
$ vault token capabilities secret/data/exercise/team-admin

# Test requirement 3
$ vault kv delete secret/exercise/test

# Test requirement 4
$ vault policy list

# Test requirement 5
$ vault auth list

# Finally, log back in with root token
$ vault login root
```

End of Lab 4

Lab 5: Secrets as a Service - Dynamic Secrets

Duration: 25 minutes

This lab demonstrates how Vault generates dynamic credentials for database on-demand.

- Task 1: Enable and Configure a Database Secret Engine
- Task 2: Generate Readonly PostgreSQL Credentials
- Task 3: Revoke Leases
- Challenge: Setup Database Secret Engine via API

The scenario is:

A privileged user (e.g. admin, security team) enables and configures the database secret engine. Also, creates a role which defines what kind of users to generate credentials for. Once the secret engine is set up, the Vault clients (apps, machine, etc.) can request a new set of database credentials. Since the clients don't need the database access for a prolong time, you are going to set its expiration time as well.

Task 1: Enable and Configure a Database Secret Engine

For a production environment, this task is performed by a privileged user.

Step 5.1.1

Most secret engines must be enabled and configured before use. Execute the following command to enable database secret engine:

```
$ vault secrets enable database
```

NOTE: By default, this mounts the database secret engine at `database/` path. If you wish the mounting path to be different, you can pass `-path` to set desired path.

Expected output:

```
Success! Enabled the database secrets engine at: database/
```

Step 5.1.2

Now that you have mounted the database secret engine, you can ask for help to configure it. Use the `path-help` command to display the help message.

```
$ vault path-help database/
```


Also, refer to the online API document: <https://www.vaultproject.io/api/secret/databases/index.html>.

Step 5.1.3

In this lab scenario, you are going to configure a database secret engine for PostgreSQL.

Execute the following command to configure the database secret engine:

```
$ vault write database/config/postgresql \  
    plugin_name=postgresql-database-plugin \  
    allowed_roles=readonly \  
    connection_url=postgresql://postgres@localhost/myapp
```

NOTE: For the purpose of training, PostgreSQL has been installed and a database name, myapp, has been created on each student workstation. It is very common to give Vault the root credentials and let Vault manage the auditing and lifecycle credentials instead of having one person manage it manually.

Step 5.1.4

Notice that you set the `allowed_roles` to be `readonly` in previous step.

Since Vault does not know what kind of PostgreSQL users you want to create. So, you supply the rule with the SQL to run and create the users.

Since this is not a SQL course, we've added the SQL on the student workstation. You can see the script:

```
$ cat readonly.sql  
CREATE ROLE "{{name}}" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL  
    '{{expiration}}';  
REVOKE ALL ON SCHEMA public FROM public, "{{name}}";  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO "{{name}}";
```

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.

Step 5.1.5

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 write database/roles/readonly db_name=postgresql \  
    creation_statements=@readonly.sql \  
    default_ttl=1h max_ttl=24h
```

NOTE: This command creates a role named, `readonly` which has a default TTL of 1 hour, and max TTL is 24 hours. The credentials for `readonly` role expires after 1 hour, but can be renewed multiple times within 24 hours of its creation. This is an example of restricting how long the database credentials should be valid.

Task 2: Generate Read-only PostgreSQL Credentials

As described earlier, privileged users (admin, security team, etc.) enable and configure the database secret engine. Therefore, Task 1 is a task that needs to be completed by the privileged users.

Now that the database secret engine has been enabled and configured, applications (Vault clients) can request a set of PostgreSQL credential to read from the database.

Step 5.2.1

Execute the following command to generate a new set of credentials:

```
$ vault read database/creds/readonly
```

The output should look similar to:

Key	Value
---	-----
lease_id	database/creds/readonly/86a2109c-780c...
lease_duration	1h
lease_renewable	true
password	A1a-u443zy2w14245784
username	v-token-readonly-x271s0zv6x42wsqx...

To generate new credentials, you simply read from the role endpoint.

Step 5.2.2

Copy the `lease_id`. You will use it later.

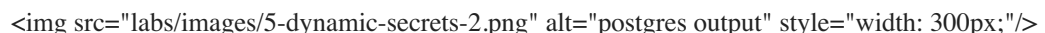
Step 5.2.3

Let's check that the newly generated username exists by logging in as the postgres user and list all accounts.

```
$ psql -U postgres
```

At the `postgres` command prompt, enter `\du` to list all accounts.

```
postgres > \du
```



The username generated at *Step 5.2.1* should be listed.

Notice that the Attributes for your username has "password valid until" clause.

This means that even if an attacker is able to DDoS Vault or take it offline, PostgreSQL will still revoke the credential. When backends support this expiration, Vault will take advantage of it.

Step 5.2.4

Enter \q to exit.

Step 5.2.5

Now, let's renew the lease for this credential.

```
$ vault lease renew <lease_id>
```

While <lease_id> is what you copied at *Step 5.2.2*.

Expected output:

Key	Value
---	-----
lease_id	database/creds/readonly/86a2109c-780c...
lease_duration	1h
lease_renewable	true

The lease duration for this credential is now reset.

For the clients to be able to read credentials and renew its lease, its policy must grants the following:

```
# Get credentials from the database backend
path "database/creds/readonly" {
  capabilities = [ "read" ]
}

# Renew the lease
path "/sys/leases/renew" {
  capabilities = [ "update" ]
}
```

Task 3: Revoke Leases

Under a certain circumstances, the privileged users may need to revoke the existing database credentials.

