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 4: Access Vault UI

Task 1: Connect to the Student 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 explicity 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: Permanenly added '<workstation_address>' (RSA) to the list of known hosts.
<username>@<workstation_address>'s password: <password>
```

Step 1.1.2

Run the following command to check the Vault server status:

Execute

```
$ vault status
```

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.1.0
Cluster Name    vault-cluster-875c9adb
Cluster ID      8917ca81-e460-49e5-b85d-db02a34d2720
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:

Execute

```
$ 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       6urXl1sr1zQJRUHD95jUzC4P
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:

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
```

This command reads a secret from a given 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/vault102`

Execute

```
$ cd /workstation/vault102
```

Step 1.3.2

Get help on the `audit enable` command:

Execute

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

Step 1.3.3

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

Execute the following command to enable audit logging:

Execute

```
$ vault audit enable file \
      file_path=/workstation/vault102/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:

Execute

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

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

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).

Optional

Often times, the logged information can help you understand what is going on with each command during the development. Invoke the following command to enable another audit device which generates a raw log entries:

Execute

```
$ vault audit enable -path=file-raw file \
      file_path=/workstation/vault102/audit-raw.log log_raw=true
```

To view the raw log:

Execute

```
$ sudo cat /workstation/vault102/audit-raw.log | jq
```

Task 4: Access Vault UI

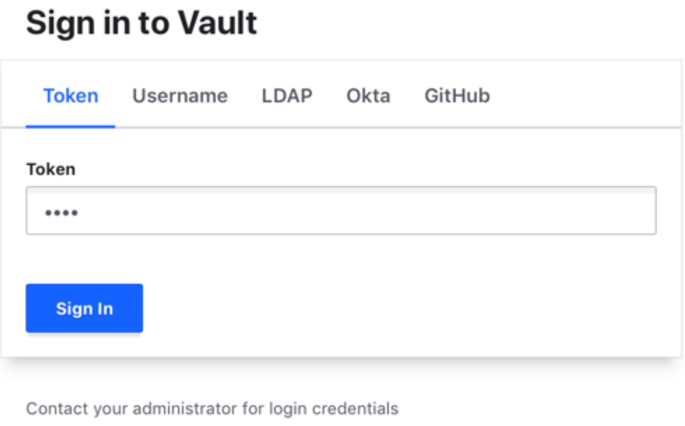
Vault UI is another useful client interface to interact with Vault.

Step 1.4.1

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

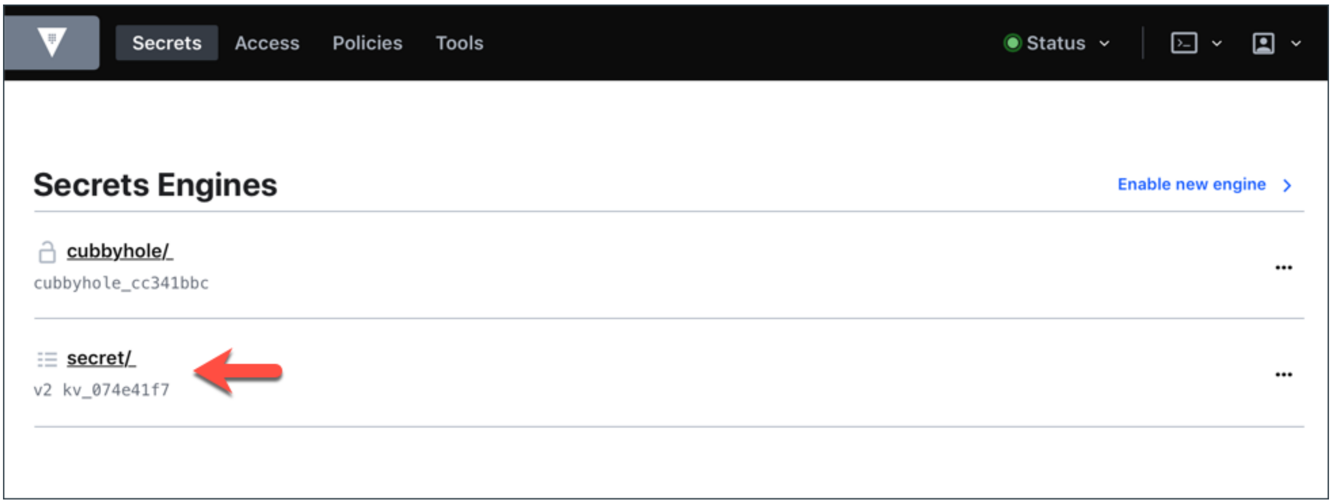
Step 1.4.2

Enter **root** in the **Token** field, and click **Sign in**.



Step 1.4.3

Notice that key/value v2 secrets engine is enabled at "secret/" path.



End of Lab 1

Lab 2: Vault Server Configuration

Duration: 10 minutes

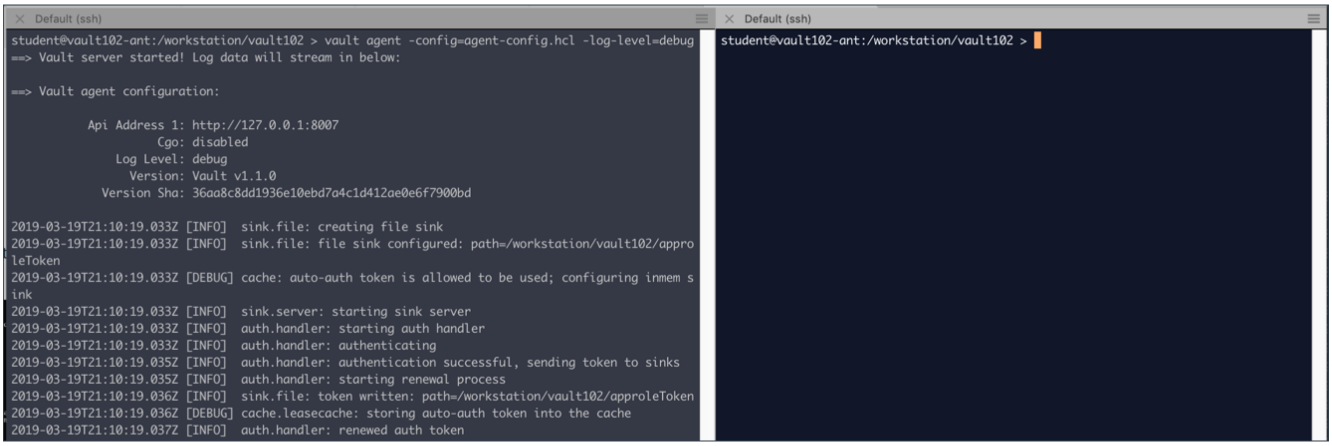
In this lab, you are going to perform the following tasks:

- Task 1: Configure and setup Vault
- Task 2: Connect to Vault UI
- Task 3: Explorer K/V Secrets Engine

Task 1: Configure and setup Vault

Step 2.1.1

You will need **two** terminals connected to your workstation for this lab. Repeat the step to start a **second** SSH session.



```
student@vault102-ant:/workstation/vault102 > vault agent -config=agent-config.hcl -log-level=debug
==> 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

2019-03-19T21:10:19.033Z [INFO]  sink.file: creating file sink
2019-03-19T21:10:19.033Z [INFO]  sink.file: file sink configured: path=/workstation/vault102/appro
leToken
2019-03-19T21:10:19.033Z [DEBUG] cache: auto-auth token is allowed to be used; configuring inmem s
ink
2019-03-19T21:10:19.033Z [INFO]  sink.server: starting sink server
2019-03-19T21:10:19.033Z [INFO]  auth.handler: starting auth handler
2019-03-19T21:10:19.033Z [INFO]  auth.handler: authenticating
2019-03-19T21:10:19.035Z [INFO]  auth.handler: authentication successful, sending token to sinks
2019-03-19T21:10:19.035Z [INFO]  auth.handler: starting renewal process
2019-03-19T21:10:19.036Z [INFO]  sink.file: token written: path=/workstation/vault102/approleToken
2019-03-19T21:10:19.036Z [DEBUG] cache.leasecache: storing auto-auth token into the cache
2019-03-19T21:10:19.037Z [INFO]  auth.handler: renewed auth token
```

```
student@vault102-ant:/workstation/vault102 >
```

Be sure to set your working directory to **/workstation/vault102**.

Step 2.1.2

Currently, a Vault server is running in development mode. Let's review the current Vault service configuration (`systemd`).

Execute

```
$ cat /etc/systemd/system/vault.service
```

```
...
[Service]
Environment=GOMAXPROCS=8
Environment=VAULT_DEV_ROOT_TOKEN_ID=root
Restart=on-failure
ExecStart=/usr/local/bin/vault server -dev -dev-listen-address=0.0.0.0:8200
...
```

Notice that the **ExecStart** starts Vault with "-dev" flag. The `-dev-listen-address` option overwrites the default address to bind the Vault server in "dev" mode. An environment variable `VAULT_DEV_ROOT_TOKEN_ID` is set to "root". This is also dev server specific environment variable.

Step 2.1.3

In this lab, you will learn how to start the Vault server in non-dev mode. Let's stop the Vault server which is currently running in "dev" mode.

Execute

```
$ sudo systemctl stop vault
```

```
[sudo] password for student:
```

> When prompted, enter your workstation password used to SSH into the student workstation.

Vault status check should fail.

Execute

```
$ vault status
```

```
Error checking seal status: Get http://127.0.0.1:8200/v1/sys/seal-status: dial tcp 127.0.0.1:8200: connect: connection refused
```

Step 2.1.3

Open the `/workstation/vault102/config.hcl` file to review its content.

