


# Lab 1: Lab Setup

Duration: 10 minutes

This first lab focuses on you connecting to a workstation and exploring a running Vault service.

Using a workstation helps create a safe, common environment to assist in learning and troubleshooting. This workstation also has all the development tools, sample configurations, and a few running services. The instructor should have provided you a workstation address and login credentials.

 If you do NOT have a workstation address and login credentials please ask your instructor. Most are delivered via email from a @hashicorp.com address.

Your goal is to accomplish these four tasks:

- Task 1: Connect to the workstation
- Task 2: Ask Vault for help
- Task 3: Enable audit logging
- Task 4: View Vault via a browser

## Task 1: Connect to the Workstation

### Step 1.1.1

The workstations are running an SSH server in a service with port 22 enabled. The credentials you were provided will grant you access to the workstation.

Launch the terminal session to use the SSH command-line tool. On Windows, launch a powershell session to use the SSH command-line tool or install and launch PuTTY.


SSH into the workstation with provided <username> and <workstation\_address>.

Execute

```
$ ssh <username>@<workstation_address>
```

Next, a warning states the authenticity of this target cannot be established. Enter `yes` to continue.

```
The authenticity of host '<workstation_address>' can't be established.
RSA key fingerprint is SHA256:.....
Are you sure you want to continue connecting (yes/no)?
```

 If you are NOT prompted then your machine may explicitly be denying username/password authentication. You may explicitly bypass that by setting the option `-o` called `PubKeyAuthentication` to `false`.

Execute

```
$ ssh -o PubKeyAuthentication=false <username>@<workstation_address>
```

Finally, enter the user's `<password>` when prompted.

```
...
Warning: Permanently added '<workstation_address>' (RSA) to the list of known hosts.
<username>@<workstation_address>'s password: <password>
```

### Step 1.1.2

A Vault service is already running. The Vault command-line interface (CLI) is installed and accessible on the PATH through the `vault` command. This command has a number of different subcommands and flags. Let's explore the `status` subcommand.


Verify the status of the Vault server.

Execute

```
$ vault status
```

Key	Value
---	----
Seal Type	shamir
Initialized	true
Sealed	false
Total Shares	1
Threshold	1
Version	1.1.3
Cluster Name	vault-cluster-3d3ac31a
Cluster ID	973088a7-2812-1f1d-8f5e-ec8ca714aaf2
HA Enabled	false

The status subcommand reports that the server is currently not sealed (`Sealed false`). The server has been started in `dev` mode. When you start a Vault server in dev mode, it automatically unseals the server.

 Running a Vault server in dev mode, unsealed, is NOT secure! This is done solely in this training environment.

### Step 1.1.3

Authenticate with Vault using the root token:

**Execute**

```
$ vault login root
```

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	6urXl1sr1zQJRUHD95jUzC4P
token_duration	∞
token_renewable	false
token_policies	["root"]
identity_policies	[]
policies	["root"]



Authenticating with the root token is NOT secure! This is done solely in this training environment.

## Task 2: Ask Vault for help

### Step 1.2.1

Display the available help:

**Execute**

```
$ vault help
```

Or, you can use short-hand:

**Execute**

```
$ vault -h
```

### Step 1.2.2

Get help on vault server commands:

**Execute**

```
$ 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:

**Execute**

```
$ vault read -h
```

Notice that there are **HTTP Options** that you can pass since the CLI is invoking the Vault Application Programming Interface (API) underneath. Also, there are **Output Options**.

### Step 1.2.4

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

**Execute**

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

## 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); prioritizing safety over availability.

### Step 1.3.1

Change directory into `/workstation/vault101`

**Execute**

```
$ cd /workstation/vault101
```

### Step 1.3.2

Get help on the `audit enable` command:

#### Execute

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

### Step 1.3.3

Enabling audit logging:

#### Execute

```
$ vault audit enable file \
  file_path=/workstation/vault101/audit.log
```

Success! Enabled the file audit device at: file/

### Step 1.3.4

You can verify that the audit log file is generated:

#### Execute

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

If prompted for password, enter the same `<password>` you used when you SSH'd into the workstation.

The audit log stores output in JSON. To make it easier to read, pipe the output through the JSON formatter tool called `jq`.

#### Execute

```
$ 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).

To disable the audit log, execute the command: `vault audit disable file`

### Optional

Often times, the logged information can help you understand what is going on with each command during the development.

Open another terminal and SSH into the workstation. Enable auditing to a raw log file:

#### Execute

```
$ vault audit enable file -path=file-raw \
  file_path=/workstation/vault101/audit-raw.log log_raw=true
```

Start monitoring the raw log file in real-time:

#### Execute

```
$ sudo tail -f /workstation/vault101/audit-raw.log | jq
```

## Task 4: View Vault via a browser

Vault UI is another useful client interface to interact with Vault.

### Step 1.4.1

Open a web browser and visit `http://<workstation_ip>:8200/ui/vault`.

### Step 1.4.2

In the **Token** field enter **root** and click **Sign in**.

# Sign in to Vault

Token

Username

LDAP

Okta

GitHub

Token

....

Sign In

Contact your administrator for login credentials

End of Lab 1

## Lab 2: Static Secrets

Duration: 20 minutes

In this lab you will employ the Vault CLI and API to interact with the key/value secrets engine.

The kv secrets engine is used to store arbitrary secrets within the configured physical storage for Vault. This backend can be run in one of two modes. It can be a generic Key-Value store that stores one value for a key. Versioning can be enabled and a configurable number of versions for each key will be stored.

- Task 1: Write key/value secrets using CLI
- Task 2: List secret keys using CLI
- Task 3: Delete secrets using CLI
- Task 4: Manage the key/value secret engine using API
- Task 5: Explore web UI exclusive features
- Challenge: Protect secrets from unintentional overwrite

### Task 1: Write key/value secrets using CLI

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

Step 2.1.1

Check the current version of the key/value secret engine:

Execute

\$ 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

Write a secret into `secret/training` path:

Execute

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

Expected output:

Key	Value
---	-----
created_time	2019-03-01T23:28:30.587223947Z
deletion_time	n/a
destroyed	false
version	1

Step 2.1.3

Now, read the secrets in `secret/training` path.

Execute

\$ vault kv get secret/training

Expected output:

```
===== Metadata =====
Key      Value
---      -
created_time 2019-03-01T23:28:30.587223947Z
deletion_time n/a
destroyed    false
version      1

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

#### Step 2.1.4

Retrieve only the **username** value from `secret/training`.

#### Execute

```
$ 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:

#### Execute

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

#### Answer

Vault creates another version of the secret.

```
Key      Value
---      -
created_time 2019-03-01T23:29:12.580401169Z
deletion_time n/a
destroyed    false
version      2
```

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

#### Execute

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

The output should look similar to:

```
===== Metadata =====
Key      Value
---      -
created_time 2019-03-01T23:29:12.580401169Z
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 2.1.5

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

#### Execute

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

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

#### Execute

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

The output should look similar to:

```
...
===== Data =====
Key      Value
---      -
```

course	Vault 101
password	another-password

### Step 2.1.6

Review a file named, `data.json` in the `/workstation/vault101` directory:

#### Execute

```
$ cat data.json
```

The output should look similar to:

```
{
  "organization": "hashicorp",
  "region": "US-West",
  "zip_code": "94105"
}
```

### Step 2.1.7

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

#### Execute

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

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

#### Execute

```
$ vault kv get secret/company
```

The output should look similar to:

```
===== Metadata =====
Key      Value
---      -
created_time  2019-03-01T23:30:35.24991211Z
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:

#### Execute

```
$ 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.

#### Execute

```
$ 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:

**Execute**

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

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

**Step 2.3.2**

Delete `secret/company`:

**Execute**

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

**Step 2.3.3**

Try reading the `secret/company` path again as you did in **Step 2.1.7**.

Expected output includes the `deletion_time`:

```
===== Metadata =====
Key      Value
---      -
created_time 2019-03-01T23:30:35.24991211Z
deletion_time 2019-03-01T23:31:17.090938047Z
destroyed    false
version      1
```

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

**Task 4: Manage the key/value secret engine using API**

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

**Step 2.4.1**

Check the vault address on your student workstation:

**Execute**

```
$ echo $VAULT_ADDR
```

Expected output:

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

**Step 2.4.2**

You have learned how to store secrets under `secret/` using CLI. To find out the equivalent API, you can use `-output-curl-string` flag:

**Execute**

```
$ vault kv put -output-curl-string secret/apikey/google apikey="my-api-key"
```

Expected output:

```
curl -X PUT -H "X-Vault-Token: $(vault print token)" \
-d '{"data":{"apikey":"my-api-key"},"options":{}}' \
http://127.0.0.1:8200/v1/secret/data/apikey/google
```

You can copy and paste the output to invoke the API using cURL.

Use the `jq` tool to parse the output for readability as follow:

**Execute**

```
$ curl -X PUT -H "X-Vault-Token: $(vault print token)" \
-d '{"data":{"apikey":"my-api-key"},"options":{}}' \
http://127.0.0.1:8200/v1/secret/data/apikey/google | jq
```

If you enabled auditing and tailing the `/workstation/vault101/audit.log` (Task 3 in Lab 1), an entry for this API request will appear.

**Step 2.4.3**

Generate the curl command to get the secret at path `secret/apikey/google`:

**Execute**

```
$ vault kv get -output-curl-string secret/apikey/google
```

Get the secret at path `secret/apikey/google`:

### Execute

```
$ curl -H "X-Vault-Token: $(vault print token)" \
  http://127.0.0.1:8200/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

The `jq` tool enables you to filter JSON to retrieve a subset of the content. To retrieve only the value of the `apikey` found within the nested data objects you would run:

### Execute

```
$ curl -H "X-Vault-Token: $(vault print token)" \
  http://127.0.0.1:8200/v1/secret/data/apikey/google | jq ".data.data.apikey"
```

#### Step 2.4.5

Generate the curl command to delete the secret at path `secret/apikey/google`:

### Execute