Step 5.3.1

When the database credentials are no longer in use, or need to be disabled, run the following command:

```
$ vault lease revoke <lease_id>
```

While <lease_id> is what you copied at *Step 5.2.2*.

Expected output:

```
All revocation operations queued successfully!
```

Step 5.3.2

You can verify that the username no longer exists by logging in as postgres user and list all accounts as you did in *Step 5.2.3*.

Step 5.3.3

Let's read a few more credentials from the postgres secret engine. Here, you will simulate a scenario where multiple applications have requested readonly database access.

```
$ vault read database/creds/readonly
Key          Value
---          -
lease_id     database/creds/readonly/563e5e58-aa31-564c-4637-
70804cc63fe1
lease_duration 1h
lease_renewable true
password      A1a-zr9q5t79391w569z
username      v-token-readonly-0306y039q232wvr2y59p-1517945642

$ vault read database/creds/readonly
Key          Value
---          -
lease_id     database/creds/readonly/67fdf769-c28c-eba7-0ac4-
ac9a52f13e4c
lease_duration 1h
lease_renewable true
password      A1a-89q59vqz83z892xs
username      v-token-readonly-74551qs2us5zzqwsqw56-1517945647

$ vault read database/creds/readonly
Key          Value
---          -
lease_id     database/creds/readonly/b422c54b-2664-e0b4-1b6e-
74badbd7ab1c
lease_duration 1h
lease_renewable true
password      A1a-0sw97r2x6s49qv9
username      v-token-readonly-838uu0r2vvzyw0p34qw4-1517945648
```

Now, you have multiple sets of credentials.

Step 5.3.4

Imagine a scenario where you need to revoke all these secrets. Maybe you detected an anomaly in the postgres logs or the vault logs indicates that there may be a breach!

If you know exactly where the root of the problem, you can revoke the specific leases as you performed in *Step 5.3.1*. But what if you don't know!?

Execute the following command to revoke all readonly credentials.

```
$ vault lease revoke -prefix database/creds/readonly
```

Expected output:

```
All revocation operations queued successfully!
```

If you want to revoke all database credentials, run:

```
$ vault lease revoke -prefix database/creds
```


Challenge: Setup Database Secret Engine with API

Perform the same tasks using API.

1. Enable database secret engine at a different path (e.g. `postgres-db/`)
2. Configure the secret engine using the same parameters in Task 1
 - `plugin_name`: postgresql-database-plugin
 - `allowed_roles`: readonly
 - `connection_url`: postgres://postgres@localhost/myapp
3. Create a new role named, `readonly`
 - `db_name`: postgresql
 - `creation_statements`: `readonly.sql`
 - `default_ttl`: 1h
 - `max_ttl`: 24h
4. Generate a new set of credentials for `readonly` role

Hint:

- [Database Secret Engine API doc](#)
- [PostgreSQL Database Secret Plugin HTTP API](#)

Lab 5: Dynamic Secrets - Challenge Solution

Challenge: Setup Database Secret Engine with API - Sample Solution

```
# Enable database secret engine at 'postgres-db'
$ curl --header "X-Vault-Token: root" --request POST \
  --data '{"type": "database"}' \
  $VAULT_ADDR/v1/sys/mounts/postgres-db

# Request message to configure the secret engine
$ tee payload.json <<EOF
{
  "plugin_name": "postgresql-database-plugin",
  "allowed_roles": "readonly",
  "connection_url": "postgresql://postgres@localhost/myapp"
}
EOF

# API call to configure the database secret engine
$ curl --header "X-Vault-Token: root" --request POST \
  --data @payload.json \
  $VAULT_ADDR/v1/postgres-db/config/postgresql

# Request message for creating a role
$ tee payload2.json <<EOF
{
  "db_name": "postgresql",
  "creation_statements": ["CREATE ROLE \"{{name}}\" WITH LOGIN
PASSWORD '{{password}}' VALID UNTIL '{{expiration}}'; REVOKE ALL ON
SCHEMA public FROM public, \"{{name}}\"; GRANT SELECT ON ALL TABLES IN
SCHEMA public TO \"{{name}}\";"],
  "default_ttl": "1h",
  "max_ttl": "24h"
}
EOF

# API call to create a role named 'readonly'
$ curl --header "X-Vault-Token: root" --request POST \
  --data @payload2.json \
  $VAULT_ADDR/v1/postgres-db/roles/readonly

# API call to get a new set of credentials
$ curl --header "X-Vault-Token: root" --request GET \
  $VAULT_ADDR/v1/postgres-db/creds/readonly | jq
```

Lab 6: Authentication and Tokens

Duration: 20 minutes

Almost all operations in Vault requires a token; therefore, it is important to understand the token lifecycle as well as different token parameters that affects the token's lifecycle. This lab demonstrates various token parameters. In addition, you are going to enable userpass auth method and test it.

- Task 1: Create a Short-Lived Tokens
- Task 2: Token Renewal
- Task 3: Create Tokens with Use Limit
- Task 4: Create a Token Role and Periodic Token
- Task 5: Create an Orphan Token
- Task 6: Enable Username and Password Auth Method

Task 1: Create Short-Lived Tokens

When you have a scenario where an app talks to Vault only to retrieve a secret (e.g. API key), and never again. If the interaction between Vault and its client takes only a few seconds, there is no need to keep the token alive for longer than necessary. Let's create a token which is only valid for 30 seconds.