Execute

```
$ cat config.hcl
```

```
disable_mlock = true
ui = true

storage "file" {
  path = "/workstation/vault102/data"
}

listener "tcp" {
  address     = "0.0.0.0:8200"
  tls_disable = 1
}
```

Notice that the storage backend is set to filesystem (`/workstation/vault102/data`).

Step 2.1.4

Get help on the `vault server` command:

Execute

```
$ vault server -h
```

```
...
Command Options:

  -config=<string>
      Path to a configuration file or directory of configuration files.
      This flag can be specified multiple times to load multiple
      configurations. If the path is a directory, all files which end
      in .hcl or .json are loaded.

  -log-level=<string>
      Log verbosity level. Supported values (in order of detail) are
      "trace", "debug", "info", "warn", and "err". The default is
      (not set). This can also be specified via the VAULT_LOG_LEVEL
      environment variable.
...
```

Locate where **Command Options** are listed. Use "-config" flag to specify the location of your server configuration file which is `/workstation/vault102/config.hcl` in this lab.

Step 2.1.5

Execute the following command to start the server:

Execute

```
$ vault server -config=config.hcl
```

```
==> Vault server configuration:

                     Cgo: disabled
              Listener 1: tcp (addr: "0.0.0.0:8200", cluster address: "0.0.0.0:8201", max_request_duration: "1m30s", max_request_size: "33554432", tls: "disabled")
               Log Level: info
                   Mlock: supported: true, enabled: false
                 Storage: file
                 Version: Vault v1.3.2
             Version Sha: 36aa8c8dd1936e10ebd7a4c1d412ae0e6f7900bd

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

2019-03-28T18:43:38.941Z [WARN]  no `api_addr` value specified in config or in VAULT_API_ADDR; falling back to detection if possible, but this value should be manually set
```

Step 2.1.6

In the **second terminal**, execute the `vault status` command to check the server status.

Execute

```
$ vault status
```

```
Key                Value
---                -----
Seal Type          shamir
Initialized        false
Sealed             true
```

```
Total Shares       0
Threshold          0
Unseal Progress    0/0
Unseal Nonce       n/a
Version            n/a
HA Enabled         false
```

Notice that the **Initialized** value is **false**, so the next step is to initialize the Vault server.

Step 2.1.7

Get help on the "`vault operator init`" command:

Execute

```
$ vault operator init -h
```

```
...
Common Options:

  -key-shares=<int>
      Number of key shares to split the generated master key into.
      This is the number of "unseal keys" to generate. This is
      aliased as "-n". The default is 5.

  -key-threshold=<int>
      Number of key shares required to reconstruct the master key.
      This must be less than or equal to -key-shares. This is aliased
      as "-t". The default is 3.
...
```

When you start the server for the first time, the server is not initialized which means that it does not have an encryption key, unseal keys nor initial root token to login.

The default number of key split is **five**, and requires **three** out of the five share keys to unseal Vault. You can tune this setting to meet your organization's standard.

Step 2.1.8

For the purpose of this exercise, run the following command to initialize Vault with number of key shares to be **one**, and the number of key threshold to be **one**, and store the output in a file named, `key.txt`.

Execute

```
$ vault operator init -key-shares=1 -key-threshold=1 > key.txt
```

Step 2.1.9

Open the `key.txt` to view its content:

Execute

```
$ cat key.txt
```

```
Unseal Key 1: bEQ+6vcC3tjWoTOuS4+e3gV+a0D3Cymqex6qwSWzylA=

Initial Root Token: 4e1WtW3bbmUTAKtv03tLPGmp

Vault initialized with 1 key shares and a key threshold of 1. Please
securely distribute the key shares printed above. When the Vault is
re-sealed, restarted, or stopped, you must supply at least 1 of these
keys to unseal it before it can start servicing requests.
...
```

Find your unseal key and the initial root token.

Step 2.1.10

Execute the following command to unseal your server:

Execute

```
$ vault operator unseal $(grep 'Key 1:' key.txt | awk '{print $NF}')
```

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.1.0
Cluster Name    vault-cluster-9a1dc60f
Cluster ID      4102f36b-76a7-f790-f4f5-20a48f172720
HA Enabled      false
```

Notice that the `Sealed` key is now set to **false**.

Step 2.1.11

Log into Vault with your initial root token which you stored in the `key.txt` file.

Execute

```
$ vault login $(grep 'Initial Root Token:' key.txt | awk '{print $NF}')
```

```
Key                  Value
---                  -----
token                s.NDsImkrtOFNV1ohfqsT4xEQe
token_accessor       Vurerxzg4RolamQca9KC8wP0
token_duration       ∞
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
```

Step 2.1.12

Press **Ctl + C** to stop the server that is running.

Step 2.1.13

Update the Vault service `systemd` to point to the `config.hcl`.

NOTE: The following instruction uses `vi` to edit the vault service file. Feel free to use other text editor of your choice (e.g. `nano`).

Execute

```
$ sudo vi /etc/systemd/system/vault.service
```

> Press **i** to insert text.

Edit the **ExecStart** (Line 11) to start the Vault server with configuration file, `/workstation/vault102/config.hcl`.

```
...
[Service]
Environment=GOMAXPROCS=8
Environment=VAULT_DEV_ROOT_TOKEN_ID=root
Restart=on-failure
ExecStart=/usr/local/bin/vault server -config=/workstation/vault102/config.hcl
...
```

Press `Esc` and then enter **:wq!** in `vi`.

Step 2.1.14

Execute the following command to start the Vault server service.

Reload vault service:

Execute

```
$ sudo systemctl daemon-reload
```

Start the vault service with new configuration:

Execute

```
$ sudo systemctl start vault
```

You need to unseal the vault server again:

Execute

```
$ vault operator unseal $(grep 'Key 1:' key.txt | awk '{print $NF}')
```

To view the service log:

Execute

```
$ sudo journalctl --no-pager -u vault
```

Or, you can pipe it into `less`:

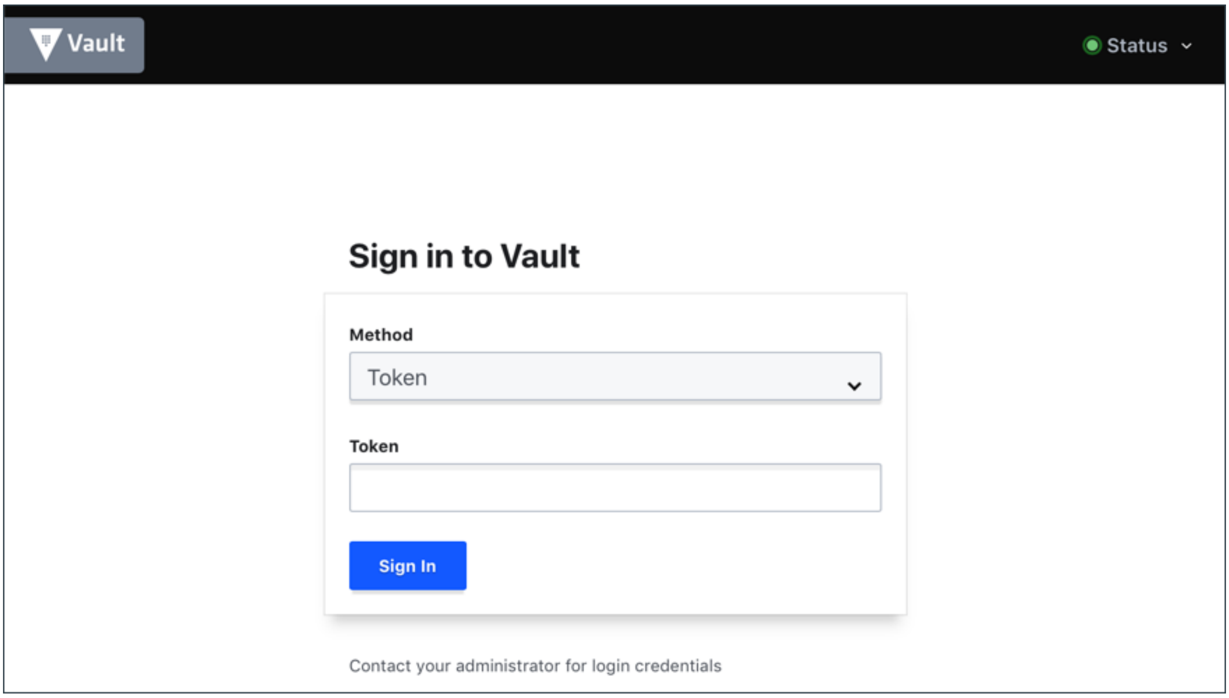
Execute