```
$ vault kv delete -output-curl-string secret/apikey/google
```

Delete the secret at path `secret/apikey/google`:

### Execute

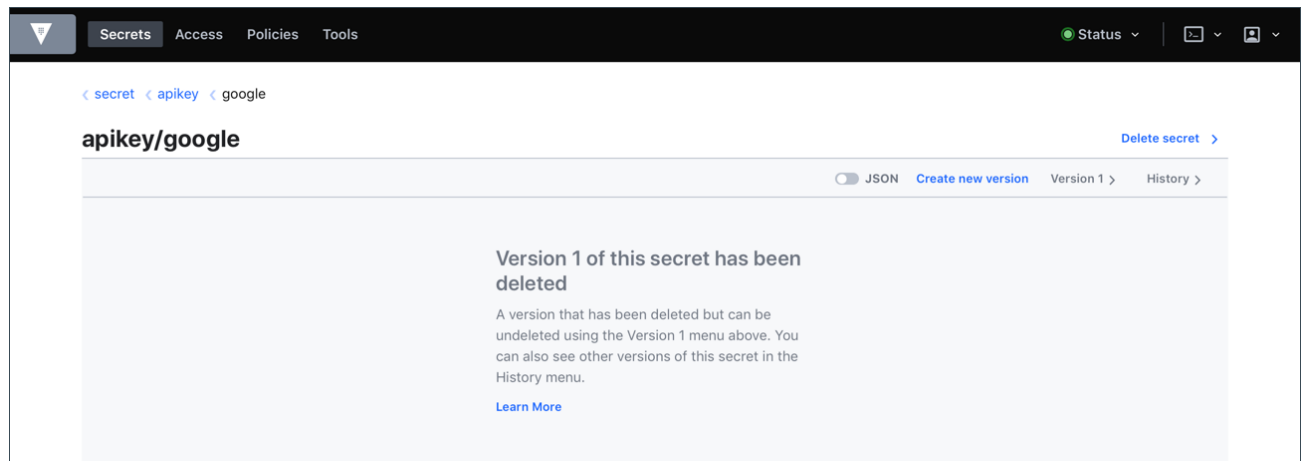
```
$ curl -X DELETE -H "X-Vault-Token: $(vault print token)" \
  http://127.0.0.1:8200/v1/secret/data/apikey/google
```

#### Step 2.4.6

Open a web browser and visit `http://<workstation_ip>:8200/ui/vault`.

In the **Token** field enter **root** and click **Sign in**.

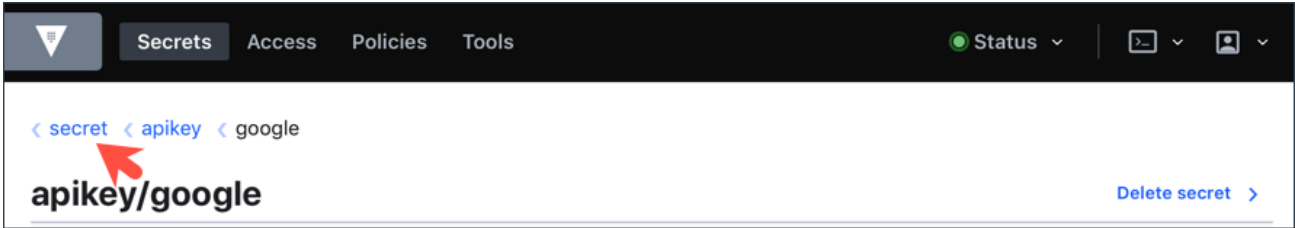
From the **Secrets Engines** list select **secret**, then **apikey** > **google**. Version 1 of this secret has been deleted as a result of the API request.



## Task 5: Explore web UI exclusive features

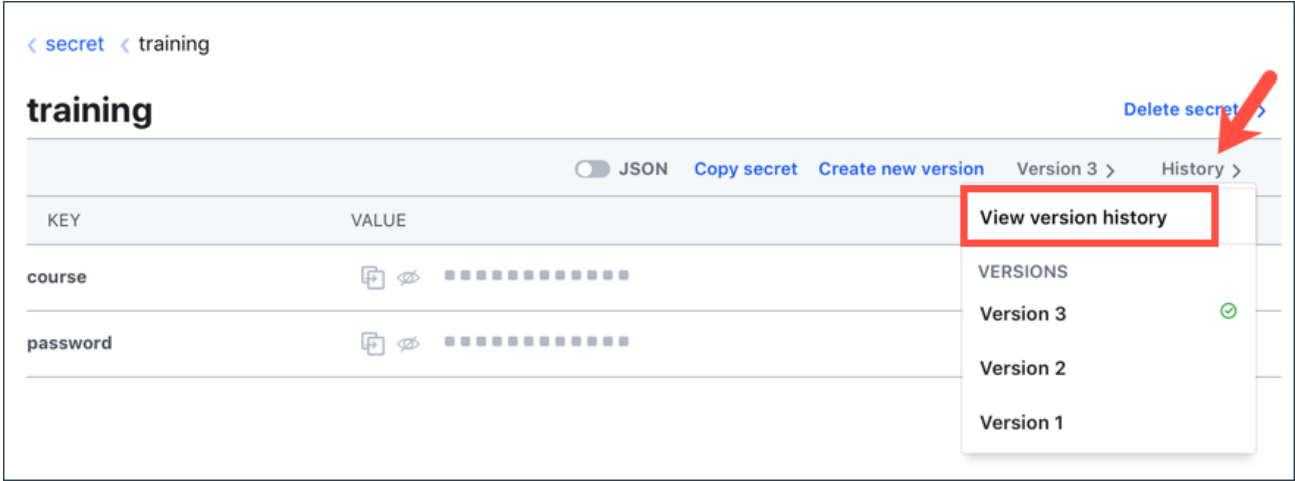
### Step 2.5.1





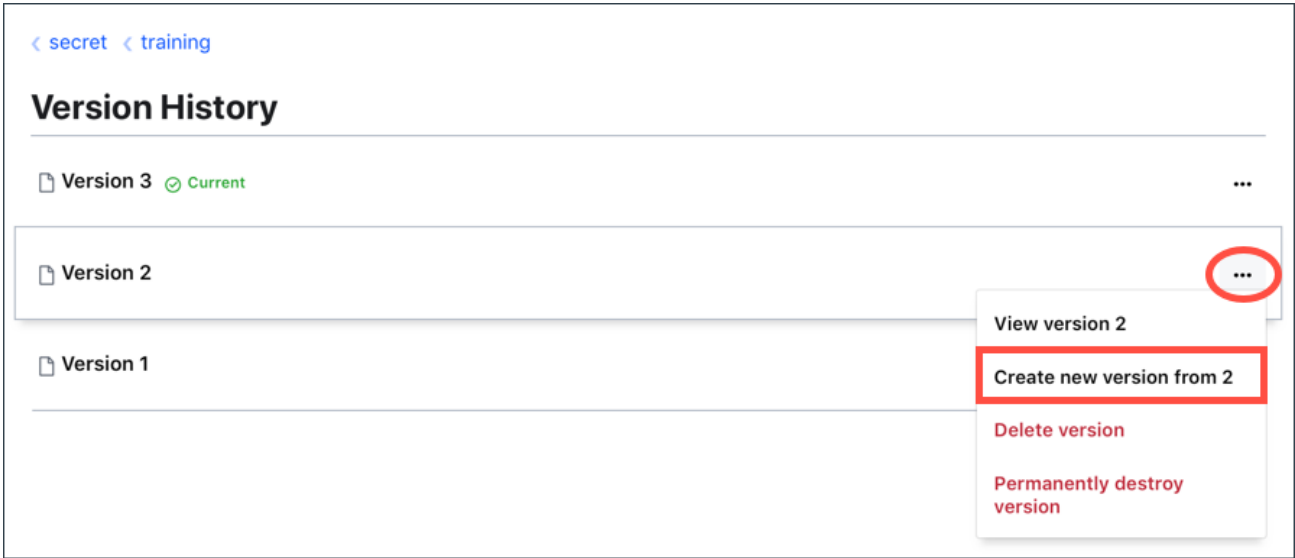
Step 2.5.2

Select **training** and then **History > View version history**.



Step 2.5.3

From the **Version 2** menu, select **Create new version from 2**.



Step 2.5.4

Add a new key called **course** with the value **Vault**.



Click **Save**.  
This creates version 4 of the data.



The `patch` command enabled you to merge new values into the latest version of the key/value secret. Only from the web UI, you can create a new version based on any of the versions. This is particularly useful when you unintentionally modified the data and wish to recover.

## Challenge: Protect secrets from unintentional overwrite

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

- Use the *check-and-set* operation via the `-cas` flag: <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

There are two options:

- **Option 1:** Enable check-and-set at the `secret/data/certificates` level
- **Option 2:** Require all team members to use the `-cas` flag with every write operation

#### Option 1

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

Execute

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

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

Example:

Execute

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

The output should look similar to:

```
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.

### Execute

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

The output should look similar to:

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:

### Execute

```
$ 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

Require all team members to use the `-cas` flag with every write operation:

### Execute

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



If a user forgets to pass the `-cas` flag then the value will be overwritten with not warning.

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: 20 minutes

In this lab you will employ the Vault CLI and API to interact with the cubbyhole secrets engine.

The cubbyhole secrets engine is used to store arbitrary secrets within the configured physical storage for Vault namespaced to a token. In cubbyhole, paths are scoped per token. No token can access another token's cubbyhole. When the token expires, its cubbyhole is destroyed.

- Task 1: Test the cubbyhole secret engine using CLI
- Task 2: Trigger response wrapping
- Task 3: Unwrap the wrapped Secret
- Task 4: Response wrapping via the web UI

### 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.

### Execute

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

Expected output look similar to:

Key	Value
---	----
token	s.FECyDay17PS0g8I4JSuJwYdj
token_accessor	3n00Dvf1106H60AkGMTvJNTx
token_duration	768h
token_renewable	true

```
token_policies ["default"]
identity_policies []
policies ["default"]
```

Step 3.1.2

Copy the value of `token`. This lab will refer to it later as `<token>`.

Step 3.1.3

Log into Vault using the `<token>`:

Execute

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

The output should look similar to:

```
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        s.FECyDayl7PS0g8I4JSuJwYdj
token_accessor 3n0ODvf1106H60AkGMTvJNTx
token_duration 767h59m48s
token_renewable true
token_policies ["default"]
identity_policies []
policies      ["default"]
```

Step 3.1.4

Write a mobile number secret to the `cubbyhole/private` path:

Execute

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

Step 3.1.5

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

Execute

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

The output should look similar to:

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

Step 3.1.6

Login with root token:

Execute

```
$ vault login root
```

Step 3.1.7

Now, try to read the `cubbyhole/private` path.

Execute

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

The Cubbyhole secret backend provides an isolated secret storage area for an individual token that no other token can violate.

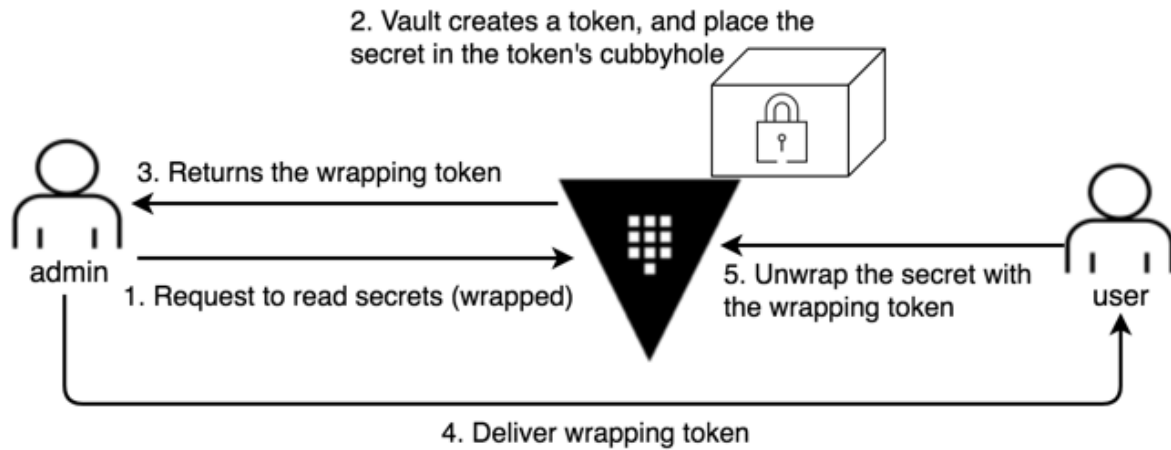
Task 2: Trigger response wrapping

Think of a scenario where you, as a privileged user (admin), have permission to read the secret stored at the path `secret/data/training`. While other non-privileged users **DO NOT** have permission to read the secret stored at that same path.

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

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

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



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.

### Step 3.2.1

Create a response wrapping token with a time-to-live (TTL) set to 360 seconds at the path `secret/training`:

#### Execute

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

Output should look similar to:

Key	Value
---	----
wrapping_token:	s.mieZRgn1hcupCqmXJEdfhnY3
wrapping_accessor:	F5BjzII8j2rHjlcIA5f4nxDN
wrapping_token_ttl:	6m
wrapping_token_creation_time:	2019-02-11 21:28:44.188122677 +0000 UTC
wrapping_token_creation_path:	secret/data/training

### Step 3.3.2

Copy the value of the `wrapping_token`. This lab will refer to it later as `<wrapping_token>`

## 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:

#### Execute

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

Key	Value
---	----
token	s.daroDM01N0NCglvwGwlyIjtS
token_accessor	aClzy3fhM06PlcPQkrD8gVvm
token_duration	768h
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

### Step 3.3.2

Copy the value of the `token`. Later steps in this lab will refer to it as the `<non-privileged-token>`

### Step 3.3.3

Login with the `<non-privileged-token>`.

#### Execute

```
$ vault login <non-privileged-token>
```

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	s.daroDM01N0NCglvwGw1YIjtS
token_accessor	aClzy3fhM06PLcPQkrD8gGVm
token_duration	767h59m23s
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

#### Step 3.3.4

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

#### Execute

```
$ 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 3.3.5

Unwrap the secret using the `<wrapping_token>`

#### Execute

```
$ vault unwrap <wrapping_token>
```

The output should look similar to:

Key	Value
---	-----
data	map[course:Vault 101 password:another-password]
metadata	map[created_time:2019-02-11T21:08:42.098087533Z deletion_time: destroyed:false version:3]

The wrapping token can only be used once. You will receive an error if you attempt to unwrap the same secret with this token.