Step 6.1.1

Review the help message on token creation:

```
$ vault token create -h
```

Expected output:

```
Usage: vault token create [options]
```

```
Creates a new token that can be used for authentication. This token will
be created as a child of the currently authenticated token. The
generated token will inherit all policies and permissions of the
currently authenticated token unless you explicitly define a subset
list policies to assign to the token.
```

```
...
```

Step 6.1.2

Execute the following command to create a token whose TTL is 30 seconds:

```
$ vault token create -ttl=30
```

Output should look similar to:

Key	Value
---	-----
token	0dcaa4d1-6b79-2022-e207-b9019152575b
token_accessor	a6ca1ff7-7d03-b707-3240-78ea36cb99b3
token_duration	30s
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]

Notice that the generated token inherits the parent token's policy. For the training, you are logged in with root token. When you create a new token, it inherits the parent token's policy unless you specify with `-policy` parameter.

Copy the token value.

Step 6.1.3

Now, test the token:

```
$ vault token lookup <token>
```

Where `<token>` is the generated token from *Step 6.1.2*.

Example:

```
$ vault token lookup db1bab9a-6660-d1d7-d049-0fabf5c953b0
```

Key	Value
---	-----
accessor	e74365dc-db2a-45e3-5242-c6632159e326
creation_time	1518119249
creation_ttl	30
display_name	token
entity_id	n/a
expire_time	2018-02-08T19:47:59.230435584Z
explicit_max_ttl	0
id	21fec160-744e-09bc-afed-b94d475df80e
issue_time	2018-02-08T19:47:29.23043453Z
meta	<nil>
num_uses	0
orphan	false
path	auth/token/create
policies	[root]
renewable	true
ttl	19

In this example, this token has 19 seconds TTL left before it expires.

Step 6.1.4

Use the upper-arrow key, and then re-run the same command again.

Expected output:

```
Error looking up token: Error making API request.  
  
URL: POST http://127.0.0.1:8200/v1/auth/token/lookup  
Code: 403. Errors:  
  
* bad token
```

After 30 seconds, the token gets revoked automatically, and you can no longer make any request with this token.

Task 2: Token Renewal

Step 6.2.1

Review the help message on token creation:

```
$ vault token renew -h
```

Expected output:

```
Usage: vault token renew [options] [TOKEN]  
  
Renews a token's lease, extending the amount of time it can be used. If  
a TOKEN is not provided, the locally authenticated token is used. Lease  
renewal will fail if the token is not renewable, the token has already  
been revoked,  
or if the token has already reached its maximum TTL.  
...  
  
Command Options:  
  
-increment=<duration>  
    Request a specific increment for renewal. Vault is not  
    required to honor this request. If not supplied, Vault  
    will use the default TTL. This is specified as a  
    numeric string with suffix like "30s" or "5m". This is  
    aliased as "-i".
```

Step 6.2.2

Let's create another token with default policy and TTL of 60 seconds:

```
$ vault token create -ttl=60 -policy="default"
```

Output should look similar to:

Key	Value
---	-----
token	52c3a5d0-2748-0906-dec7-1ace910b8455
token_accessor	d3cbe6fd-ff6e-d111-4047-31ba096cde79
token_duration	1m
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Step 6.2.3

Let's take a look at the token details:

```
$ vault token lookup <token>
```

While <token> is the token from *Step 6.2.2*.

Output should look similar to:

Key	Value
---	-----
accessor	d3cbe6fd-ff6e-d111-4047-31ba096cde79
creation_time	1529610992
creation_ttl	60
display_name	token
entity_id	n/a
expire_time	2018-06-21T19:57:32.332225162Z
explicit_max_ttl	0
id	52c3a5d0-2748-0906-dec7-1ace910b8455
issue_time	2018-06-21T19:56:32.332224318Z
meta	<nil>
num_uses	0
orphan	false
path	auth/token/create
policies	[default]
renewable	true
ttl	32

Step 6.2.4

Renew the token and double its TTL:

```
$ vault token renew -increment=120 <token>
```

While <token> is the token from *Step 5.2.1*.

Output should look similar to:

Key	Value
---	-----
token	d20b3fa5-fd04-6d88-8097-7f3a57f11344
token_accessor	59a63394-2445-85f1-a06b-021a70424a98
token_duration	2m
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Now the token duration is extended to 2 minutes.

Step 6.2.5

Look up the token details again to verify that is TTL has been updated.

```
$ vault token lookup <token>
```

Output should look similar to:

Key	Value
---	-----
accessor	59a63394-2445-85f1-a06b-021a70424a98
creation_time	1529611156
creation_ttl	60
display_name	token
entity_id	n/a
expire_time	2018-06-21T20:01:35.25178929Z
explicit_max_ttl	0
id	d20b3fa5-fd04-6d88-8097-7f3a57f11344
issue_time	2018-06-21T19:59:16.088831304Z
last_renewal	2018-06-21T19:59:35.251790088Z
last_renewal_time	1529611175
meta	<nil>
num_uses	0
orphan	false
path	auth/token/create
policies	[default]
renewable	true
ttl	116

Task 3: Create Tokens with Use Limit

Step 6.3.1

Create a token with use limit of 2.

```
$ vault token create -use-limit=2
```

Output should look similar to:

Key	Value
---	-----
token	9a7ba780-4275-f190-05eb-555456028cd4
token_accessor	5a5131b6-fc9a-102a-7cdc-7414f6730eb3
token_duration	768h
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]

Step 6.3.2

Test the token with use limit.