```
$ sudo journalctl --no-pager -u vault | less
```

Task 2: Connect to Vault UI

Step 2.2.1

Return to the Web UI and refresh to reload the page. If you don't have the Web UI already running, open a web browser, and enter the Web UI address: `http://<workstation_public_IP_address>:8200/ui`



Enter the initial root token in the **Token** text field, and click **Sign In**.

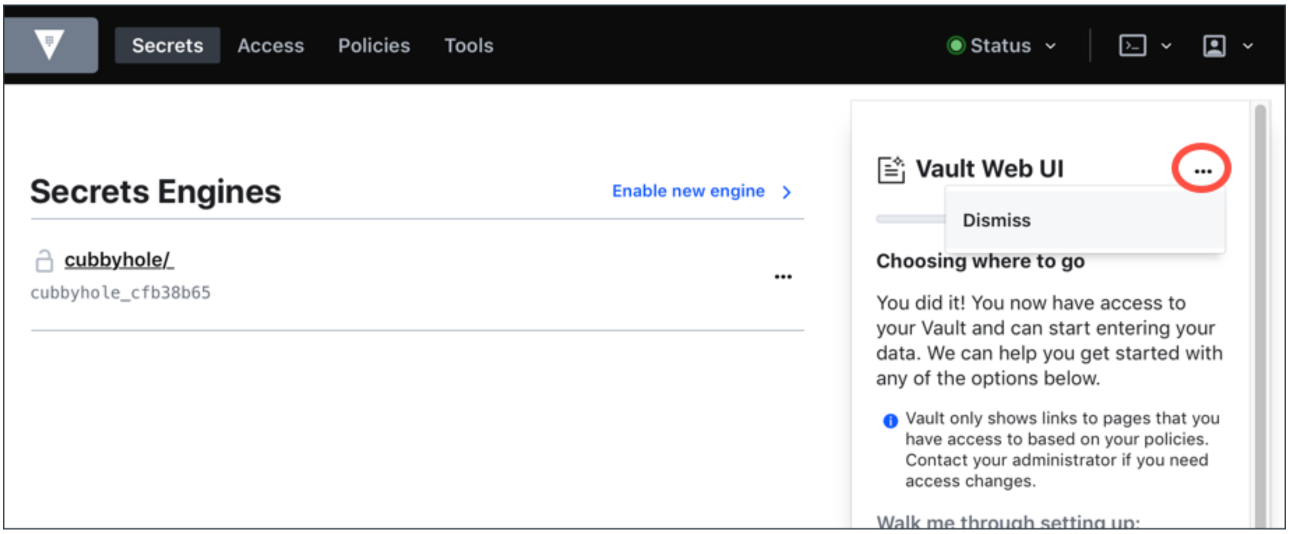
Remember that initial root token is stored in the `key.txt` file:

Execute

```
$ echo $(grep 'Initial Root Token:' key.txt | awk '{print $NF}')
```

Step 2.2.2

For now, click **Dismiss** to close out the guide.



Secrets Engines

cubbyhole/
cubbyhole_cfb38b65

Vault Web UI

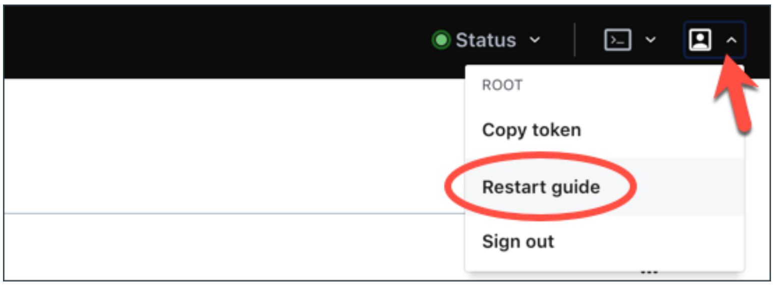
Choosing where to go

You did it! You now have access to your Vault and can start entering your data. We can help you get started with any of the options below.

Vault only shows links to pages that you have access to based on your policies. Contact your administrator if you need access changes.

Walk me through setting up:

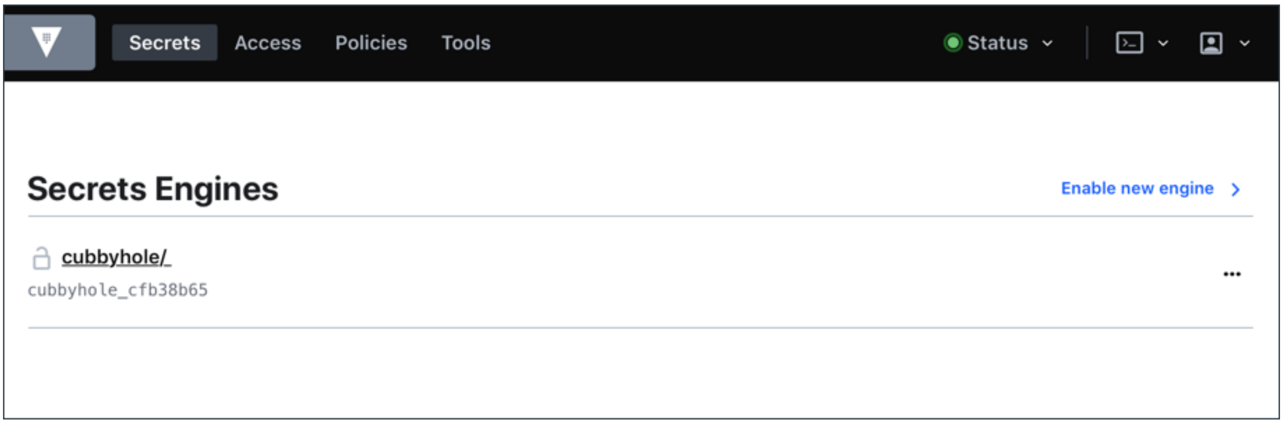
You can relaunch the guide later to explorer different features.



The built-in guide helps you complete some of the common tasks to setup your Vault environment.

Step 2.2.3

Notice that the only secrets engine enabled is `cubbyhole`.



Secrets Engines

cubbyhole/
cubbyhole_cfb38b65

Only when you start the Vault server in development mode, you would see the key/value v2 secrets engine enabled at "secret/" path by default. However, when you run Vault in non-development mode, you would need to enable key/value secrets engine explicitly like other secrets engine.

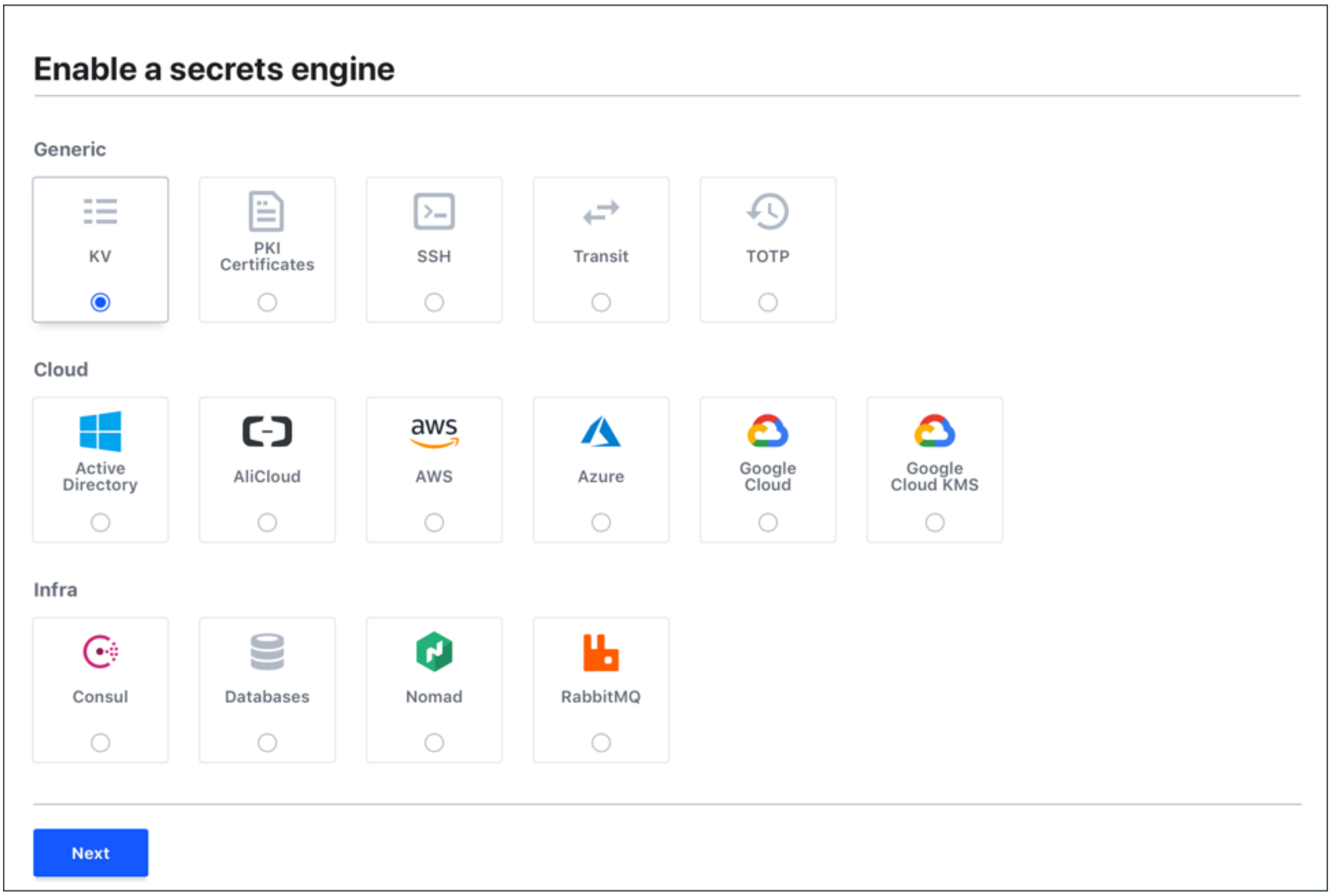
Task 3: Explorer K/V Secrets Engine

In Module 1, it was discussed that the **storage backend** is responsible for persisting data. All data flows between Vault and the storage backend passes through the **barrier** (cryptographic seal); therefore, data is encrypted in-transit as well as at-rest.

Let's explore the behavior. Remember that the storage backend is a filesystem in this lab (`/workstation/vault102/data`). First, create some test data in Vault, and then examine the data stored in the storage backend.

Step 2.3.1

In UI, select **Enable new engine**, and then check **KV** radio button.



Enable a secrets engine

Generic

Cloud

Infra

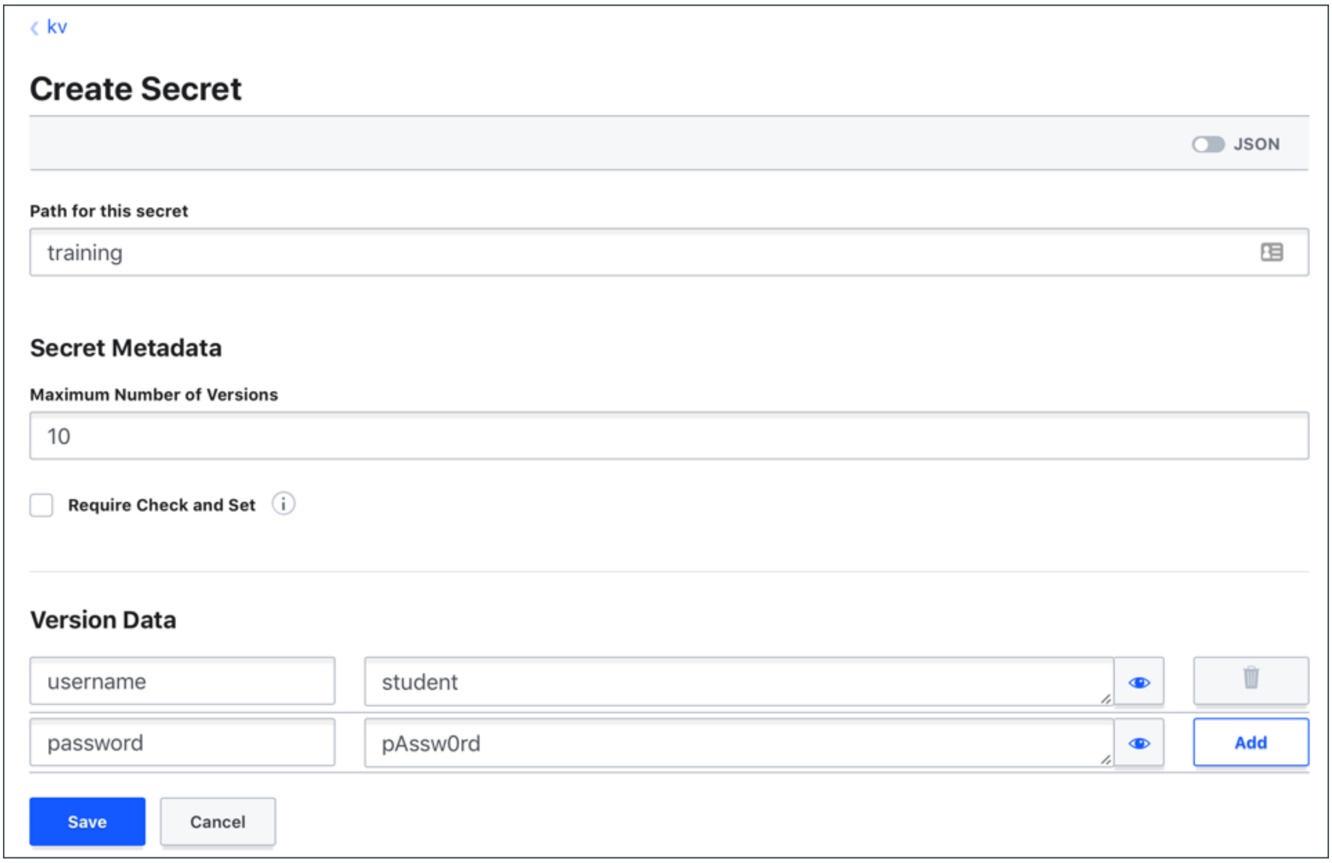
Click **Next**.

Step 2.3.2

Accept the defaulted values and click **Enable Engine**.

Step 2.3.3

Select **Create secret**, and enter **training** in the **Path for this secret** field. Create some key-value pairs of data to store.



Create Secret

Path for this secret: training

Secret Metadata

Maximum Number of Versions: 10

□ Require Check and Set

Version Data

username	student
password	pAssw0rd

Click **Save**.

Step 2.3.4

Now, return to the terminal window. Explore the **/workstation/vault102/data/logical** directory.

If you don't remember, print out your current username:

Execute