#### Step 3.3.6

Log back in as root:

#### Execute

```
$ vault login root
```

#### Question

What would happen to the token if no one unwraps its containing secrets within the specified TTL?

#### Answer

Generate a new token with short TTL (e.g. 15 seconds):

#### Execute

```
$ vault token create -wrap-ttl=15 -field=wrapping_token > wrapping-token.txt
```

The above command generates a new wrapping token and directs the returned token value to a file.

Wait 15 seconds and then attempt to unwrap the secret with the token:

#### Execute

```
$ vault unwrap $(cat wrapping-token.txt)
```

The output should look similar to:

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

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

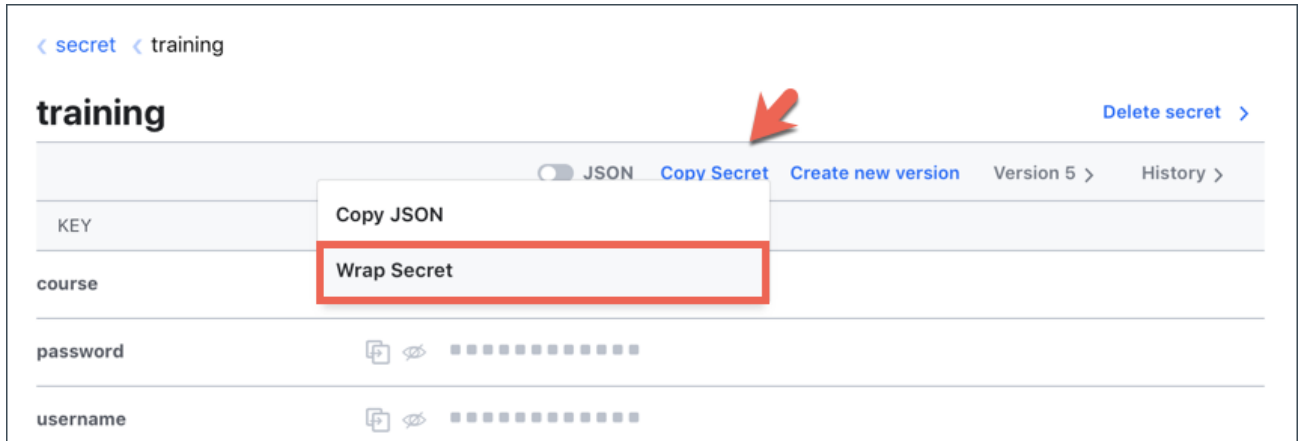
## Task 4: Response wrapping via the web UI

Open a web browser and enter the following address to launch the Vault web UI: `http://<workstation_ip>:8200/ui/vault`.

In the **Token** field enter **root** and click **Sign in**.

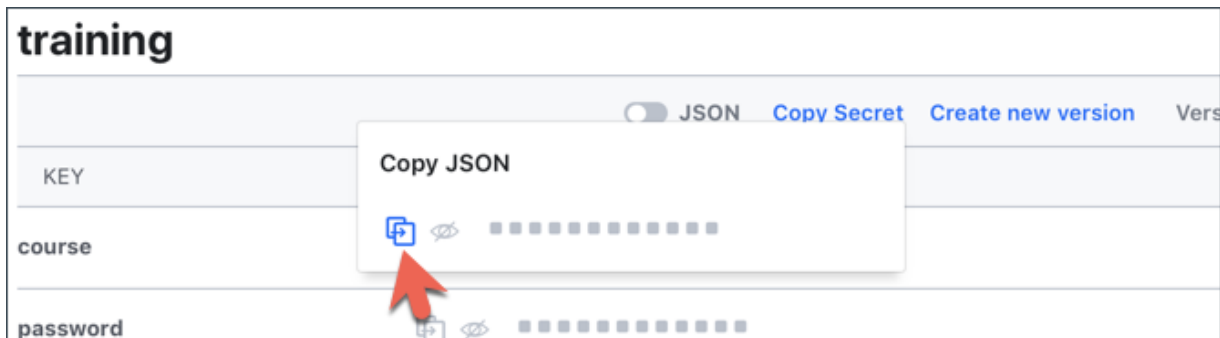
### Step 3.4.1

From the **Secrets Engines** list select **secret > training**. Next, select **Copy Secret > Wrap Secret**.



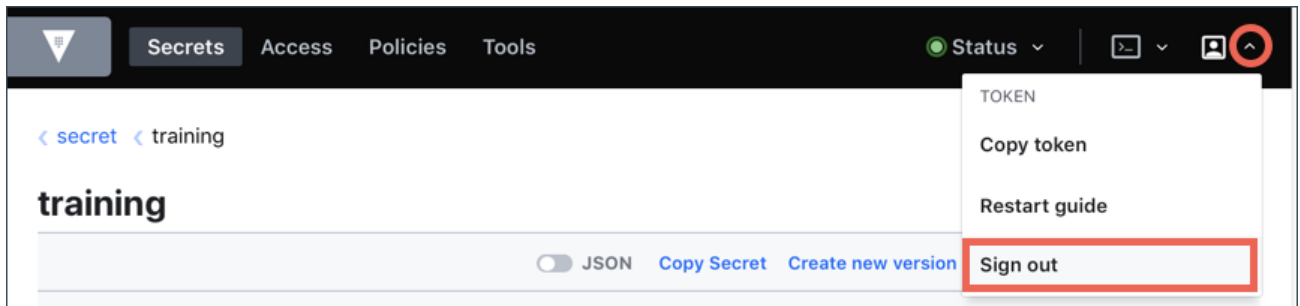
### Step 3.4.2

Next, copy the wrapping token value into your text editor. This lab will refer to it later as `<ui_wrapping_token>`



### Step 3.4.3

Next, sign out of the web UI.



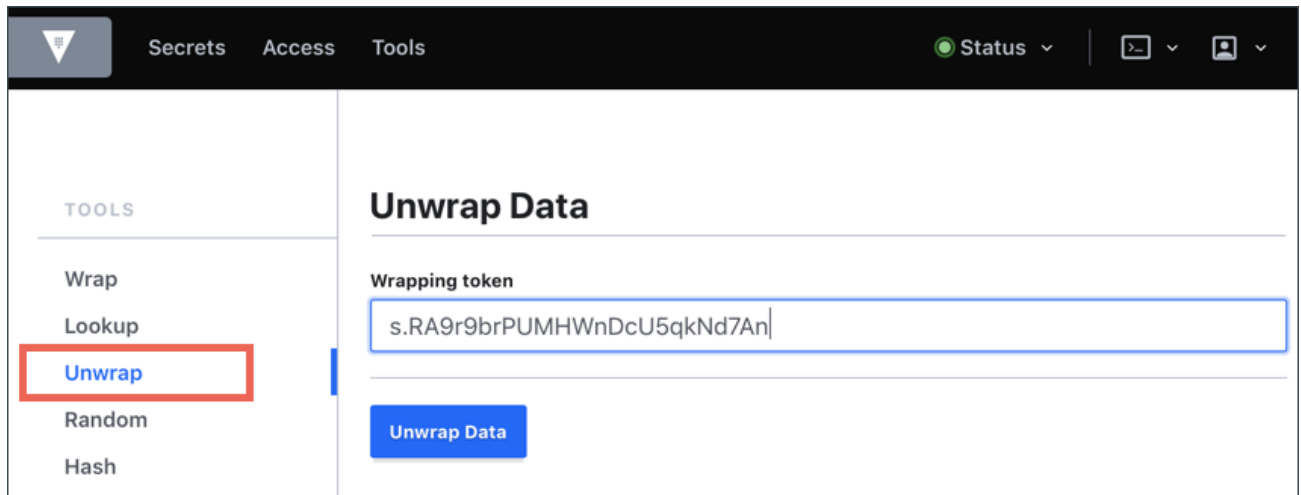
### Step 3.4.4

Now, in the **Token** field enter `<non-privileged-token>`, the token created in **Step 3.3.1**, and click **Sign in**.

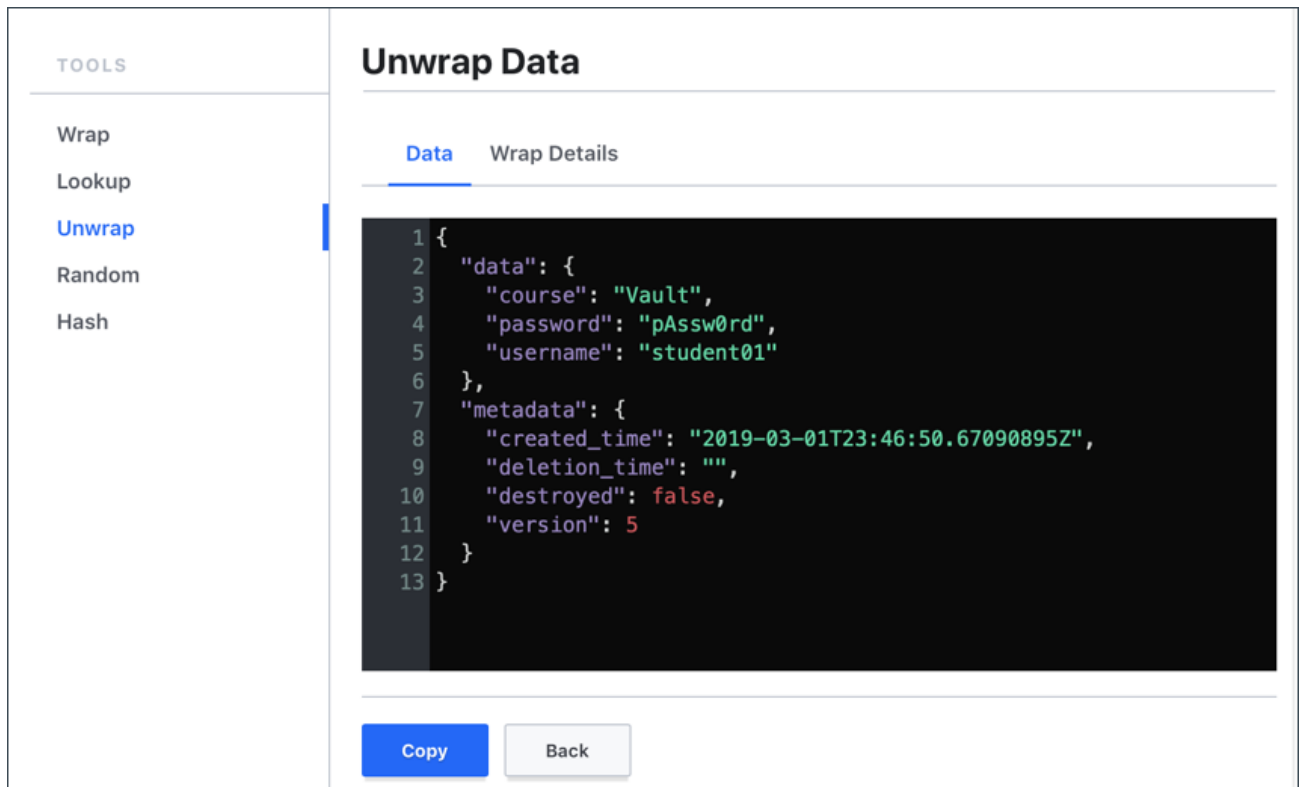
This non-root token does not have a permission to view the `secret/training` path.