Example:

```
$ VAULT_TOKEN=<token> vault token lookup
Key          Value
---          -
accessor     355f9e38-5162-a6cb-cc4c-bc5f145c1162
creation_time 1518127372
creation_ttl  2764800
display_name  token
entity_id     n/a
expire_time   2018-03-12T22:02:52.557681218Z
explicit_max_ttl 0
id            62ee52e7-31f4-c59f-ebcf-4b21bf594d34
issue_time    2018-02-08T22:02:52.557680295Z
meta          <nil>
num_uses      1
...

$ VAULT_TOKEN=<token> vault write cubbyhole/test name="student01"
Success! Data written to: cubbyhole/test

$ VAULT_TOKEN=<token> vault read cubbyhole/test
Error making API request.

URL: GET http://127.0.0.1:8200/v1/sys/internal/ui/mounts/cubbyhole/test
Code: 403. Errors:

* permission denied
```

Task 4: Create a Token Role and Periodic Token

A common use case of periodic token is long-running processes where generation of a new token can interrupt the entire system flow. This task demonstrates the creation of a role and periodic token for such long-running process.

Step 6.4.1

Get help on auth/token path:

```
$ vault path-help auth/token
...
## PATHS
...
^roles/(?P<role_name>\w+)?$
    This endpoint allows creating, reading, and deleting
    roles.
...
```

The API endpoint to create a token role is `auth/token/roles`.

Step 6.4.2

First, create a token role named, `monitor`. This role has default policy and token renewal period of 24 hours.

```
$ vault write auth/token/roles/monitor \
    allowed_policies="default" period="24h"
```

Expected output:

```
Success! Data written to: auth/token/roles/monitor
```

Step 6.4.3

Now, create a token for role, `monitor`:

```
$ vault token create -role="monitor"
```

Output should look similar to:

Key	Value
---	-----
token	c1e30bba-492a-fb40-b51a-6db6c85f7e3f
token_accessor	5285b325-9c38-69f6-36cf-5081a3073285
token_duration	24h
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

This token can be renewed multiple times indefinitely as long as it gets renewed before it expires.

Step 6.4.4

Renew the token using API. The endpoint for renewing a token is `auth/token/renew`.

```
$ curl --header "X-Vault-Token: root" --request POST \
    --data '{"token":"<token>"}' \
    $VAULT_ADDR/v1/auth/token/renew | jq
```

Where `<token>` is the generated token at *Step 6.4.3*.

Output should look similar to:

```
{
  "request_id": "a4a92f0b-16c7-9136-5803-4ba8f6cadbef",
  "lease_id": "",
  "renewable": false,
  "lease_duration": 0,
  "data": null,
  "wrap_info": null,
  "warnings": null,
  "auth": {
    "client_token": "5261419d-65ec-c380-60f0-e1f7635dc175",
    "accessor": "d33959e3-39ed-335b-5d41-cdea0323a1a4",
    "policies": [
      "default"
    ],
    "metadata": null,
    "lease_duration": 86400,
    "renewable": true,
    "entity_id": ""
  }
}
```

Task 5: Create an Orphan Token

Step 6.5.1

Create a token with TTL of 60 seconds.

```
$ vault token create -ttl=60
```

Output should look similar to:

Key	Value
---	-----
token	d3f9538e-32d4-bcb1-c982-c335af532d66
token_accessor	cdf7ab42-9b7d-cec5-bae1-fafab6aa9593
token_duration	1m
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]

Step 6.5.2

Using the generated token, create a child token with longer TTL, 120 seconds.

```
$ VAULT_TOKEN=<token> vault token create -ttl=120
```

Output should look similar to:

Key	Value
---	-----
token	89e11854-8fd3-f86b-3862-34157ecf34c7
token_accessor	2671d179-a8a8-bf30-a300-d19aeb5ece50
token_duration	2m
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]

In this example, the token hierarchy is:

```
root
|__ d3f9538e-32d4-bcb1-c982-c335af532d66 (TTL = 60 seconds)
    |__ 89e11854-8fd3-f86b-3862-34157ecf34c7 (TTL = 120 seconds)
```

Step 6.5.3

After *one minute* and let the token from *Step 6.5.1* expires!

This automatically revokes its child token. If you try to look up the child token, you should receive bad token error since the token was revoked when its parent expired.

```
$ vault token lookup <child_token>
```

Now, if this behavior is undesirable, you can create an orphan token instead.

Step 6.5.4

Now, repeat the exercise with `-orphan` flag.

```
$ vault token create -ttl=60

$ VAULT_TOKEN=<token> vault token create -ttl=120 -orphan
```

Now, manually revoke the parent token instead of waiting for it to expire.

```
$ vault token revoke <token>

# Verify that the orphan token still exists
$ vault token lookup <orphan_token>
```


Task 6: Enable Username & Password Auth Method

Now, shift a gear and you are going to enable userpass auth method.

Step 6.6.1

Execute the following command to list which authentication methods have been enabled:

```
$ vault auth list
```

Expected output:

Path	Type	Description
token/	token	token based credentials

Step 6.6.2

Userpass auth method allows users to login with username and password. Execute the CLI command to enable the userpass auth method.

```
$ vault auth enable userpass
```

Now, when you list the enabled auth methods, you should see userpass.