```
$ whoami
```

Execute the `chown` command to change the directory ownership.

Execute

```
$ sudo chown -R "<username>:<username>" "/workstation/vault102"
```

Example: If your username is `student`, the command would look:

Execute

```
$ sudo chown -R "student:student" "/workstation/vault102"
```

Now, let's explore the storage backend.

```
cd /workstation/vault102/data/logical
```

> Hope it is obvious, but the `chown` command is executed only so that you can examine how the secrets are stored in the storage backend. In a real scenario, your Vault server is backed by a scalable system such as Consul, and depends on your role, you may or may not have an access to view.

Step 2.3.5

Vault generates a unique ID for each path. At this point, you find only one directory since you enabled key/value v2 secrets engine at **kv/** path and nothing else.

Example:

Execute

```
$ tree
```

```
.
└── a96595ed-65d7-2142-a346-759f825085d9
    └── efef524e-5f6a-3590-ee8a-45fab3521249
        ├── archive
        │   └── _metadata
        ├── metadata
        │   └── _p0E3LngPpxsNbGon7hdGzzS72W0Td77NfAppZGNChJYVLrVP...
        ├── policy
        │   └── _metadata
        ├── _salt
        ├── _upgrading
        └── versions
            └── 145
                └── _173bce8277e4d202f419951691c81feed23d1a3ca2862...
```

Change directory into kv/training --> a96595ed.../efef524e-5f6...

```
$ cd a96595ed-65d7-2142-a346-759f825085d9/efef524e-5f6a-3590-ee8a-45fab3521249
```

Currently, there is only one version of the secret at `/kv/training`.

Look into the versioned k/v data.

```
$ cd versions/145/
```

Open the secret:

```
$ cat _173bce8277e4d202f419951691c81feed23d1a3ca2806297f6570baab3e21

{"Value":"AAAAAQLU6bcOO9VRMmZB3JQ52VNkA98+enN/CswGSwCvHhyE7jX9RPzQsSNc++mtIwQqOVHlrCgTPRKZLpw5ra4OyWQZC1iAeiZ72YZrS7SfaBCMmv5CQbgjD4UF"}
```

> The data was encrypted by Vault before it was passed to the storage backend to persist; therefore, the storage backend will never see the plaintext of the secret.

Step 2.3.6

Change the working directory back to `/workstation/vault102`.

Execute

```
$ cd /workstation/vault102
```

Step 2.3.7

Execute the following command to enable audit logging:

Execute

```
$ vault audit enable file file_path=/workstation/vault102/audit.log
```

NOTE: In Lab 1, you enabled audit against the dev server that was running. Now, enable an audit device against the non-dev server you started.

End of Lab 2

Lab 3: Auto-Unseal

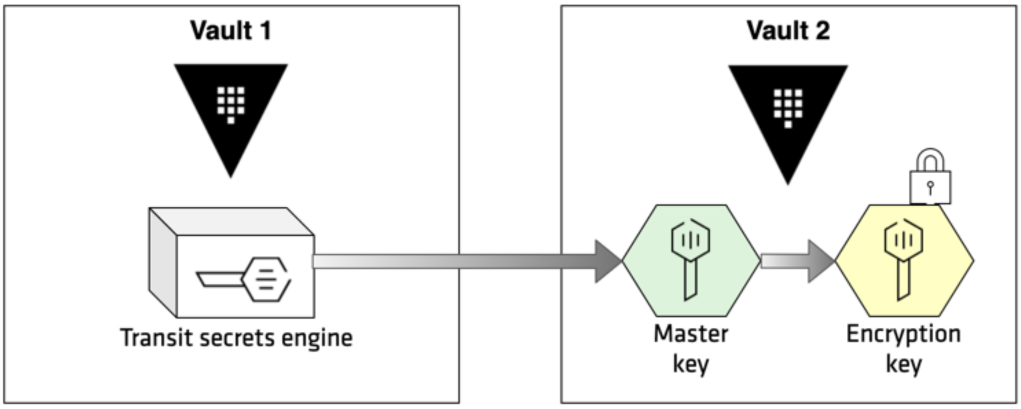
Duration: 10 minutes

In this lab, you are going to perform the following tasks:

- Task 1: Configure Auto-unseal Key Provider
- Task 2: Configure Auto-unseal
- Task 3: Audit the incoming request

Introduction

To enable **Transit Auto-Unseal**, you would need **two** Vault servers. Two options are provided to perform this lab. **Choose one** of the following options and perform the tasks as instructed.



Option 1: Work with your neighbor

It's time to make a friend! Turn to your **neighbor** and decide which one of you will provide the encryption service.

If your Vault server will provide the encryption service (**Vault 1**), perform ***Task 1*** and ***Task 3***.

If you are enabling auto-unseal on your Vault server (**Vault 2**), perform ***Task 2***. Those tasks must be completed in sequence, so work together and observe.

Option 2: Work on your own

Depending on the classroom environment (e.g. online virtual course), it may be difficult to pair up with someone.

You will run 2 vault server instances (**Vault 1** and **Vault 2**) on your student workstation. Proceed to perform ***Task 4***, ***Task 5***, and ***Task 6***.

Option 1: Work with your neighbor

Task 1: Configure Auto-unseal Key Provider (Vault 1)

Step 3.1.1

Enable the `transit` secrets engine and create a key.

Enable the transit secrets engine:

Execute