### Step 3.4.5

Select **Tools** tab, and then **Unwrap**. Paste in the `<ui_wrapping_token>`.

**Step 3.4.6**

Click **Unwrap Data** to view secret and its metadata.

**Summary**

The Cubbyhole response wrapping allows privileged token to wrap a secret so that non-privilege token can read the secrets once. This does not require any policy change on the non-privilege token. Since you are sending the wrapping token which is a reference to the wrapped secrets and not the actual secrets over the wire, it is more secure.

**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: Database Secrets Engine**

Duration: 25 minutes

This lab demonstrates how you can delegate the database credential management tasks to Vault.

- Task 1: Enable and configure a database secret engine
- Task 2: Create and manage a static role
- Task 3: Generate dynamic readonly credentials
- Task 4: Revoke leases
- Challenge: Setup database secret engine via API



## Task 1: Enable and configure a database secret engine

Most secret engines must be enabled and configured before use. In a production environment, this task is performed by a privileged user.

### Step 4.1.1

Enable the database secrets engine:

#### Execute

```
$ vault secrets enable database
```

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 4.1.2

Now that you have mounted the database secret engine, you can ask help to learn more about how to configure it. Use the `path-help` subcommand to display the available help.

#### Execute

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

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

### Step 4.1.3

PostgreSQL is installed and running as a service on the workstation. A database named, myapp, has been created. It is very common to give Vault the root credentials and let Vault manage the auditing and lifecycle credentials instead managing it manually.

Configure the database secret engine to communicate with PostgreSQL:

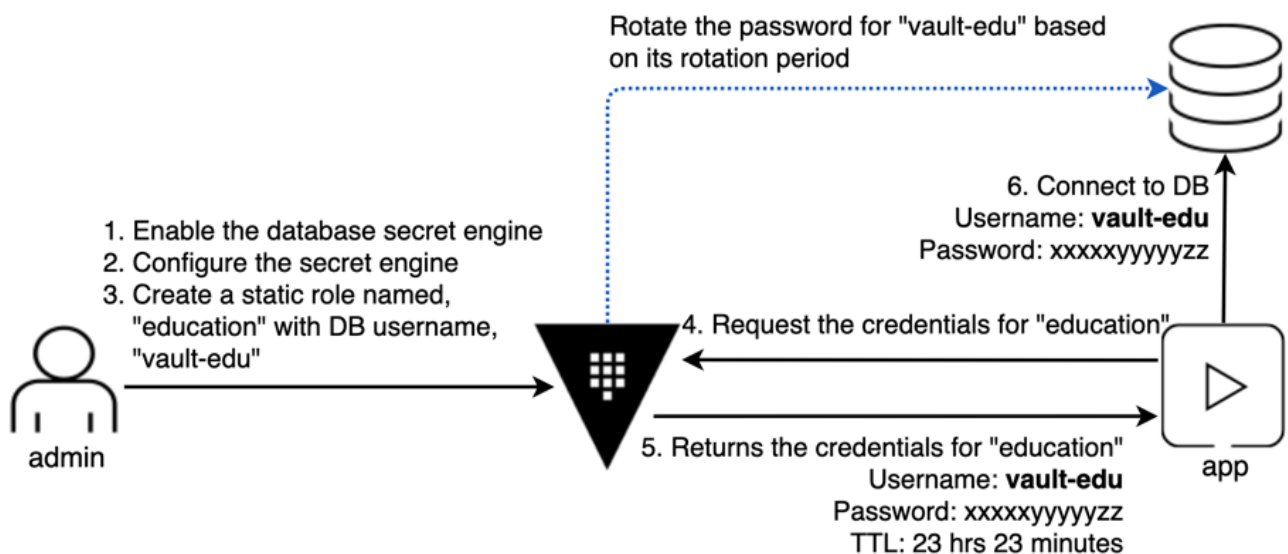
#### Execute

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

Each database defines its own plugin, provides a list of allowed roles (The `*` means all roles), connection, and credentials.

## Task 2: Create and manage a static role

The scenario is:



A database username, "vault-edu" already exists and some applications connect to the database using the credentials. It is required that the password gets rotated every 24 hours.

### Step 4.2.1

First, create a database user named, `vault-edu` with password, `mypassword`.

#### Execute

```
$ psql -U postgres
```

Once you are in the `psql` client execute the following SQL statements:

Execute

```
$ postgres > CREATE ROLE "vault-edu" WITH LOGIN PASSWORD 'mypassword';
postgres > GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO "vault-edu";
```

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

```
postgres > \du
Role name | List of roles | Member of
-----|-----|-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
vault-edu | | {}
```

Enter `\q` to exit.

Step 4.2.2

In order for Vault to rotate the database password, you must provide Vault with the SQL statments necessary to permit it to change passwords.

View the SQL statements within `rotation.sql` :

Execute

```
$ cat rotation.sql
```

The SQL statement allows Vault to alter the password.

```
ALTER USER "{{name}}" WITH PASSWORD '{{password}}';
```

Step 4.2.3

Create a static role, named `education`, with username of `vault-edu`, the SQL statments found in `rotation.sql` and a rotation period of `86400` seconds (1 day).

Execute

```
$ vault write database/static-roles/education \
  db_name=postgresql \
  rotation_statements=@rotation.sql \
  username="vault-edu" \
  rotation_period=86400
```

Step 4.2.4

Verify that the education static-role exists:

Execute

```
$ vault read database/static-roles/education
```

The output should look similar to:

Key	Value
db_name	postgresql
last_vault_rotation	2019-08-08T22:44:07.204054122Z
rotation_period	24h
rotation_statements	[ALTER USER "{{name}}" WITH PASSWORD '{{password}}';]
username	vault-edu

Step 4.2.5

Retrieve static credentials for the `education` role:

Execute

```
$ vault read database/static-creds/education
```

The output should look similar to:

Key	Value
last_vault_rotation	2019-08-08T22:44:07.204054122Z
password	A1a-PrQX2ZJk83ayNbQm
rotation_period	24h
ttl	23h53m42s
username	vault-edu

Re-running the command again returns the same password within the rotation period. In this example, the password is returned for another 23 hours 53 minutes and 42 seconds.

Step 4.2.5

Rotate the password for `education` immediately:

Execute

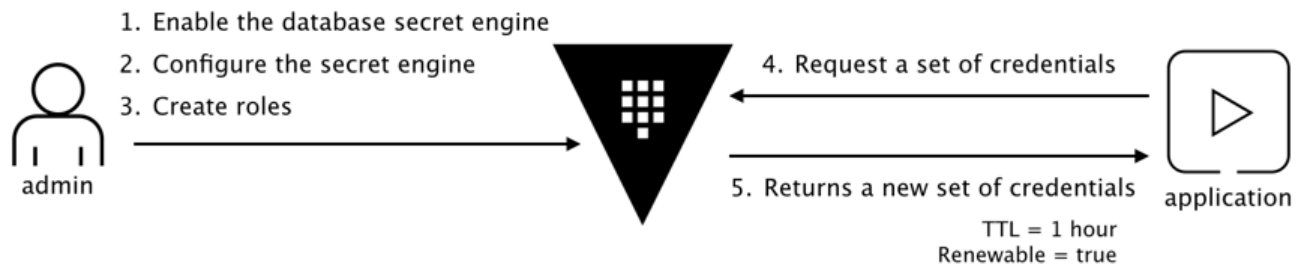
```
$ vault write -f database/rotate-role/education
```

Retrieving the static credentials for the `education` role will show a new password:

```
$ vault read database/static-creds/education
Key      Value
----      -
last_vault_rotation 2019-08-08T22:56:31.147886493Z
password      A1a-drknwwT4txj74u5V
rotation_period      24h
ttl            23h59m57s
username      vault-edu
```

### Task 3: Generate dynamic readonly credentials

The scenario is:



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

#### Step 4.3.1

Vault requires that you define the SQL to create credentials associated with this dynamic, readonly role. The SQL required to generate this role can be found in the file `readonly.sql`.

#### Execute

```
$ cat readonly.sql
```

The contents of the file should match:

```
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}}";
```

This is SQL that has been templated. 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 4.3.2

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.

#### Execute

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

This command creates a role named, `readonly` which has a default TTL of 1 hour, and max TTL is 24 hours. The credentials for the `readonly` role expires after 1 hour, but can be renewed multiple times within 24 hours of its creation.

#### Step 4.3.3

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 credentials to read from the database.

Generate a new set of credentials:

#### Execute

```
$ 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 4.3.3

Copy the value of `lease_id`. This lab will refer to it later as `<lease_id>`

Step 4.3.4

Check that the newly generated username exists by logging in as the postgres user and list all accounts.

Execute

```
$ psql -U postgres
```

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

```
postgres > \du

training@customer-training-shark:/workstation/vault > psql -U postgres
Welcome to PostgreSQL for HashiCorp training!
Type \q to exit.

postgres > \du

                                List of roles
-----+-----
Role name | Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replie
v-token-readonly-x271s0zv6x42wsqxrquq-1517942778 | Password valid until 2018-02-06 19:46:18-
postgres > \q
```

The username generated at Step 4.3.3 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 4.3.5

Enter `\q` to exit.

Step 4.3.6

Now, renew the lease for this credential the `<lease_id>`

Execute

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

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 grant the following:

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

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

Step 4.3.7

Renew and increment the TTL of the lease with the same `<lease_id>` :

Execute

```
$ vault lease renew -increment=2h <lease_id>
```

Expected output:

Key	Value
---	----
lease_id	database/creds/readonly/86a2109c-780c...
lease_duration	2h
lease_renewable	true

Task 4: Revoke leases

Under certain circumstances, privileged users may need to revoke the existing database credentials (e.g. database credentials are no longer in use or need to be disabled).

Step 4.4.1

Revoke the credentials associated with the `<lease_id>`:

Execute

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

Expected output:

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

Step 4.4.2

You can verify that the username no longer exists by logging in as postgres user and list all accounts as you did in **Step 4.3.4**.

```
training@customer-training-shark:/workstation/vault > psql -U postgres
Welcome to PostgreSQL for HashiCorp training!
Type \q to exit.

postgres > \du

                                List of roles
Role name |                               Attributes                               | Member of
-----+-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Step 4.4.3

Now its time to read a few more credentials from the postgres secret engine. Here, you will simulate a scenario where multiple applications have requested readonly database access.