Step 6.6.3

Everything in Vault is path based, and you can enable the same method at multiple paths. The data is isolated by each path, and they are not shared between paths even among the same auth method.

Execute the following command to enable userpass at a different path, training-userpass:

```
$ vault auth enable -path=training-userpass userpass
```

Now, list the enabled auth methods should look like:

```
$ vault auth list
```

Path	Type	Description
token/	token	token based credentials
training-userpass/	userpass	n/a
userpass/	userpass	n/a

Step 6.6.4

Let's create your first user.

```
$ vault write auth/userpass/users/<user_name> \
    password="training" policies="default"
```

While <user_name> is your desired user name.

Notice that the username is a part of the path and the two parameters are password (in plain-text) and the list of policies as comma-separated value.

Example:

```
$ vault write auth/userpass/users/student01 \
    password="training" policies="default"
Success! Data written to: auth/userpass/users/student01
```

Step 6.6.5

You can test it.

```
$ vault login -method=userpass username=<user_name> \
    password="training"
```

When you successfully authenticate with Vault using your username and password, Vault returns a token. From then on, you can use this token to make API calls and/or run CLI commands.

Example:

```
$ vault login -method=userpass username="student01" \
    password="training"
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	9c8f144f-e7d3-1847-dc68-67436648b9f4
token_accessor	e0262587-da0e-a3fd-a034-4f6ee4267248
token_duration	768h
token_renewable	true
token_policies	[default]
token_meta_username	student01

Step 6.6.6

Log back in with the root token.

```
$ vault login root
```

End of Lab 6

Lab 7: Entities and Groups

Duration: 30 minutes

In this lab, you are going to learn the API-based commands to create entities, entity aliases, and groups. For the purpose of the training, you are going to leverage the userpass auth method. The challenge exercise walks you through creating an external group by mapping a GitHub group to an identity group.

- Task 1: Create an Entity with Alias
- Task 2: Test the Entity
- Task 3: Create an Internal Group
- Task 4: Test the Internal Group
- Challenge: Create an External Group and Group Alias

NOTE: The challenge exercise requires a GitHub account.

Task 1: Create an Entity with Alias

You are going to create a new entity with base policy assigned. The entity defines two entity aliases with each has a different policy assigned.

Scenario: A user, Bob Smith at ACME Inc. happened to have two sets of credentials: **bob** and **bsmith**. To manage his accounts and link them to identity **Bob Smith** in team, QA, you are going to create an entity for Bob.

For the purpose of training, you are going to work with the userpass auth method. But in reality, the user **bob** might be a username exists in Active Directory, and **bsmith** might be Bob's username in GitHub.

Step 7.1.1

Execute the following command to list the enabled auth methods:

```
$ vault auth list
```

Expected output:

Path	Type	Description
token/	token	token based credentials
training-userpass/	userpass	n/a
userpass/	userpass	n/a

If you don't see the userpass auth method listed, be sure to enable it before resume.

```
$ vault auth enable userpass
```

Step 7.1.2

Next, list all policies:

```
$ vault policy list
```

Expected output:

```
base
default
exercise
root
```

If you don't have the `base` policy, be sure to create it:

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

NOTE: It is fine if you don't have `exercise` policy.

Step 7.1.3

Create a new policy named, `test`:

```
$ vault policy write test ./test.hcl
```

The policy file, `test.hcl` can be found in the `/workstation/vault` directory.

Step 7.1.4

Create another policy named, `team-qa`:

```
$ vault policy write team-qa ./team-qa.hcl
```

The policy file, `team-qa.hcl` can be found in the `/workstation/vault` directory.

Step 7.1.5

At this point, you should have `base`, `test`, and `team-qa` policies:

```
$ vault policy list
base
default
exercise
team-qa
test
root
```

Step 7.1.6

Create a new user in userpass backend:

- username: bob
- password: training
- policy: test

```
$ vault write auth/userpass/users/bob password="training" \
  policies="test"
```

Expected output:

```
Success! Data written to: auth/userpass/users/bob
```

Step 7.1.7

Create another user in userpass backend:

- username: bsmith
- password: training
- policy: team-qa

```
$ vault write auth/userpass/users/bsmith \
  password="training" \
  policies="team-qa"
```

Expected output:

```
Success! Data written to: auth/userpass/users/bsmith
```

Step 7.1.8

Execute the following command to discover the mount accessor for the userpass auth method:

```
$ vault auth list -detailed
```

In the output, locate the Accessor value for userpass (not the training-userpass):

Path	Type	Accessor	...
----	----	-----	...
token/	token	auth_token_bec8530a	...
training-userpass/	userpass	auth_userpass_e63addf6	...
userpass/	userpass	auth_userpass_70eba76b	...

For convenience, save the accessor value for userpass in a file named, `accessor.txt` by executing the following command.

```
$ vault auth list -format=json \
  | jq -r '["userpass/"].accessor' > accessor.txt
```

Step 7.1.9

Execute the following command to create a new entity named, "bob-smith" and save its entity ID in `entity_id.txt` for later use.