```
$ vault secrets enable transit
```

Create a key named `autounseal` :

Execute

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

Step 3.1.2

Create a `autounseal` policy defined by `/workstation/vault102/autounseal.hcl` policy file.

Explorer the policy definition:

Execute

```
$ cat autounseal.hcl
```

Create autounseal policy.

Execute

```
$ vault policy write autounseal autounseal.hcl
```

Step 3.1.3

Create a new token with `autounseal` policy.

Execute

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

```
Key                  Value
---                  -----
token                s.iuYhAza1g0kIDbWsooq4npLA
token_accessor       5MyorQzN93hT8zXDZE5mh6kI
token_duration       768h
token_renewable      true
token_policies       ["autounseal" "default"]
identity_policies    []
policies             ["autounseal" "default"]
```

Now, provide your workstation's **IP address** and the client **token** (`s.iuYhAza1g0kIDbWsooq4npLA` in this example) to your neighbor.

Task 2: Configure Auto-unseal (Vault 2)

Step 3.2.1

You are going to add `seal` stanza in your `config.hcl` to enable auto-unseal. Stop the Vault server currently running.

Execute

```
$ sudo systemctl stop vault
```

Step 3.2.2

Now, modify the server configuration: `/workstation/vault102/config-autounseal.hcl`

Execute

```
$ vi config-autounseal.hcl
```

Inside the **seal** block, you need to set the **address** value to your neighbor's Vault server.

Press **i** to insert text.

Example:

If your neighbor's workstation IP address is `192.0.2.3` and the client token from ***Step 3.1.3*** is `s.AawRF0rtUygIAlz3nn1NGgnB`, your `config-autounseal.hcl` would look like:

```
disable_mlock = true
ui=true

...

seal "transit" {
  address = "http://192.0.2.3:8200"
  token = "s.AawRF0rtUygIAlz3nn1NGgnB"
  disable_renewal = "false"
  key_name = "autounseal"
  mount_path = "transit/"
  tls_skip_verify = "true"
}
```

Press `Esc` and then enter **:wq!** in `vi` to save and exit.

Since each student has only one workstation assigned, the Vault client **token** value was set in the `seal` block; however, it is recommended to set the token value as `VAULT_TOKEN` environment variable.

Step 3.2.3

Start the vault server with configuration file.

Execute

```
$ vault server -config=config-autounseal.hcl
```

Step 3.2.4

In the second terminal, execute the migration command.

You will need the unseal key.

Execute

```
$ echo $(grep 'Key 1:' key.txt | awk '{print $NF}')
```

Execute the following command, and you will be prompted to enter the unseal key:

Execute

```
$ vault operator unseal -migrate
```

```
Unseal Key (will be hidden):

Key                      Value
---                      -----
Recovery Seal Type       shamir
Initialized              true
Sealed                   false
...
```

Step 3.2.5

Press **Ctl + C** to stop the server, and then update the Vault service `systemd` to point to the `config-autounseal.hcl` file.

Execute

```
$ sudo vi /etc/systemd/system/vault.service
```

Press **i** to insert text.

Edit the **ExecStart** (Line 11) to start the Vault server with configuration file, `/workstation/vault102/config-autounseal.hcl`.

```
...
[Service]
...
ExecStart=/usr/local/bin/vault server -config=/workstation/vault102/config-autounseal.hcl
...
```

Press `Esc` and then enter **:wq!** in `vi`.

Step 3.2.6

Execute the following command to start the Vault server service.

Reload the vault service:

Execute

```
$ sudo systemctl daemon-reload
```

Start the vault service with new configuration.

Execute

```
$ sudo systemctl start vault
```

Verify the Vault status.

Execute

```
$ vault status
```

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
...
```

Notice that the Vault server is already unsealed.

Task 3: Audit the incoming request (Vault 1)

Step 3.3.1

You have enabled audit device at the end of Lab 2. Tail the audit log.

Execute

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

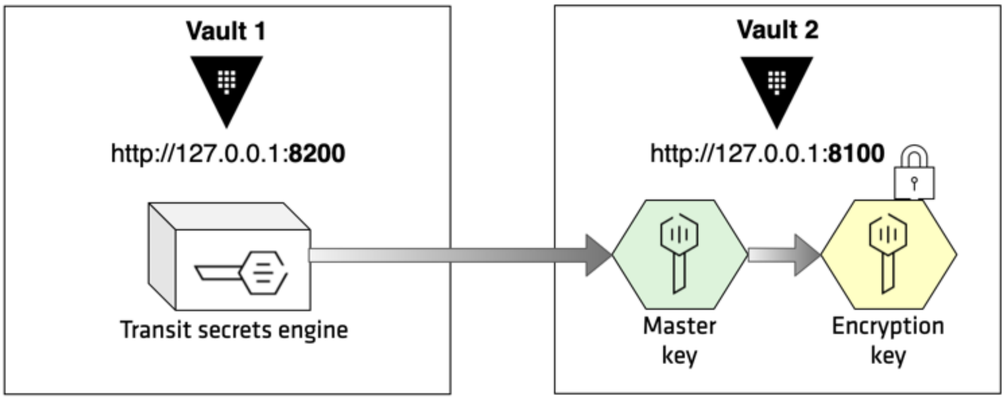
```
...
"request": {
   ...
   "path": "transit/encrypt/autounseal",
   "data": {
     "plaintext": "hmac-sha256:886e8dd6474c099b967d2bfc..."
   },
   "policy_override": false,
   "remote_address": "198.51.100.44",
   "wrap_ttl": 0,
   "headers": {}
 },
 ...
"request": {
   ...
   "path": "transit/decrypt/autounseal",
   "data": {
     "ciphertext": "hmac-sha256:15cbe1fd0bb2745a755d0e9..."
   },
   ...
 },
```

When Vault 2 was successfully auto-unsealed, the Vault 1 audit log should indicate that a request came from a remote address (`198.51.100.44` in this example) against "transit/encrypt/autounseal" endpoint. When Vault 2 was stopped (Step 3.2.5) and restarted (Step 3.2.6), a request was made against the "transit/decrypt/autounseal" endpoint so that Vault 2 can be automatically unsealed.

Step 3.3.2

Press **Ctrl + C** to stop tailing the audit log.

Option 2: Work on your own



Task 4: Configure Auto-unseal Key Provider (Vault 1)

Step 3.4.1

Enable the `transit` secrets engine and create a key.

Enable the transit secrets engine.

Execute

```
$ vault secrets enable transit
```

Create a key named `autounseal` :

Execute

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

Step 3.4.2

Create a `autounseal` policy defined by `/workstation/vault102/autounseal.hcl` policy file.

Explorer the provided policy file:

Execute

```
$ cat autounseal.hcl
```

Create `autounseal` policy.

Execute

```
$ vault policy write autounseal autounseal.hcl
```

Step 3.4.3

Create a new token with autounseal policy.

Execute

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

```
Key                  Value
---                  -----
token                s.iuYhAza1g0kIDbWsooq4npLA
token_accessor       5MyorQzN93hT8zXDZE5mh6kI
token_duration       768h
token_renewable      true
token_policies       ["autounseal" "default"]
identity_policies    []
policies             ["autounseal" "default"]
```

Copy the client **token** (`s.iuYhAza1g0kIDbWsooq4npLA` in this example).

Task 5: Configure Auto-unseal (Vault 2)

Step 3.5.1

Modify the configuration file for Vault 2: `/workstation/vault102/config-autounseal-2.hcl`

Execute

```
$ vi config-autounseal-2.hcl
```

Notice that the storage backend is set to "`/workstation/vault102/data-2`", and the Vault 2 will be listening to port **8100**:

```
...
storage "file" {
  path = "/workstation/vault102/data-2"
}

listener "tcp" {
  address     = "0.0.0.0:8100"
  tls_disable = 1
}
...
```

Inside the **seal** stanza, set the `token` value which you acquired in ***Step 3.4.3***

Press **i** to insert text.

Example:

If the client token from ***Step 3.4.3*** was `s.AawRF0rtUygIAlz3nn1NGgnB`, your `config-autounseal-2.hcl` would look like:

```
disable_mlock = true
ui=true
...

seal "transit" {
  address = "http://127.0.0.1:8200"
  token = "s.AawRF0rtUygIAlz3nn1NGgnB"
  disable_renewal = "false"
  key_name = "autounseal"
  mount_path = "transit/"
  tls_skip_verify = "true"
}
```

Press `Esc` and then enter **:wq!** in `vi` to save and exit.

Since each student has only one workstation assigned, the Vault client **token** value was set in the `seal` block; however, it is recommended to set the token value as `VAULT_TOKEN` environment variable.

Step 3.5.2

Start the vault server with configuration file.

Execute

```
$ vault server -config=config-autounseal-2.hcl
```

Step 3.5.3

In the second terminal, initialize your second Vault server (**Vault 2**).

Execute

```
$ VAULT_ADDR=http://127.0.0.1:8100 vault operator init -recovery-shares=1 \
        -recovery-threshold=1 > recovery-key.txt
```

By passing the `VAULT_ADDR`, the subsequent command gets executed against the second Vault server (http://127.0.0.1:8100). Notice that you are setting the number of **recovery** key and **recovery** threshold because there is no unseal keys with auto-unseal. Vault 2's master key is now protected by the `transit` secret engine of **Vault 1**.

In the terminal where the server is running, you should see entries similar to:

```
...
[INFO]  core: security barrier not initialized
[INFO]  core: security barrier initialized: shares=1 threshold=1
[INFO]  core: post-unseal setup starting
...
[INFO]  core: vault is unsealed
[INFO]  core.cluster-listener: starting listener: listener_address=0.0.0.0:8101
...
```

Step 3.5.4

Check the Vault 2 status.