Execute

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

The output should look similar to:

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

Execute

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

The output should look similar to:

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

Execute

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

The output should look similar to:

Key	Value
---	----
lease_id	database/creds/readonly/b422c54b-2664-e0b4-1b6e-74badbd7ab1c
lease_duration	1h
lease_renewable	true
password	A1a-0wsW97r2x6s49qv9
username	v-token-readonly-838uu0r2vvzyw0p34qw4-1517945648

Now, you have multiple sets of credentials.

Step 4.4.4

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 4.4.1**. But what if you don't know!?

Revoke all readonly credentials:

Execute

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

Expected output:

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

Revoke all database credentials:

Execute

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

## Challenge: Setup database secret engine via 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 & Tips:

- Remember the `-output-curl-string` flag
- [Database Secret Engine API doc](#)
- [PostgreSQL Database Secret Plugin HTTP API](#)



## Lab 4: Working with Database Secrets Engine - Challenge Solution

**Challenge: Setup Database Secret Engine with API - Sample Solution**

Generate the curl command to create database secret engine at `postgres-db`:

**Execute**

```
$ vault secrets enable -path=postgres-db -output-curl-string database
```

Create the database secret engine at `postgres-db`:

**Execute**

```
$ curl -X POST -H "X-Vault-Token: ${vault print token}" \
-d '{"type":"database","description":"","config":{"options":null,
"default_lease_ttl":"0s","max_lease_ttl":"0s","force_no_cache":false},
"local":false,"seal_wrap":false,"options":null}' \
$VAULT_ADDR/v1/sys/mounts/postgres-db
```

Generate the curl command to configure the secret engine at `postgres-db`:

**Execute**

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

Configure the secret engine at `postgres-db`:

**Execute**

```
$ curl -X PUT -H "X-Vault-Token: ${vault print token}" \
-d '{"allowed_roles":"readonly",
"connection_url":"postgresql://postgres@localhost/myapp",
"plugin_name":"postgresql-database-plugin"}' \
$VAULT_ADDR/v1/postgres-db/config/postgresql
```

Generate the curl command to create a role named `readonly` at path `postgres-db`:

**Execute**

```
$ vault write -output-curl-string postgres-db/roles/readonly db_name=postgresql \
creation_statements=@readonly.sql \
default_ttl=1h max_ttl=24h
```

Create a role named `readonly` at `postgres-db`:

**Execute**

```
$ curl -X PUT -H "X-Vault-Token: ${vault print token}" \
-d '{"creation_statements":
"CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD \"{{password}}\" VALID UNTIL \"{{expiration}}\";\nREVOKE ALL ON SCHEMA public FROM public, \"{{n
\"db_name\":\"postgresql\",\"default_ttl\":\"1h\",\"max_ttl\":\"24h\"}' \
$VAULT_ADDR/v1/postgres-db/roles/readonly
```

Generate the curl command to get a new set of credentials:

**Execute**

```
$ vault read -output-curl-string postgres-db/creds/readonly
```

Get a new set of credentials at `postgres-db` with the output formatted with `jq`.

**Execute**

```
$ curl -H "X-Vault-Token: ${vault print token}" \
$VAULT_ADDR/v1/postgres-db/creds/readonly | jq
```

**End of Lab 4**

## Lab 5: Encryption as a Service - Transit Secrets Engine

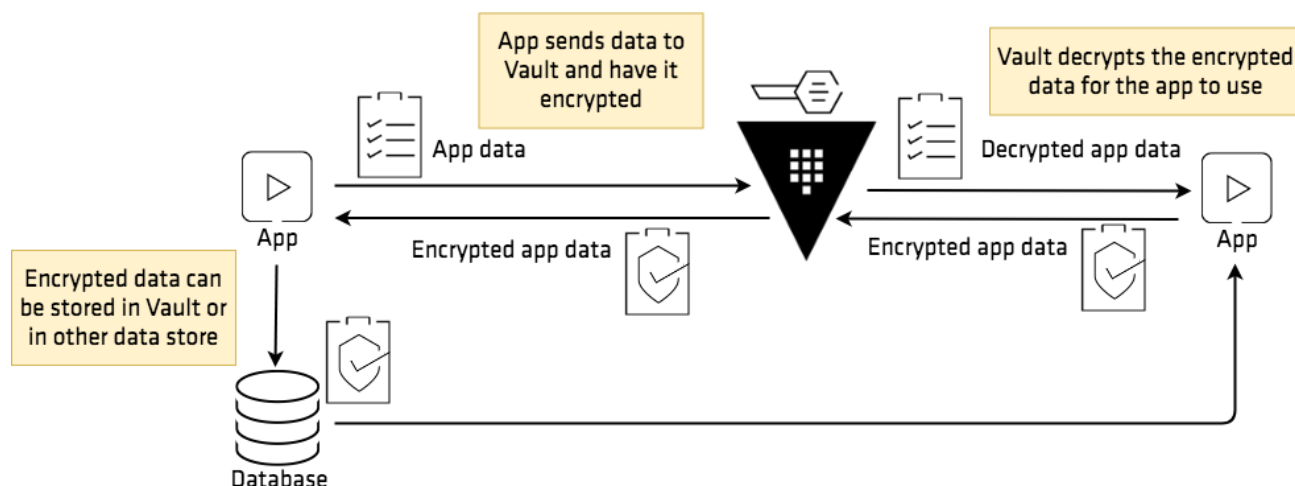
Duration: 25 minutes

In this lab you will employ the Vault CLI and web UI provide cryptographic services.

The `transit` secrets engine enables security teams to fortify data during transit and at rest. So even if an intrusion occurs, your data is encrypted with AES 256-bit CBC encryption (TLS in transit). Even if an attacker were able to access the raw data, they would only have encrypted bits. This means attackers would need to compromise multiple systems before exfiltrating data.

- Task 1: Configure transit secret engine
- Task 2: Encrypt secrets
- Task 3: Decrypt a cipher-text

- Task 4: Rotate the encryption Key
- Task 5: Update the key configuration
- Task 6: Encrypt data via web UI
- Challenge: Sign and validate data



## Task 1: Configure transit secret engine

The `transit` secrets engine must be configured before it can perform its operations. This step is usually done by an **operator** or a configuration management tool.

### Step 5.1.1

Enable the `transit` secret engine by executing the following command:

#### Execute

```
$ vault secrets enable transit
```

### Step 5.1.2

Now, create an encryption key ring named `cards`:

#### Execute

```
$ vault write -f transit/keys/cards
```

## Task 2: Encrypt secrets

Once the `transit` secrets engine has been configured, any client with a valid token with proper permission can send data to encrypt.

Here, you are going to encrypt a plaintext, `"credit-card-number"`.

Plaintext must be base64-encoded before it can be encrypted.

### Step 5.2.1

Create a secret from the base64 encoded text `"credit-card-number"` with the transit key at the path `transit/encrypt/cards`

#### Execute

```
$ vault write transit/encrypt/cards plaintext=$(base64 <<< "credit-card-number")
```

Output should look similar to:

Key	Value
ciphertext	vault:v1:cZNHVx+sxdMErXRSuDa1q/pz49fXTn1PScKfhf+PIZPvy8xKfkytpwKcbC0FF2U=

### Step 5.2.2

Vault does *NOT* store any of this data. The output you received is the ciphertext. You can store this ciphertext at the desired location (e.g. MySQL database) or pass it to another application.

Copy the value of `ciphertext`. This lab will refer to it later as `<ciphertext>`

## Task 3: Decrypt a ciphertext

Any client with a valid token with proper permission can decrypt the ciphertext generated by Vault. To decrypt the ciphertext, invoke the `transit/decrypt` endpoint.

### Step 5.3.1

Decrypt the `<ciphertext>`:



**Execute**

```
$ vault write transit/decrypt/cards \
  ciphertext="<ciphertext>"
```

Output should look similar to:

Key	Value
---	-----
plaintext	Y3JlZG10LWnhcmQtnVtYmVyCg==

**Step 5.3.2**

Decode the base-64 encoded text:

**Execute**

```
$ base64 --decode <<< "Y3JlZG10LWnhcmQtnVtYmVyCg=="
```

Expected output:

```
credit-card-number
```

**Task 4: Rotate the encryption Key**

One of the benefits of using the Vault `transit` secrets engine is its ability to easily rotate the encryption keys. Keys can be rotated manually by a human, or an automated process which invokes the key rotation API endpoint through cron, a CI pipeline, a periodic Nomad batch job, Kubernetes Job, etc.

Vault maintains the versioned keyring and the operator can decide the minimum version allowed for decryption operations. When data is encrypted using Vault, the resulting ciphertext is prepended with the version of the key used to encrypt it.

**Step 5.4.1**

Rotate the encryption keys at the path `transit/keys/cards/rotate`:

**Execute**

```
$ vault write -f transit/keys/cards/rotate
```

**Step 5.4.2**

Create a secret from the base64 encoded text `"visa-card-number"` with the transit key at the path `transit/encrypt/cards`:

**Execute**

```
$ vault write transit/encrypt/cards plaintext=$(base64 <<< "visa-card-number")
```

Output should look similar to:

Key	Value
---	-----
ciphertext	vault:v2:45f9zw6cglbrzCjI0yCyC6DBYtSBSxnMgUn9B5aHcGE...

**Step 5.4.3**

Compare the ciphertexts:

ciphertext	vault:v1:cZNVHx+xdMErXRSuDa1q/pz49fXTn1PScKfhf+PIZPvy...
------------	--

Notice that the first ciphertext starts with `"vault:v1:"`. After rotating the encryption key, the ciphertext starts with `"vault:v2:"`. This indicates that the data gets encrypted using the latest version of the key after the rotation.

**Step 5.4.4**

Rewrap the `<ciphertext>` with a write at the path `transit/rewrap/cards`:

**Execute**

```
$ vault write transit/rewrap/cards ciphertext="<ciphertext>"
```

Output should look similar to:

Key	Value
---	-----
ciphertext	vault:v2:kChHZ9w4ILRfw+Dz053IZ8m5PyB2yp2/tKbub34...

Notice that the resulting ciphertext now starts with `vault:v2:`.

This operation does not reveal the plaintext data. But Vault will decrypt the value using the appropriate key in the keyring and then encrypted the resulting plaintext with the newest key in the keyring.

## Task 5: Update the key configuration

The operators can update the encryption key configuration to specify the minimum version of ciphertext allowed to be decrypted, the minimum version of the key that can be used to encrypt the plaintext, the key is allowed to be deleted, etc.

This helps further tightening the data encryption rule.

### Step 5.5.1

Rotate the keys a **few times** to generate multiple versions:

#### Execute

```
$ vault write -f transit/keys/cards/rotate
```

### Step 5.5.2

Now, read the `cards` key information:

#### Execute

```
$ vault read transit/keys/cards
```

Output should look similar to:

Key	Value
keys	map[6:1531439714 1:1531439594 2:1531439667 3:1531439714 4:1531439714 5:1531439714]
latest_version	6
min_decryption_version	1
min_encryption_version	0

In the example, the current version of the key is **6**. However, there is no restriction about the minimum encryption key version, and any of the key versions can decrypt the data (`min_decryption_version`).