```
$ vault write -format=json identity/entity name="bob-smith" \
  policies="base" \
  metadata=organization="ACME Inc." \
  metadata=team="QA" \
  | jq -r ".data.id" > entity_id.txt
```

Notice that the metadata are passed in `metadata=<key>=<value>` format. In the above command, the entity has organization and team as its metadata.

For convenience, the above command used `jq` to parse the resulting JSON output, retrieved the entity ID, and saved it in a file. Therefore, you did not see the actual response.

The command returns the entity ID as follow:

Key	Value
---	-----
aliases	<nil>
id	631256b1-8523-9838-5501-d0a1e2cdad9c

The `id` is the entity ID.

Step 7.1.10

Now, add the user `bob` to the `bob-smith` entity by creating an entity alias:

```
$ vault write identity/entity-alias name="bob" \
  canonical_id=$(cat entity_id.txt) \
  mount_accessor=$(cat accessor.txt)
```

The name, "bob" is the username you created in the userpass at *Step 7.1.6*.

The output should look similar to:

Key	Value
---	----
canonical_id	631256b1-8523-9838-5501-d0a1e2cdad9c
id	873f7b12-dec8-c182-024e-e3f065d8a9f1

Step 7.1.11

Repeat the step to add user bsmith to the bob-smith entity.

Example:

```
$ vault write identity/entity-alias name="bsmith" \
  canonical_id=$(cat entity_id.txt) \
  mount_accessor=$(cat accessor.txt)
```

Step 7.1.12

Review the entity details by:

```
$ vault read identity/entity/id/$(cat entity_id.txt)
```

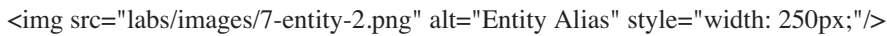
By passing the `-format=json` flag, the output will be printed in JSON format.

```
$ vault read -format=json identity/entity/id/$(cat entity_id.txt) | jq
```

The output includes the entity aliases, metadata (organization, and team), and base policy.

Task 2: Test the Entity

In Task 1, you defined entity and entity aliases. To better understand how a token inherits the capabilities from entity's policy, you are going to test it by logging in as bob.



The diagram shows a box labeled "Entity Alias" with a width of 250px.

Step 7.2.1

Login as bob:

```
$ vault login -method=userpass username=bob \
  password=training
```

Output should look like:

Key	Value
---	-----
token	b0fcf14e-6542-79dd-0232-27b0834dbcbc
token_accessor	38f15139-ace8-e72e-6f5c-3a740a4c24cd
token_duration	768h
token_renewable	true
token_policies	["default" "test"]
identity_policies	["base"]
policies	["base" "default" "test"]
token_meta_username	bob

Step 7.2.2

Remember that the test policy grants CRUD operations on the secret/data/test path. Test to make sure that you can access the path.

For example:

```
$ vault kv put secret/test owner="bob"
Success! Data written to: secret/test

$ vault kv get secret/test
===== Metadata =====
Key          Value
---          -
created_time  2018-09-14T00:17:31.282678321Z
deletion_time n/a
destroyed     false
version       1

==== Data ====
Key      Value
---      -
owner    bob
```

Step 7.2.3

Although the username bob does not have base policy attached, the token inherits the capabilities granted in the base policy because bob is a member of the bob-smith entity, and the entity has base policy attached.

Check to see that the bob's token inherited the capabilities.

For example:

```
$ vault token capabilities secret/data/training_test
create, read
```

Remember that the base policy grants create and read capabilities on `secret/data/training_*` path.

Question

What about the `secret/data/team/qa` path?

Does user `bob` have any permission on that path?

Answer

The user `bob` only inherits capability from its associating entity's policy. The base policy nor test policy grants permissions on the `secret/data/team/qa` path. Only the `team-qa` policy does.

```
$ vault token capabilities secret/data/team/qa
deny
```

Therefore, the current token has no permission to access the `secret/data/team/qa` path.

Step 7.2.4

Repeat the steps and login as a user, `bsmith`. Test to see the token's capabilities.

Step 7.2.5

Log back in with the root token.

```
$ vault login root
```

Task 3: Create an Internal Group

Now, you are going to create an internal group named, `engineers`. Its member is `bob-smith` entity that you created in Task 1.

Step 7.3.1

Let's first create a new policy so that you can test the capability inheritance.

Create a policy named, `team-eng` which grants CRUD operations on the `secret/data/team/eng` path. (The policy file, `team-eng.hcl` is provided.)

```
$ vault policy write team-eng ./team-eng.hcl
```

You can review the created policy:

```
$ vault policy read team-eng
```

Step 7.3.2

Execute the following command to create an internal group named, `engineers` and add `bob-smith` entity as a group member. Also, assign the newly created `team-eng` policy to the group. For later use, parse the JSON output and save the generated group ID in a file named, `group_id.txt`.

```
$ vault write -format=json identity/group name="engineers" \
  policies="team-eng" \
  member_entity_ids=$(cat entity_id.txt) \
  metadata=team="Engineering" \
  metadata=region="North America" \
  | jq -r ".data.id" > group_id.txt
```

Where `<entity_id>` is the value you copied at *Step 7.1.9*.

Due to the extra command (`jq -r ".data.id" > group_id.txt`), you do not see the output. The command returns the group ID and name.

Key	Value
---	-----
id	81bdac90-284a-7b8c-6289-5fa7693bcb4a
name	engineers

Step 7.3.3

Read the details of the group, `qa-engineers`.