Execute

```
$ VAULT_ADDR=http://127.0.0.1:8100 vault status
```

```
Key                      Value
---                      -----
Recovery Seal Type       shamir
Initialized              true
Sealed                   false
Total Recovery Shares    1
Threshold                1
...
```

Step 3.5.5

Press **Ctrl + C** to stop the Vault 2 server that is running.

```
...
[INFO]  core.cluster-listener: rpc listeners successfully shut down
[INFO]  core: cluster listeners successfully shut down
[INFO]  core: vault is sealed
```

Note that Vault 2 is now ***sealed***.

Step 3.5.6

Press the upper-arrow key, and execute the `vault server` command again to start Vault 2 and see what happens.

Execute

```
$ vault server -config=config-autounseal-2.hcl
```

```
==> Vault server configuration:

               Seal Type: transit
         Transit Address: http://127.0.0.1:8200
        Transit Key Name: autounseal
      Transit Mount Path: transit/
                     Cgo: disabled
              Listener 1: tcp (addr: "0.0.0.0:8100", cluster address: "0.0.0.0:8101", max_request_duration: "1m30s", max_request_size: "33554432", tls: "disabled")
               Log Level: info
                   Mlock: supported: true, enabled: false
                 Storage: file
                 Version: Vault v1.3.2
             Version Sha: 36aa8c8dd1936e10ebd7a4c1d412ae0e6f7900bd

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

[WARN]  no `api_addr` value specified in config or in VAULT_API_ADDR; falling back to detection if possible, but this value should be manually set
[INFO]  core: stored unseal keys supported, attempting fetch
[INFO]  core: vault is unsealed
```

Notice that the Vault server is already unsealed. The **Transit Address** is set to your Vault 1 which is listening to port 8200 (http://127.0.0.1:8200).

Task 6: Audit the incoming request (Vault 1)

Step 3.6.1

You have enabled audit device at the end of Lab 2. Tail the audit log.

Execute

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

```
...
"request": {
    ...
    "path": "transit/encrypt/autounseal",
    "data": {
      "plaintext": "hmac-sha256:31302463ac9978b0413fd89e2a9d4aafc5055..."
    },
    "policy_override": false,
    "remote_address": "127.0.0.1",
    "wrap_ttl": 0,
    "headers": {}
  },
  ...

"request": {
    ...
    "path": "transit/decrypt/autounseal",
    "data": {
      "ciphertext": "hmac-sha256:f03c0d85b132591693b6032fff0ee2b1e5e2cc92..."
    },
    "policy_override": false,
    "remote_address": "127.0.0.1",
    "wrap_ttl": 0,
    "headers": {}
  },
 ...
```

When Vault 2 was successfully auto-unsealed, the Vault 1 audit log should indicate that a request came from `127.0.0.1` against "transit/encrypt/autounseal" during the initialization of Vault 2. When Vault 2 was stopped and restarted, a request was made against the "transit/decrypt/autounseal" endpoint so that Vault 2 can be automatically unsealed.

Step 3.6.2

Press **Ctrl + C** to stop the **Vault 2** instance (`http://127.0.0.1:8100`).

End of Lab 3

Lab 5: Vault Operations

Duration: 15 minutes

In this lab, you are going to perform the following tasks:

- Task 1: Generate a root token
- Task 2: Rekeying Vault
- Task 3: Rotate the encryption key

Task 1: Generate a root token

A recommended best practice is to not persist your root tokens. The root tokens should be used only for just enough initial setup or in emergencies. Once appropriate policies are created, use tokens with assigned set of policies based on your role in the organization.

When a situation arise and you need a root token, this task walks you through the steps to generate one.

Step 5.1.1

Get help on the `vault operator generate-root` command:

Execute

```
$ vault operator generate-root -h
```

Step 5.1.2

Execute the following command to generate a one-time password (OTP) and save it in the `otp.txt` file:

Execute

```
$ vault operator generate-root -generate-otp > otp.txt
```

Step 5.1.3

Execute the following command to initialize a root token generation with the OTP code (`otp.txt`) and save the resulting nonce in the `nonce.txt` file:

Execute

```
$ vault operator generate-root -init -otp=$(cat otp.txt) -format=json \
      | jq -r ".nonce" > nonce.txt
```

For the convenience of this lab, the above command outputs the response in JSON format (`-format=json`), and then uses `jq` tool (https://stedolan.github.io/jq/) to parse the output, retrieve only the nonce value, and store it in the `nonce.txt` file.

If you execute the command without these parameters, the output would look similar to the following.

```
Nonce         e856b8f3-97aa-8c43-c8ed-b28e2825b09b
Started       true
Progress      0/1
Complete      false
OTP Length    26
```

The **nonce** value (`nonce.txt`) should be distributed to all unseal key holders.

The generation of a root token requires a quorum of unseal keys.

Step 5.1.4

Each unseal key holder must execute the following command along with their unseal key. In this lab, you only have one unseal key.

Execute

```
$ vault operator generate-root -nonce=$(cat nonce.txt) $(grep 'Key 1:' key.txt | awk '{print $NF}')
```

The output should look similar to:

```
Nonce            e7a59dbc-4cc6-c49f-dfe9-0e585a8aefa9
Started          true
Progress         1/1
Complete         true
Encoded Token    BAB/NUgSDCIbDFt0AWtEaBQlBCApODU/
```

Step 5.1.5

Copy the resulting `Encoded Token` value.

Step 5.1.6

Execute the following command to decode the encoded token:

Execute

```
$ vault operator generate-root -decode=<Encoded Token> \
      -otp=$(cat otp.txt)
```

Copy the resulting root token.

Step 5.1.7

Now, verify the newly generated root token:

Execute

```
$ vault login <new_root_token>
```

```
Key                  Value
---                  -----
token                3zNdyJ...
token_accessor       1xm6s8C...
token_duration       ∞
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
```

The output should show that the token policy is **root**.

Task 2: Rekeying Vault

During the initialization, the encryption keys and unseal keys were generated. This only happens **once** when the server is started against a new backend that has never been used with Vault before.

Under some circumstances, you may want to re-generate the master key and key shares. For examples:

- Someone joins or leaves the organization
- Security wants to change the number of shares or threshold of shares
- Compliance mandates the master key be rotated at a regular interval

IMPORTANT: If you chose ***Option 1*** in ***Lab 3*** and you enabled auto-unseal (**Vault 2**), your master key is now protected by **Vault 1**. Therefore, the rekey operation has a slightly different meaning. Instead of regenerating the unseal keys, rekey regenerates the **recovery** keys. Recovery keys are used for high-privilege operations such as root token generation you performed in Task 1. Recovery keys are also used to make Vault operable if Vault has been manually sealed through the " `vault operator seal` " command.

Step 5.2.1

First, initialize a rekeying operation. At this point, you can specify the desired number of key shares and threshold. Execute the following command to rekey Vault where the number of key shares is `3` and key threshold is `2` and save the generated nonce in `nonce.txt`.

Execute

```
$ vault operator rekey -init -key-shares=3 -key-threshold=2 \
      -format=json | jq -r ".nonce" > nonce.txt
```

If you performed **Vault 2** tasks in Option 1 of Lab 3, execute the command with `-target="recovery"` flag.

Execute

```
$ vault operator rekey -init -key-shares=3 -key-threshold=2 \
        -target="recovery" -format=json | jq -r ".nonce" > nonce.txt
```

All shared key holders must provide this nonce value to rekey. This nonce value is NOT a secret, so it is safe to distribute over insecure channels like chat, email, or carrier pigeon.

Step 5.2.2

At the moment, there is only **one** unseal key in this lab. Execute the following command to rekey:

Execute

```
$ vault operator rekey -nonce=$(cat nonce.txt) $(grep 'Key 1:' key.txt | awk '{print $NF}')
```

NOTE: If you performed **Vault 2** tasks in Option 1 of Lab 3, execute the command with " `-target="recovery"` " flag.

Vault will output the new unseal keys similar to following:

```
Key 1: a4By/JU6xqMxXG95FtcShLldGS4GDZmcUcCD4Q83cl2b
Key 2: dWBDfbTicxDwCbmi7TQnKBdecdyfWWi+25Pj2xN+vlnb
Key 3: zZk7kYLu02E/UENLmCjBSzu76SQaqnVt9RtcYeTQYsf4

Operation nonce: cc9f9311-7945-3b91-9af4-96d94eba83ae

Vault rekeyed with 3 key shares an a key threshold of 2. Please
securely distributed the key shares printed above. When the Vault
is re-sealed, restarted, or stopped, you must supply at least 2 of
these keys to unseal it before it can start servicing requests.
```

Vault supports **PGP encrypting** the resulting unseal keys and creating backup encryption keys for disaster recovery.

> The generated keys are the new unseal keys required to unseal your Vault server. Replace the contents of your `key.txt` file with new keys.

Discussion: Auto-unseal Key Rotation

When auto-unseal was enabled, your master key is protected by the cloud provider's key and NOT by the Shamir's keys. If the recovery keys have nothing to do with your master key, how do you rotate the encryption key that is protecting your master key?

The answer is to rotate your cloud provider's key.

In Lab 3, *Transit Auto-Unseal* method was used. Therefore, unseal key rotation equals to transit key rotation. To perform the key rotation, execute the following command on the Vault where you configured the transit engine in Lab 3.