### Step 5.5.3

Enforce the use of the encryption key at version **5** or later to decrypt the data:

#### Execute

```
$ vault write transit/keys/cards/config min_decryption_version=5
```

### Step 5.5.4

Now, verify the `cards` key configuration:

#### Execute

```
$ vault read transit/keys/cards
```

Output should look similar to:

Key	Value
allow_plaintext_backup	false
deletion_allowed	false
derived	false
exportable	false
keys	map[5:1531811719 6:1531811721]
latest_version	6
min_decryption_version	5
min_encryption_version	0

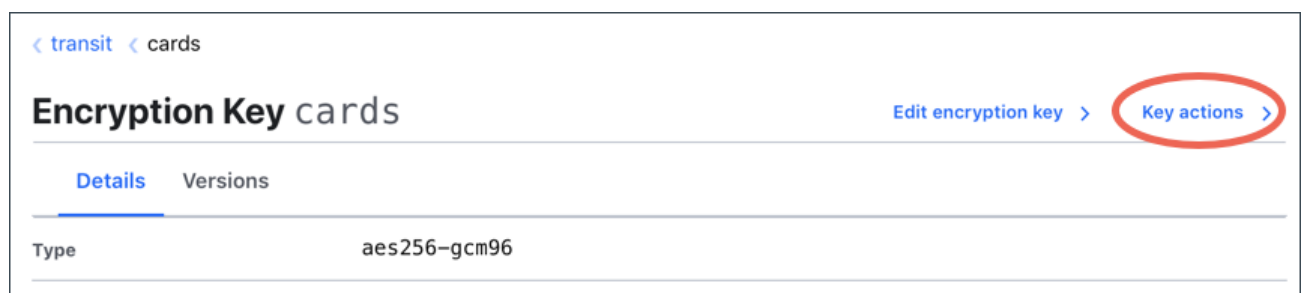
## Task 6: Encrypt data via web UI

Open a web browser and visit `http://<workstation_ip>:8200/ui/vault`.

In the **Token** field enter **root** and click **Sign in**.

### Step 5.6.1

Under **Secrets**, select **transit** > **cards**, and then select **Key actions**.



**Step 5.6.2**

With **Encrypt** selected, enter "my-master-card-number" in the **Plaintext** field.

The screenshot shows the Vault Transit web interface. On the left, under 'TRANSIT ACTIONS', 'Encrypt' is selected. The main area shows the 'cards' key with 'Key version' set to '6 (latest)'. The 'Plaintext' field contains the text 'my-master-card-number'. At the bottom right, the 'Encode to base64' button is highlighted with a red rectangular box. Below the plaintext field is an 'Encrypt' button.

Click **Encode to base64**.

**Step 5.6.3**

Click **Encrypt**.

**Step 5.6.4**

Click **Copy** to copy the ciphertext.

**Step 5.6.5**

Now, select **Decrypt**. (The ciphertext field should be already populated. If not, paste in the ciphertext you copied in **Step 5.6.4**.)

Finally, click **Decode from base64** to reveal the original text.

## Challenge: Sign and validate data

Consider a scenario where you want to ensure that the data came from a trusted source. You don't care who can read the data but you care about the *source* of the data. In such a case, you use data signing instead of encrypting.

During the lecture, it was mentioned that the `transit` secrets engine supports a number of key types and some support signing and signature verification. You can use Vault CLI or Web UI, and perform the following tasks:

1. Create a key named, `newsletter` which uses `rsa-4096` as its key type
2. Sign some data using the `newsletter` key
3. Verify the signature using the `newsletter` key

**Hint:**

- Create key: <https://www.vaultproject.io/api/secret/transit/index.html#create-key>
- Sign data: <https://www.vaultproject.io/api/secret/transit/index.html#sign-data>
- Verify signed data: <https://www.vaultproject.io/api/secret/transit/index.html#verify-signed-data>



## Lab 5: Encryption as a Service - Transit Secrets Engine

### Challenge: Sign and validate data - sample solution

CLI

Create a key

Execute

```
$ vault write transit/keys/newsletter type="rsa-4096"
```

Sign some test data

```
vault write transit/sign/newsletter \
  input=$(base64 <<< "HashiCorp Vault is awesome")
```

Output should look similar to:

Key	Value
---	-----
signature	vault:v1:NCKg6X0AMdLSt4G+R4k80GcaeVjSN5ZKmpXGFqxFYS8utV+aIahTv5vDCv26...

Execute

```
$ vault write transit/verify/newsletter \
  input=$(base64 <<< "HashiCorp Vault is awesome") \
  signature="vault:v1:NCKg6X0AMdLSt4G+R4k80GcaeVjSN5ZKmpXGFqxFYS8utV+aIahTv5vDCv26..."
```

Output should look similar to:

Key	Value
---	-----
valid	true

Web UI

1. Launch the Vault web UI: `http://<workstation_ip>:8200/ui`
2. Enter `root` in the **Token** field and then click **Sign In**
3. Select **transit** from the **Secrets Engine** list
4. Select **Create encryption key**
5. Enter **newsletter** in the **Name** field, select `rsa-4096` from the **Type** drop-down list
6. Click **Create encryption key**
7. Select **Key actions** and then **Sign**
8. Enter "HashiCorp Vault is awesome" in the **Plaintext** field, and then click **Encode to base64**
9. Click **Encrypt**
10. Click **Copy**
11. Now, select **Verify**. The **Input** and **Signature** fields should be pre-populated for you
12. Click **Verify**. If it's successful, "The input is valid for the given signature" message should display

End of Lab 5

## 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
- Challenge: Generate batch tokens

### 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:

Execute

```
$ 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

Create a token whose TTL is 30 seconds:

Execute

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

Output should look similar to:

Key	Value
---	----
token	s.5cR91QmyMfiOGQb8znnq3mDT
token_accessor	4T9AqnCAF8PH9Dm9NOKB9PN8
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.

#### Step 6.1.3

Copy the value of `token`. The remaining steps in this task refer to it as `<token_task_1>`.

#### Step 6.1.4

Lookup information about the `<token_task_1>`:

Execute

```
$ vault token lookup <token_task_1>
```

Key	Value
---	----
accessor	4T9AqnCAF8PH9Dm9NOKB9PN8
creation_time	1544643551
creation_ttl	30s

```

display_name    token
entity_id       n/a
expire_time     2018-12-12T19:39:41.802136869Z
explicit_max_ttl 0s
id              s.5cR9lQmyMfiOGQb8znnq3mDT
issue_time      2018-12-12T19:39:11.802135892Z
meta            <nil>
num_uses        0
orphan          false
path            auth/token/create
policies         [root]
renewable       true
ttl             22s
type            service

```

Notice that the token `type` is set to *service*. And this token has 22 seconds TTL left before it expires.

### Step 6.1.5

Wait until the TTL has expired and then lookup information about the `<token_task_1>`:

#### Execute

```
$ vault token lookup <token_task_1>
```

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:

#### Execute

```
$ 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 120 seconds:

#### Execute

```
$ vault token create -ttl=120 -policy="default"
```

Output should look similar to:

```

Key          Value
---          -
token        s.1SVfy0LBzGDzUK0kCKaMq6aY
token_accessor 86Rp1mgqUydD0Ns10NbYPgq4
token_duration 2m
token_renewable true
token_policies ["default"]
identity_policies []
policies       ["default"]

```

### Step 6.2.3

Copy the value of `token`. The remaining steps in this task refer to it as `<token_task_2>`.

### Step 6.2.4

Lookup information about the `<token_task_2>`:

#### Execute

```
$ vault token lookup <token_task_2>
```

Output should look similar to:

Key	Value
accessor	86Rp1mgqUydD0Ns10NbYPgq4
creation_time	1544643665
creation_ttl	2m
display_name	token
entity_id	n/a
expire_time	2018-12-12T19:43:23.7754948Z
explicit_max_ttl	0s
id	s.1SYfy0LBzGDzUK0kCKaMq6aY
issue_time	2018-12-12T19:41:05.531310836Z
last_renewal	2018-12-12T19:41:23.775495601Z
last_renewal_time	1544643683
meta	<nil>
num_uses	0
orphan	false
path	auth/token/create
policies	[default]
renewable	true
ttl	32s
type	service

### Step 6.2.5

Renew the token and double its TTL:

#### Execute

```
$ vault token renew -increment=240 <token_task_2>
```

Output should look similar to:

Key	Value
token	s.1SYfy0LBzGDzUK0kCKaMq6aY
token_accessor	86Rp1mgqUydD0Ns10NbYPgq4
token_duration	4m
token_renewable	true
token_policies	["default"]
identity_policies	[]
policies	["default"]

Now the token duration is extended to 4 minutes.

### Step 6.2.6

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

#### Execute

```
$ vault token lookup <token_task_2>
```

Output should look similar to:

Key	Value
accessor	WUQBraq0Hy8coew1B75ubypjI
creation_time	1552699378
creation_ttl	2m
display_name	token
entity_id	n/a
expire_time	2019-03-15T18:26:31.1699-07:00
explicit_max_ttl	0s
id	s.YYu0mojeAtd9ytpDC5n1RpvS
...	
ttl	2m42s
type	service

## Task 3: Create Tokens with Use Limit

### Step 6.3.1

Create a token with use limit of 2.

#### Execute

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

Output should look similar to:

Key	Value
token	s.1MXcFZsHMQdniRV4RS9kQfAW
token_accessor	20y827S8hQC3fRpQFHsZ6Ymu
token_duration	∞
token_renewable	false
token_policies	["root"]
identity_policies	[]
policies	["root"]

### Step 6.3.2

Copy the value of `<token>`. The remaining steps in this task refer to it as `<token_task_3>`.

### Step 6.3.3

Look up information about the token to consume 1 of the token's uses:

Example:

#### Execute

```
$ VAULT_TOKEN=<token_task_3> vault token lookup
```

Key	Value
---	-----
accessor	20y827S8hQC3fRpQFHsZ6Ymu
...	
num_uses	1
...	

Write a key/value to path `cubbyhole/test` to consume another of the token's uses:

#### Execute

```
$ VAULT_TOKEN=<token_task_3> vault write cubbyhole/test name="student01"
```

Success! Data written to: cubbyhole/test

Fail to read the key/value at path `cubbyhole/test` as the token is out of uses:

#### Execute

```
$ VAULT_TOKEN=<token_task_3> 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 an Orphan Token

### Step 6.4.1

Create a token with TTL of 90 seconds.

#### Execute

```
$ vault token create -ttl=90
```

Output should look similar to:

Key	Value
---	-----
token	d3f9538e-32d4-bcb1-c982-c335af532d66
token_accessor	cdf7ab42-9b7d-cec5-bae1-fafab6aa9593
token_duration	1m30s
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]

### Step 6.4.2

Copy the value of `<token>`. The remaining steps in this task refer to it as `<token_task_4>`.

### Step 6.4.3

Create a **child** token of the `<token_task_4>` with a longer TTL of 180 seconds:

#### Execute

```
$ VAULT_TOKEN=<token_task_4> vault token create -ttl=180
```

Output should look similar to:

Key	Value
---	-----
token	89e11854-8fd3-f86b-3862-34157ecf34c7
token_accessor	2671d179-a8a8-bf30-a300-d19aeb5ece50
token_duration	3m
token_renewable	true
token_policies	["root"]
identity_policies	[]
policies	["root"]



In this example, the token hierarchy is:

```
root
|__ d3f9538e-32d4-bcb1-c982-c335af532d66 (TTL = 90 seconds)
    |__ 89e11854-8fd3-f86b-3862-34157ecf34c7 (TTL = 180 seconds)
```

### Step 6.4.3

Copy the value of `token`. The remaining steps in this task refer to it as `<child_token>`.

### Step 6.4.3

After *90 seconds*, the `<token_task_4>` 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.

Wait 90 seconds and then lookup details about the `<child_token>`:

#### Execute

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

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

### Step 6.4.4

Create a new token with a TTL of 90 seconds.

#### Execute

```
$ vault token create -ttl=90
```

Copy the value of `token`. The remaining commands in this step will refer to it as `<token>`.

Next, create a **child** token with the `-orphan` flag.

#### Execute

```
$ VAULT_TOKEN=<token> vault token create -ttl=180 -orphan
```

Copy the value of `token`. The remaining commands in this step will refer to it as `<orphan_token>`.

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

#### Execute

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

Finally, verify that the `<orphan_token>`, with the expired parent `<token>`, still exists:

#### Execute

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

## Task 5: 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 6.5.1

Get help on `auth/token` path:

#### Execute

```
$ vault path-help auth/token
```

```
...
## PATHS
...
^roles/(?P<role_name>\w{1,64})$
This endpoint allows creating, reading, and deleting
roles.
...
```

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

### Step 6.5.2

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

#### Execute

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

Expected output:

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

### Step 6.5.3

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

#### Execute

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

Output should look similar to:

Key	Value
token	s.r5pALuXAxo1b01kle0h7m0R10
token_accessor	1gpzNPcBpcAHJ6sYy07rg9Ry
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.

## Task 6: Enable Username & Password Auth Method

Now, you are going to enable userpass auth method.

### Step 6.6.1

List the enabled authentication methods:

#### Execute

```
$ 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.