```
$ vault read identity/group/id/$(cat group_id.txt)
```

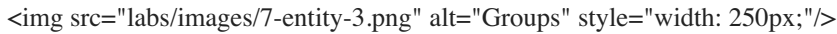
Example output:

Key	Value
---	-----
alias	map[]
creation_time	2018-02-14T22:29:51.324466238Z
id	81bdac90-284a-7b8c-6289-5fa7693bcb4a
last_update_time	2018-02-14T22:29:51.324466238Z
member_entity_ids	[631256b1-8523-9838-5501-d0a1e2cdad9c]
member_group_ids	<nil>
metadata	map[region:North America team:Engineering]
modify_index	1
name	engineers
policies	[team-eng]
type	internal

By default, Vault creates an internal group. When you create an internal group, you specify the group members, so you don't specify any group alias. Group aliases are mapping between Vault and external identity providers (e.g. LDAP, GitHub, etc.). Therefore, you define group aliases only when you create external groups. For internal groups, you have `member_entity_ids` and/or `member_group_ids` instead.

Task 4: Test the Group

Now, test to understand how a token inherits the capabilities from its associating group.



Step 7.4.1

Login as bob with userpass auth method:

```
$ vault login -method=userpass username="bsmith" \
password="training"
```

Step 7.4.2

Test to see if the token has an access to the following paths:

```
secret/data/test
secret/data/training_test
secret/data/team/qa
secret/data/team/eng
```

Challenge: Create an External Group and Group Alias

Instructions

The most common use case is to create external groups each of those groups maps to an external group defined in a third-party identity provider (e.g. Active Directory, OpenLDAP, etc.).

This challenge section requires a GitHub account with a team membership to perform.

Create an external group which maps to a GitHub team that your user account belongs to. For example, if your GitHub username, `sammy22` which is a member of the `training` team in hashicorp organization. Then, create an external group named, `education`, and a group alias named, `training` pointing to the GitHub auth backend (via `github auth mount accessor`).

To find out which GitHub team you belong to:

```
$ curl -H "Authorization: token <your_token>" \
https://api.github.com/user/teams
```

While <your_token> is your GitHub API token. If you do not have one, follow the [GitHub documentation](#) to create one.

The output should look like:

```
[
  {
    "name": "Training",
    "id": 2074701,
    "slug": "training",
    "description": "Training stuff",
    "privacy": "closed",
    "url": "https://api.github.com/teams/2074701",
    ...
  }
]
```

NOTE: You want to use the slugified team name.

Hint:

- [Enable github auth method](#)
- Configure your GitHub team (auth/github/map/teams/<team_name> endpoint)
- [Create a new external group \(identity/group endpoint\)](#)
- [Create a group alias](#) (to get the mount accessor for github, refer to *Step 7.1.8*)

Lab 7: Entities and Groups - Challenge Solution

Sample Solution

```
# Write a new policy file
$ vi education.hcl
path "secret/education" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}

# Create a new policy named 'education'
$ vault policy write education ./education.hcl

# Enable GitHub auth method
$ vault auth enable github

# Configure to point to your GitHub organization (e.g. hashicorp)
$ vault write auth/github/config organization=<org_name>
$ vault write auth/github/config organization=hashicorp

# Get the github auth accessor and save it in github_accessor.txt
$ vault auth list -format=json \
  | jq -r '["github/"].accessor' > github_accessor.txt

# Create a new external group and save the group ID in group_id.txt
$ vault write -format=json identity/group name="education" \
  type=external \
  policies="education" \
  | jq -r ".data.id" > group_id.txt

# Create a group alias where 'training' is the slugfied team name in
# GitHub
$ vault write identity/group-alias name="training" \
  mount_accessor=$(cat github_accessor.txt) \
  canonical_id=$(cat group_id.txt)

# Make sure that you can login with your GitHub API token
$ vault login -method=github token="<your_github_token>"

# Check the permissions on the secret/education path
$ vault token capabilities secret/education

# Check the permissions on the secret/training_test path
$ vault token capabilities secret/training_test
```

End of Lab 7

Lab 8: ACL Policy Path Templating

Duration: 15 minutes

You are going to perform the following tasks:

- Task 1: Create policies with templated paths
- Task 2: Update Entities and Groups
- Task 3: Test the policies

Lab Scenario

In this lab, you are going to write policies which fulfill the following policy requirements:

1. Each *user* can perform all operations on their allocated key/value secret path (`user-kv/data/<user_name>`)
2. The education *group* has a dedicated key/value secret store for each region where all operations can be performed by the group members (`group-kv/data/education/<region>`)
3. The *group* members can update the group information such as metadata about the group (`identity/group/id/<group_id>`)

Available Templating Parameters

You can pass in a policy path containing double curly braces as templating delimiters: `{{<parameter>}}`.

Name	Description
<code>identity.entity.id</code>	The entity's ID
<code>identity.entity.name</code>	The entity's name
<code>identity.entity.metadata.<<metadata key>></code>	Metadata associated with the entity for the given key
<code>identity.entity.aliases.<<mount accessor>>.id</code>	Entity alias ID for the given mount
<code>identity.entity.aliases.<<mount accessor>>.name</code>	Entity alias name for the given mount
<code>identity.entity.aliases.<<mount accessor>>.metadata.<<metadata key>></code>	Metadata associated with the alias for the given mount and metadata key
<code>identity.groups.ids.<<group id>>.name</code>	The group name for the given group ID
<code>identity.groups.names.<<group name>>.id</code>	The group ID for the given group name
<code>identity.groups.names.<<group id>>.metadata.<<metadata key>></code>	Metadata associated with the group for the given key
<code>identity.groups.names.<<group name>>.metadata.<<metadata key>></code>	Metadata associated with the group for the given key

Task 1: Create policies with templated paths

Step 8.1.1

Let's review the policy file, `user-tmpl.hcl`:

```
$ cat user-tmpl.hcl

# Grant permissions on user specific path
path "user-kv/data/{{identity.entity.name}}/*" {
    capabilities = [ "create", "update", "read", "delete", "list" ]
}
```

This policy fulfills the policy requirement 1.

Step 8.1.2

Let's review the policy file, `group-tmpl.hcl`:

```
$ cat group-tmpl.hcl

# Grant permissions on the group specific path
# The region is specified in the group metadata
path "group-
kv/data/team/{{identity.groups.names.engineers.metadata.team}}/*" {
    capabilities = [ "create", "update", "read", "delete", "list" ]
}

# Group member can update the group information for group named,
'engineers'
path "identity/group/id/{{identity.groups.names.engineers.id}}" {
    capabilities = [ "update", "read" ]
}
```

This policy fulfills the policy requirement 2 and 3.

Step 8.1.3

Execute the following command to create `user-tmpl` policy:

```
$ vault policy write user-tmpl ./user-tmpl.hcl
```

Step 8.1.4

Execute the following command to create `group-tmpl` policy:

```
$ vault policy write group-tmpl ./group-tmpl.hcl
```


Step 8.1.5

List the available policies to verify:

```
$ vault policy list
```

Task 2: Update Entities and Groups

Now, you want to add the `user-tmpl` policy to the `bob-smith` entity, and `group-tmpl` policy to the `engineers` group that you created in Lab 7.

Step 8.2.1

Execute the following command to add `user-tmpl` policy to the `bob-smith` entity. (NOTE: In Lab 7, you stored the entity ID in `entity_id.txt`.)

```
$ vault write identity/entity/id/$(cat entity_id.txt) \  
    policies="base, user-tmpl"
```

To verify, you can run the following command:

```
$ vault read -format=json identity/entity/id/$(cat entity_id.txt) | jq  
...  
  "name": "bob-smith",  
  "policies": [  
    "base",  
    "user-tmpl"  
  ]  
...
```

Step 8.2.2

Execute the following command to add `group-tmpl` policy to the `engineers` group. (NOTE: In Lab 7, you stored the group ID in `group_id.txt`.)

```
$ vault write identity/group/id/$(cat group_id.txt) \  
    policies="team-eng, group-tmpl"
```

To verify, you can run the following command:

```
$ vault read -format=json identity/group/id/$(cat group_id.txt) | jq
...
  "policies": [
    "team-eng",
    "group-tmpl"
  ],
  ...
```

Task 3: Test the policies

Now, let's test to make sure that the templated policies work as expected.

Step 8.3.1

First, enable key/value v2 secrets engine at `user-kv` and `group-kv` to match the policy:

```
$ vault secrets enable -path=user-kv kv-v2
$ vault secrets enable -path=group-kv kv-v2
```

Step 8.3.2

Log in as bob.

```
$ vault login -method=userpass username="bob" password="training"
```

Notice that the `identity_policies` includes `user-tmpl` and `group-tmpl` policies.

Key	Value
---	-----
...	
token_policies	["default" "test"]
identity_policies	["base" "group-tmpl" "team-eng" "user-tmpl"]
policies	["base" "default" "group-tmpl" "team-eng" "test"
"user-tmpl"]	
token_meta_username	bob

Step 8.3.3

Since bob is a member of the `bob-smith` entity; therefore, the `user-kv/data/{{identity.entity.name}}/*` expression in the `user-tmpl` policy translates to `user-kv/data/bob-smith/*`.

Run the following command to test:

```
$ vault kv put user-kv/bob-smith/apikey webapp="12344567890"
```

The secret should be created successfully.

Step 8.3.4

The bob-smith entity is a member of engineers group. The engineers group has two metadata: region and team. The team was set to Engineering. Therefore, the group-kv/data/team/{{identity.groups.names.engineers.metadata.team}}/* expression in the group-tmpl policy translates to group-kv/data/team/Engineering/*.

Run the following command to test:

```
$ vault kv put group-kv/team/Engineering/db_pswd \  
password="ABCDEFGHIJKLMN"
```

The secret should be created successfully.

Step 8.3.5

Now, verify that you can update the group information by adding contact_email metadata. The group-tmpl policy permits "update" and "read" on the identity/group/id/{{identity.groups.names.engineers.id}} path. The education group ID is saved in the group_id.txt file.

```
$ vault write identity/group/id/$(cat group_id.txt) \  
metadata=region="North America" \  
metadata=team="Engineering" \  
metadata=contact_email="james@example.com"
```

Read the group information to verify that the data has been updated.

```
$ vault read -format=json identity/group/id/$(cat group_id.txt)  
...  
  "metadata": {  
    "contact_email": "james@example.com",  
    "region": "North America",  
    "team": "Engineering"  
  },  
  ...
```

You should see that contact_email metadata has been added.

Step 8.3.6

Log back in as root:

```
$ vault login root
```

Additional Exercises

To learn more about ACL Policy Path Templating, try additional hands-on exercises:

- ACL Policy Path Templating guide: <https://www.vaultproject.io/guides/identity/policy-templating.html>
- Katacoda scenarios authored by HashiCorp: <https://www.katacoda.com/hashicorp/scenarios/vault-policy-templating>

End of Lab 8