Execute the following command to rotate the encryption key:

Execute

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

To view key versions:

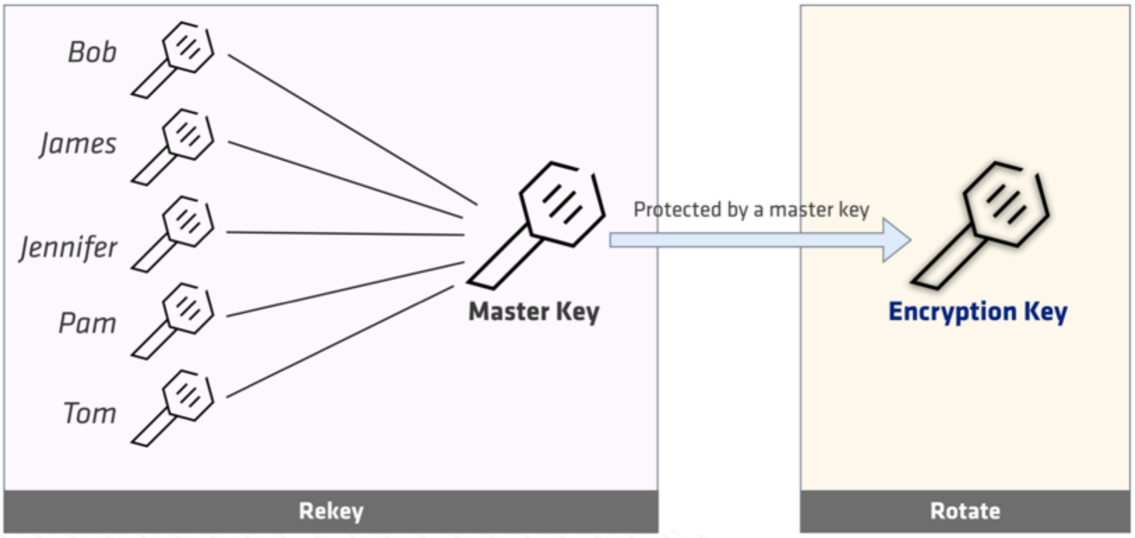
Execute

```
$ vault read transit/keys/autounseal
```

```
Key                       Value
---                       -----
allow_plaintext_backup    false
deletion_allowed          false
derived                   false
exportable                false
keys                      map[1:1553708455 2:1553901526]
...
```

Task 3: Rotate the encryption key

In addition to rekeying the master key, there may be an independent desire to rotate the underlying encryption key Vault uses to encrypt data at rest.



In Vault, `rekeying` and `rotating` are two separate operations. The process for generating a new master key and applying Shamir's algorithm is called "rekeying". The process for generating a new encryption key for Vault to encrypt data at rest is called "rotating".

Step 5.3.1

Unlike rekeying the Vault, rotating Vault's encryption key **does not** require a quorum of unseal keys. Anyone with the proper permissions in Vault can perform the encryption key rotation.

To trigger a key rotation, execute the following command:

Execute

```
$ vault operator rotate
```

The output shows the key version and installation time. This will add a new key to the keyring. All new values written to the storage backend will be encrypted with this new key.

End of Lab 5

Lab 6: 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 6.1.1

Let's review the policy file, `/workstation/vault102/base.hcl`

Execute

```
$ cat base.hcl
```

Step 6.1.2

The policy defines the following rule:

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

path "kv/data/+/apikey" {
   capabilities = ["create", "read", "update", "delete"]
}
```

Notice that the path has the "splat" operator (`training_*`) as well as single directory wildcard (`+`). This is helpful in working with namespace patterns.

> When you are working with *key/value secret engine v2*, the path to write policy would be `kv/data/<path>` even though the CLI command to the path is `kv/<path>`. When you are working with **v1**, the policy should be written against `kv/<path>`. This is because the API endpoint to invoke key/value v2 is different from v1.

Step 6.1.3

Get help for the vault policy command:

Execute

```
$ vault policy -h
```

Step 6.1.4

Execute the following commands to create a policy:

Execute

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

Step 6.1.5

Execute the following CLI command to list existing policy names:

Execute

```
$ vault policy list
```

Expected output:

```
autounseal
base
default
root
```

Step 6.1.6

To read back the `base` policy, execute the following command:

Execute

```
$ vault policy read base
```

Step 6.1.7

Enabled Key/Value v2 secrets engine at `kv` path by executing the following command:

Execute

```
$ vault secrets enable -path=kv/ kv-v2
```

Step 6.1.8

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:

Execute

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

Example output:

```
Key                  Value
---                  -----
token                s.Zh71oWCxDCZETPtLrvFLBF7m
token_accessor       Hgs1Zqg6tWdAaxrHcjxHJh1l
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 6.1.9

Authenticate with Vault using the token generated at *Step 6.1.8*:

Example:

```
vault login s.Zh71oWCxDCZETPtLrvFLBF7m
```

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 6.2.1

Using the base token, you have very limited permissions.

Execute

```
$ 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 6.2.2

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

Execute

```
$ vault kv put kv/training_test password="p@ssw0rd"
```

```
Key              Value
---              -----
created_time     2019-03-19T19:18:58.731152233Z
deletion_time    n/a
destroyed        false
version          1
```

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

Step 6.2.3

Read the data back:

Execute

```
$ vault kv get kv/training_test
```

```
====== Metadata ======
Key              Value
---              -----
created_time     2019-03-19T19:18:58.731152233Z
deletion_time    n/a
destroyed        false
version          1

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

Step 6.2.4

Pass a different password value to update it.

Execute

```
$ vault kv put kv/training_test password="password1234"
```

```
Error writing data to kv/training_test: Error making API request.

URL: PUT http://127.0.0.1:8200/v1/kv/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**.

Step 6.2.5

Execute the following command:

Execute

```
$ vault kv put kv/team-eng/apikey api_key="123456789"
```

The path `kv/team-eng/apikey` matches the `kv/<string>/apikey` pattern, so the command should execute successfully.

Step 6.2.6

Since the policy allows **delete** operation, the following command should execute successfully as well:

Execute

```
$ vault kv delete kv/team-eng/apikey
```

Question

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

Execute

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

Will this work?

Answer:

This is going to work.

```
Key              Value
---              -----
created_time     2019-03-19T19:20:57.053554118Z
deletion_time    n/a
destroyed        false
version          1
```

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 6.3.1

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

Execute

```
$ 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 6.3.2

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

Execute

```
$ vault token capabilities kv/data/training_dev
```

Expected output:

```
create, read
```

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

Step 6.3.3

How about `kv/data/splunk/apikey` path?

Execute

```
$ vault token capabilities kv/data/splunk/apikey
```

Expected output:

```
create, delete, read, update
```

Step 6.3.4

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

Execute

```
$ vault token capabilities kv/data/test
```

Expected output:

```
deny
```

Step 6.3.5

Now, authenticate with root token again. (Remember that the initial root token is stored in the `key.txt` file.)

Execute

```
$ vault login $(grep 'Initial Root Token:' key.txt | awk '{print $NF}')
```

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 `kv/data/exercise`
2. Forbids any operation against `kv/data/exercise/team-admin` (this is an exception to the requirement #1)
3. Forbids **deleting** anything in paths prefixed with `kv/data/exercise`
4. List existing policies (CLI command: `vault policy list`)
5. View available auth methods (CLI command: `vault auth list`)

> 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

To write policies, you need to know the exact path which maps to the API endpoint.

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

```
...
{
  ...
  "request": {
    "id": "7d9bad0e-f3d1-c8be-9f6a-01dd2c869a1e",
    "operation": "update",
    "client_token": "hmac-sha256:793057fc17d1f4c959d2b31b3c07191...",
    "client_token_accessor": "hmac-sha256:a27bcedad78e3afb3ed01d...",
    "namespace": {
      "id": "root",
      "path": ""
    },
    "path": "kv/data/exercise/test",
    "data": {
      "data": {
        "date": "hmac-sha256:69777dec8f24e96bb76a7ec2311d61371e91..."
      },
      "options": {}
    },
    ...
```

Another convenient trick you should remember is **-output-curl-string** CLI flag. For example, to find out the cURL equivalent of API call:

cURL equivalent for `vault policy list`:

Execute

```
$ vault policy list -output-curl-string
```

cURL equivalent for `vault auth list`

Execute

```
$ vault auth list -output-curl-string
```

The resulting API endpoint tells you for which path you need to write a policy for.

Lab 6: Working with Policies - Challenge Solution

Verification

Before looking at the sample solution, test your policy for both **happy** path and **failure** path. Make adjustments to your policy until you it fulfills the policy requirements. Sample solution is provided so that you can compare it with your policy.

Execute

```
$ # 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 kv/exercise/test date="today"
$ vault kv get kv/exercise/test
$ vault token capabilities kv/data/exercise/test

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

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

# Test requirement 4
$ vault policy list

# Test requirement 5
$ vault auth list

# Finally, log back in with root token
$ vault login $(grep 'Initial Root Token:' key.txt | awk '{print $NF}')
```

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 "kv/data/exercise/*" {
     capabilities = [ "create", "read", "update" ]
}

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

# Requirement 4
path "sys/policies/acl" {
     capabilities = [ "list" ]
}

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

End of Lab 6

Lab 7: Secure Introduction with Vault Agent

Duration: 10 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: Test Vault Agent Caching
- Task 3: Evict Cached Leases

Resources:

- Vault Agent: https://www.vaultproject.io/docs/agent/

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.