#### Execute

```
$ vault auth enable userpass
```

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

### Step 6.6.3

Create a user with the name `student01`, password `training` with the `default` policies.

#### Execute

```
$ vault write auth/userpass/users/student01 \
  password="training" policies="default"
```

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

```
Success! Data written to: auth/userpass/users/student01
```

### Step 6.6.4

Login with the user `student01` and their password `training`.

#### Execute

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

Output should look similar to:

```
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        s.1mhSiEOEQs21uW546ItFWm9m
token_accessor 6MliAbcbZgzX2lQTShBr3PSF
```

```
token_duration    768h
token_renewable   true
token_policies    ["default"]
identity_policies []
policies          ["default"]
token_meta_username student01
```

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.

### Step 6.6.5

Log back in with the root token.

#### Execute

```
$ vault login root
```

### Challenge: Generate batch tokens

Now, you should be familiar with `vault token` commands. Perform the following tasks.

**Task 1:** Create a token of type `batch` with `default` policy attached, and its TTL is set to 360 seconds.

**Task 2:** Enable another `userpass` auth method at `userpass-batch` path which generates a `batch` token upon a successful user authentication. Be sure to test and verify.

*Hint:* Run `vault auth enable -h` to see the available parameters.



## Lab 6: Authentication and Tokens - Challenge Solution

### Task 1: Solution

Create a `batch` token with default policy attached, and its TTL is set to 360 seconds.

#### Execute

```
$ vault token create -type=batch -policy=default -ttl=360
```

Key	Value
----	----
token	b.AAAAAQKbcL1gc7zZ57FcrH123AcHnprewUsV75CIck0PqLQ18nmXHv...
token_accessor	n/a
token_duration	6m
token_renewable	false
token_policies	["default"]
identity_policies	[]
policies	["default"]

## Task 2: Solution

Enable another `userpass` auth method at 'userpass-batch' which generates batch tokens.

### Execute

```
$ vault auth enable -path="userpass-batch" -token-type=batch userpass
```

Create a user called 'john' with the password 'training':

### Execute

```
$ vault write auth/userpass-batch/users/john \
  password="training" policies="default"
```

Authenticate as 'john' to verify its generate token type The token should starts with 'b.'

```
vault login -method=userpass -path="userpass-batch" \
  username="john" password="training"
```

Key	Value
----	----
token	b.AAAAAQIEQsY0RjktFBMD3U_8_w0_S0qCBreHJTW3tBwvXLxRk-in...
token_accessor	n/a
token_duration	768h
token_renewable	false
token_policies	["default"]
identity_policies	[]
policies	["default"]
token_meta_username	john

## End of Lab 6

# Lab 7: Direct Application Integration

Duration: 35 minutes

This lab demonstrates the use of Consul Template and Envconsul tools. To understand the difference between the two tools, you are going to retrieve the same information from Vault.

- Task 1: Run Vault Agent
- Task 2: Use Envconsul to populate DB credentials
- Task 3: Use Consul template to populate DB credentials
- Task 4: Use Vault Agent Templates

## Resources:

- Vault Agent: <https://www.vaultproject.io/docs/agent/>
- Consul Template: <https://github.com/hashicorp/consul-template>
- Envconsul: <https://github.com/hashicorp/envconsul>

## Task 1: Run Vault agent

Vault Agent runs on the **client** side to automate leases and tokens lifecycle management.

Since each student has only one workstation assigned, you are going to run the Vault Agent on the same machine as where the Vault server is running. The only difference between this lab and the real world scenario is that you set `VAULT_ADDR` to a remote Vault server address.

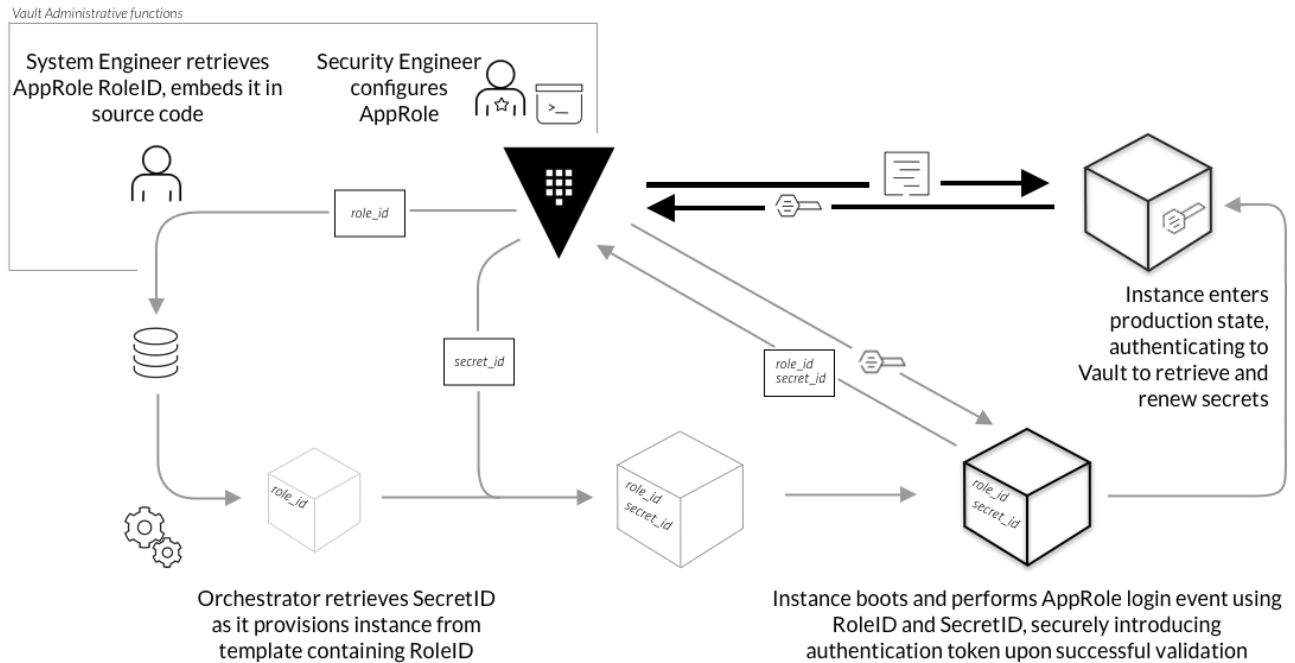
### Step 7.1.1

First, review the `/workstation/vault101/setup-approle.sh` script to examine what it performs.

### Execute

```
$ cat setup-approle.sh
```

This script creates a new policy called, `db_readonly`. (This assumes that you have completed Lab 4.) It enables `approle` auth method, generates a role ID and stores it in a file named, "roleID". Also, it generates a secret ID and stores it in the "secretID" file.



The `approle` auth method allows machines or apps to authenticate with Vault using Vault-defined roles. The **role ID** is equivalent to username, and the **secret ID** is equivalent to a password.

Refer to the *AppRole Pull Authentication* guide (<https://learn.hashicorp.com/vault/identity-access-management/iam-authentication>) as well as *AppRole with Terraform & Chef* guide (<https://learn.hashicorp.com/vault/identity-access-management/iam-approle-trusted-entities>) to learn more.

### Step 7.1.2

Execute the `setup-approle.sh` script.

Execute

```
$ ./setup-approle.sh
```

### Step 7.1.3

Examine the Vault Agent configuration file, `/workstation/vault101/agent-config.hcl`.

Execute

```
$ cat agent-config.hcl
```

```
pid_file = "./pidfile"

auto_auth {
  method "approle" {
    mount_path = "auth/approle"
    config = {
      role_id_file_path = "roleID"
      secret_id_file_path = "secretID"
      remove_secret_id_file_after_reading = false
    }
  }

  sink "file" {
    config = {
      path = "/workstation/vault101/approleToken"
    }
  }
}

cache {
  use_auto_auth_token = true
}

listener "tcp" {
  address = "127.0.0.1:8007"
  tls_disable = true
}

vault {
  address = "http://127.0.0.1:8200"
}
```

The `auto_auth` block points to the `approle` auth method which `setup-approle.sh` script configured. The acquired token gets stored in `/workstation/vault101/approleToken` (this is the sink location).

The `cache` block specifies the agent to listen port `8007`.

### Step 7.1.4

If you want to run Vault Agent against your neighbor's Vault server instead, edit the `vault` block so that it points to the correct Vault server address. Needless to say, your neighbor has to provide you the **roleID** and **secretID** to successfully authenticate.

Start the Vault Agent with `debug` logs.

Execute

```
$ vault agent -config=agent-config.hcl -log-level=debug
```

Expected Output:

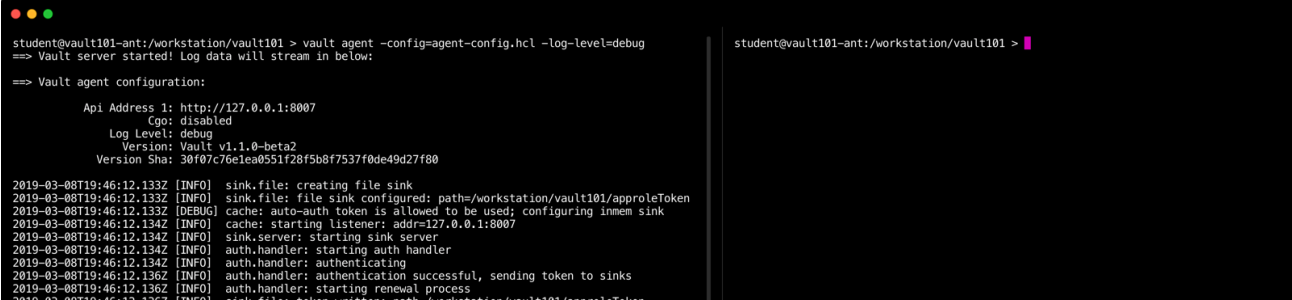
```
==> Vault server started! Log data will stream in below:

==> Vault agent configuration:

    Api Address 1: http://127.0.0.1:8007
      Cgo: disabled
    Log Level: debug
      Version: Vault v1.1.0
    Version Sha: 36aa8c8dd1936e10ebd7a4c1d412ae0e6f7900bd
    ...
```

Step 7.1.5

Open another terminal, and then SSH into your student workstation. Be sure to change the working directory to `/workstation/vault101`. Place the two terminals side-by-side if possible so that you can examine the logs as you execute commands.



Step 7.1.6

Vault Agent successfully authenticated with Vault using the `roleID` and `secretID`, and stored the acquired token in the `approleToken` file.

Execute

```
$ more approleToken
```

```
s.DL0ToAJKVjOSXXZdfzAKPWLY
```

Notice the following entries in the agent log in the first terminal:

```
[INFO] sink.file: creating file sink
[INFO] sink.file: file sink configured: path=/workstation/vault101/approleToken
[DEBUG] cache: auto-auth token is allowed to be used; configuring inmem sink
```

Step 7.1.7

Set the `VAULT_AGENT_ADDR` environment variable.

Execute

```
$ export VAULT_AGENT_ADDR="http://127.0.0.1:8007"
```

Step 7.1.8

Create a short-lived token and see how agent manages its lifecycle:

Execute

```
$ VAULT_TOKEN=$(cat approleToken) vault token create -ttl=30s -explicit-max-ttl=2m
```

Key	Value
---	----
token	s.qaP0odPTUdtbj5REak2ICuyg
token_accessor	Bov810fwIPlp48bENCuW8xv9
token_duration	30s
token_renewable	true
token_policies	["db_readonly" "default"]
identity_policies	[""]
policies	["db_readonly" "default"]

Examine the agent log:

```
...
[INFO] cache: received request: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: forwarding request: path=/v1/auth/token/create method=POST
[INFO] cache.apiproxy: forwarding request: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: processing auth response: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: setting parent context: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: storing response into the cache: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: initiating renewal: path=/v1/auth/token/create method=POST
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
```

The request was first sent to `VAULT_AGENT_ADDR` (agent proxy) and then forwarded to the Vault server (`VAULT_ADDR`). You should find an entry in the log indicating that the returned token was stored in the cache.

**Step 7.1.9**

The token's TTL was intentionally set to very short (30 seconds). Examine the agent log to see how it manages the token's lifecycle.

```
...
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
[DEBUG] cache.leasecache: secret renewed: path=/v1/auth/token/create
[DEBUG] cache.leasecache: renewal halted; evicting from cache: path=/v1/auth/token/create
[DEBUG] cache.leasecache: evicting index from cache: id=1f9d3e6d037d18f1e91b70be9918f95009433bf585252134de6a41a187e873ee path=/v1/auth/token/create method=POST
```

Vault Agent renews the token before its TTL was reached until the token reaches its maximum TTL (2 minutes). When the token renewal failed, the agent automatically evicts the token from the cache.

**Step 7.1.10**

Now, request database credentials for role, "readonly" which was configured in Lab 4.

**Execute**

```
$ VAULT_TOKEN=$(cat approleToken) vault read database/creds/readonly
```

Key	Value
lease_id	database/creds/readonly/2Tw9uVXkMB5oBw1DhtgzQZZb
lease_duration	1h
lease_renewable	true
password	A1a-5ZqdiR8AD5N46Mk6
username	v-approle-readonly-vFmdbjZ1HGxSkKPTzpa-1552079424

You should find the following entries in the agent log:

```
...
[INFO] cache: received request: path=/v1/database/creds/readonly method=GET
[DEBUG] cache.leasecache: forwarding request: path=/v1/database/creds/readonly method=GET
[INFO] cache.apiproxy: forwarding request: path=/v1/database/creds/readonly method=GET
[DEBUG] cache.leasecache: processing lease response: path=/v1/database/creds/readonly method=GET
[DEBUG] cache.leasecache: storing response into the cache: path=/v1/database/creds/readonly method=GET
...
```

**Step 7.1.11**

Request the database credentials for role, "readonly" again and examine its behavior:

**Execute**

```
$ VAULT_TOKEN=$(cat approleToken) vault read database/creds/readonly
```

Expected Output:

Key	Value
lease_id	database/creds/readonly/2Tw9uVXkMB5oBw1DhtgzQZZb
lease_duration	1h
lease_renewable	true
password	A1a-5ZqdiR8AD5N46Mk6
username	v-approle-readonly-vFmdbjZ1HGxSkKPTzpa-1552079424

Exactly the same set of database credentials are returned. The `lease_id` should be identical as well.

In the agent log, you find the following:

```
...
[INFO] cache: received request: path=/v1/database/creds/readonly method=GET
[DEBUG] cache.leasecache: returning cached response: path=/v1/database/creds/readonly
```

**Step 7.1.12**

Press **Ctrl + C** in the first terminal to stop the Vault Agent.



Do NOT forget to stop the Vault Agent before resume to the next task.

**Task 2: Use Envconsul to populate DB credentials**

Vault Agent can authenticate with Vault and acquire a client token. Now, use Envconsul to retrieve dynamic secrets from Vault and populate the username and password for your application to connect with database.

**Step 7.2.1**

View the `app.sh` file exists on the `/workstation/vault101` directory:

**Execute**

```
$ cat app.sh
```

```
#!/usr/bin/env bash

cat <<EOT
My connection info is:

username: "${DATABASE_CREDS_READONLY_USERNAME}"
password: "${DATABASE_CREDS_READONLY_PASSWORD}"
database: "my-app"
EOT
```

The main difference here is that the `app.sh` is reading the environment variables to retrieve `username` and `password`.

**Step 7.2.2**

Run the Envconsul tool using the Vault token acquired by the Vault Agent Auto-Auth.

**Execute**

```
$ VAULT_TOKEN=$(cat approleToken) envconsul -upcase -secret database/creds/readonly ./app.sh
```

```
My connection info is:

username: "v-token-readonly-wv1tq33s7z5uprpxxy68-1527631219"
password: "A1a-u54wut0v605qzw95"
database: "my-app"
```

If it returned `Error looking up token: Post http://127.0.0.1:8007/v1/auth/token/lookup: dial tcp 127.0.0.1:8007: connect: connection refused` error, unset the `VAULT_AGENT_ADDR` environment variable and then re-run the command.

**Execute**

```
$ unset VAULT_AGENT_ADDR
```

The output should display the `username` and `password` generated by Vault.

The `-upcase` flag tells Envconsul to convert all environment variable keys to uppercase. The default is lowercase (e.g. `database_creds_readonly_username`).

**Step 7.2.3**

Show the environment variables created by the Envconsul:

**Execute**

```
$ VAULT_TOKEN=$(cat approleToken) envconsul -upcase -secret database/creds/readonly \
  env | grep DATABASE
```

Output should look similar to:

```
DATABASE_CREDS_READONLY_PASSWORD=A1a-6808qrp9t64utw3t
DATABASE_CREDS_READONLY_USERNAME=v-token-readonly-31sq7t64pp2379r55s49-1527631800
```

If your application is designed to read values from environment variables, Envconsul provides the easiest app integration with Vault.

## Task 3: Use Consul template to populate DB credentials

In the **Secrets as a Service - Dynamic Secrets** lab, you enabled and configured a database secret engine. Assuming that you have an application that needs database credentials, use Consul Template to properly update the application file.

**Step 7.3.1**

Review the contents of the file `/workstation/vault101/config.yml.tpl`:

**Execute**

```
$ cat config.yml.tpl
```

The contents of the file should match:

```
---
{{- with secret "database/creds/readonly" -}}
username: "{{ .Data.username }}"
password: "{{ .Data.password }}"
database: "myapp"
{{- end -}}
```

It sends the `vault read` (HTTP GET) request to the `database/creds/readonly` endpoint. The `{{ .Data.username }}` expression renders the `username` value, and `{{ .Data.password }}` renders the `password` value from the returned data.

For more details, refer to the Consul Template README: <https://github.com/hashicorp/consul-template#secret>.

**Step 7.3.4**

Generate a version of this template with the values populated with the `consul-template` command:



### Execute

```
$ VAULT_TOKEN=$(cat approleToken) consul-template -template="config.yml.tpl:config.yml" -once
```

Defining the `VAULT_TOKEN` here sets an environment variable for the execution of this command. The `-template` flag defines the input template file `config.yml.tpl` followed by the output file `config.yml`. The `-once` flag tells Consul Template not to run this process as a daemon, and just run it once.

#### Step 7.3.5

Open the generated `config.yml` file to verify its content.

### Execute

```
$ cat config.yml
```

The contents of the file should like similar to:

```
---
username: "v-token-readonly-tu17xrtz345uz643980r-1527630039"
password: "A1a-7s0z9y223x2rp6v9"
database: "myapp"
```

The `username` and `password` were retrieved from Vault and populated the `config.yml` file.

## Task 4: Use Vault Agent Templates

Consul Template is the client directly interacting with Vault; therefore, you had to pass a client token to Consul Template so that it can interact with Vault. This requires you to operate two distinct tools (Vault Agent and Consul Template) to provide secrets to applications.

Vault Agent Templates allow Vault secrets to be rendered to files using the Consul Template markup language. This significantly simplifies the workflow when you are integrating your applications with Vault.

#### Step 7.4.1

Examine the Vault Agent configuration file, `/workstation/vault101/agent-templates-config.hcl`.

### Execute

```
$ cat agent-templates-config.hcl
```

```
pid_file = "./pidfile"

auto_auth {
  method "approle" {
    mount_path = "auth/approle"
    config = {
      role_id_file_path = "roleID"
      secret_id_file_path = "secretID"
      remove_secret_id_file_after_reading = false
    }
  }

  sink "file" {
    config = {
      path = "/workstation/vault101/approleToken"
    }
  }
}

cache {
  use_auto_auth_token = true
}

listener "tcp" {
  address = "127.0.0.1:8007"
  tls_disable = true
}

vault {
  address = "http://127.0.0.1:8200"
}

template {
  source      = "/workstation/vault101/config.yml.tpl"
  destination = "/workstation/vault101/config-agent.yml"
}
```

Notice the `template` block which points to the source template file (`/workstation/vault101/config.yml.tpl`) and rendered secrets to `/workstation/vault101/config-agent.yml`.

#### Step 7.4.2

Start the Vault Agent with `agent-templates-config.hcl`:

### Execute

```
$ vault agent -config-agent-templates-config.hcl -log-level=debug
```

Find the following in the agent log:

```
...
[DEBUG] (runner) checking template 3107094745b87a915dc20dbd69adef26
[DEBUG] (runner) was not watching 1 dependencies
[DEBUG] (watcher) adding vault.read(database/creds/readonly)
[DEBUG] (runner) diffing and updating dependencies
[DEBUG] (runner) watching 1 dependencies
[INFO] auth.handler: renewed auth token
```

```
[DEBUG] (runner) receiving dependency vault.read(database/creds/readonly)
[DEBUG] (runner) initiating run
[DEBUG] (runner) checking template 3107094745b87a915dc20dbd69adef26
[DEBUG] (runner) rendering "/workstation/vault101/config.yml.tpl" => "/workstation/vault101/config-agent.yml"
[INFO] (runner) rendered "/workstation/vault101/config.yml.tpl" => "/workstation/vault101/config-agent.yml"
...
```

Step 7.4.3

Verify that the secrets are rendered:

Execute

```
$ cat config-agent.yml
```

```
---
username: "v-approle-readonly-QexWd3VHxCKURvdyKh1-1576278894"
password: "A1a-mHMBEcmYBqVQ8ctg"
database: "myapp"
```

Vault Agent is a client daemon that solves the secret-zero problem by authenticating with Vault and manage the client tokens on behalf of the client applications. The Consul Template tool is widely adopted by the Vault users since it allowed applications to be "Vault-unaware".

End of Lab 7