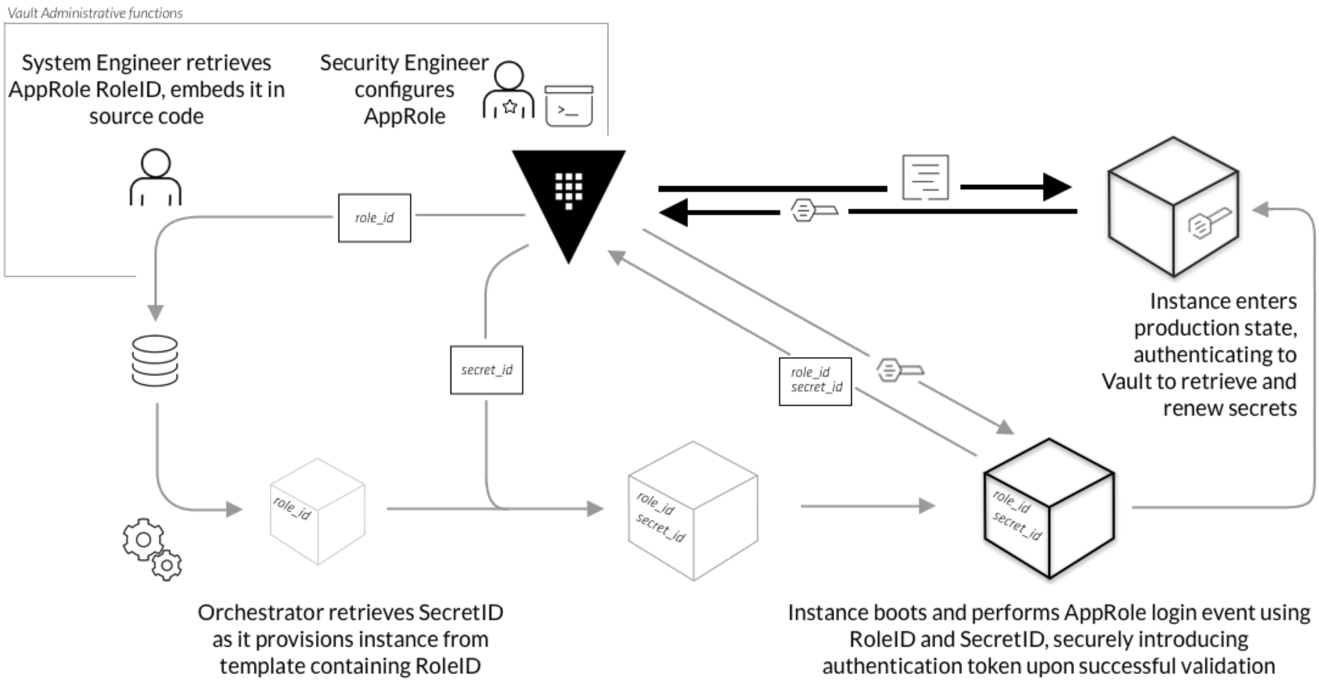
Step 7.1.1

First, review the `/workstation/vault102/setup-approle.sh` script to examine what it performs.

Execute

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

This script creates a new policy called, `db_readonly`, and enable `database` secrets engine. It also 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.


Vault Administrative functions
System Engineer retrieves AppRole RoleID, embeds it in source code
Security Engineer configures AppRole
role_id
secret_id
role_id
secret_id
role_id
role_id
secret_id
role_id
secret_id
Instance enters production state, authenticating to Vault to retrieve and renew secrets
Orchestrator retrieves SecretID as it provisions instance from template containing RoleID
Instance boots and performs AppRole login event using RoleID and SecretID, securely introducing authentication token upon successful validation

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/vault102/agent-config.hcl`.

Execute

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

Expected contents:

```
exit_after_auth = false
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/vault102/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/vault102/approleToken` (this is the sink location).

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

Execute the following command to start the Vault Agent with `debug` logs.

Execute

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

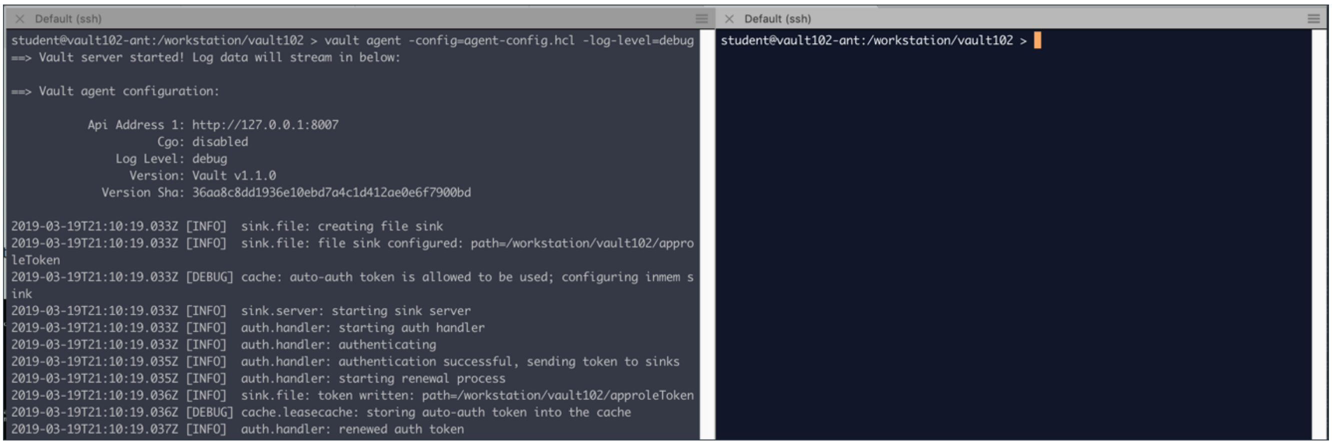
Output should look similar to:

```
==> 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 v1.3.2
          Version Sha: 36aa8c8dd1936e10ebd7a4c1d412ae0e6f7900bd
...
```

Task 2: Test Vault Agent Caching

Step 7.2.1

Open another terminal and then SSH into your student workstation if you don't have one already. Be sure to change the working directory to `/workstation/vault102`. Place the two terminals side-by-side if possible so that you can examine the logs as you execute commands.



```
student@vault102-ant:/workstation/vault102 > vault agent -config=agent-config.hcl -log-level=debug
==> 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

2019-03-19T21:10:19.033Z [INFO]  sink.file: creating file sink
2019-03-19T21:10:19.033Z [INFO]  sink.file: file sink configured: path=/workstation/vault102/appro
leToken
2019-03-19T21:10:19.033Z [DEBUG] cache: auto-auth token is allowed to be used; configuring inmem s
ink
2019-03-19T21:10:19.033Z [INFO]  sink.server: starting sink server
2019-03-19T21:10:19.033Z [INFO]  auth.handler: starting auth handler
2019-03-19T21:10:19.033Z [INFO]  auth.handler: authenticating
2019-03-19T21:10:19.035Z [INFO]  auth.handler: authentication successful, sending token to sinks
2019-03-19T21:10:19.035Z [INFO]  auth.handler: starting renewal process
2019-03-19T21:10:19.036Z [INFO]  sink.file: token written: path=/workstation/vault102/approleToken
2019-03-19T21:10:19.036Z [DEBUG] cache.leasecache: storing auto-auth token into the cache
2019-03-19T21:10:19.037Z [INFO]  auth.handler: renewed auth token
```

```
student@vault102-ant:/workstation/vault102 >
```

Step 7.2.2

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

Execute

```
$ more approleToken
```

Output should look similar to:

```
s.DL0ToAJKVjOSXXZdfzAKPWLY
```

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

```
[INFO]  sink.file: creating file sink
[INFO]  sink.file: file sink configured: path=/workstation/vault102/approleToken
[DEBUG] cache: auto-auth token is allowed to be used; configuring inmem sink
```

Step 7.2.3

Set the `VAULT_AGENT_ADDR` environment variable.

Execute

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

Step 7.2.4

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
```

Output should look similar to:

```
Key                  Value
---                  -----
token                s.qaPOodPTUdtbj5REak2ICuyg
token_accessor       Bov810fwIPlp48bENCuW8xv9
token_duration       30s
token_renewable      true
token_policies       ["db_readonly" "default"]
identity_policies    []
policies             ["db_readonly" "default"]
```

The created token has a TTL of **30 seconds**. It can be renewed; however, its max TTL is **2 minutes**.

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 7.2.5

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
```

Vault Agent renews the token before its TTL until the token reaches its maximum TTL (2 minutes). Once the token reaches its max TTL, agent fails to renew it because the Vault server revokes it.

```
[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
```

When the token renewal failed, the agent automatically evicts the token from the cache since it's a stale cache.

Step 7.2.6

Now, request database credentials for role, "readonly" which was configured by the `setup-approle.sh` script.

Execute

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

Output should look similar to:

```
Key                Value
---                -----
lease_id           database/creds/readonly/2TW9uVXkMB5oBw1DhtgzQZZb
lease_duration     1h
lease_renewable    true
password           A1a-5ZqdiR8AD5N46Mk6
username           v-approle-readonly-vFmdbjZ1HGXsKsKPTzpa-1552079424
```

You should find the following entires 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 7.2.7

Re-run the same command and examine the behavior.

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-vFmdbjZ1HGXsKsKPTzpa-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 7.2.8

Now, invoke the following command to see the agent behavior.

Execute

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

```
Key                Value
---                -----
lease_id           database/creds/readonly/q5jFoOHmDrpcLFeXGpCWzkEY
lease_duration     1h
lease_renewable    true
password           A1a-VUo0zaSVEl7HuRob
username           v-root-readonly-ccWqTFwA55GG22suOH2p-1554271579
```

The agent log indicates "pass-through lease response" which means that the returned lease (database credentials) was not cached. This is because the request was made with token which is not managed by the Vault agent.

```
[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: pass-through lease response; token not managed by agent: path=/v1/database/creds/readonly method=GET
```

Since you have been working with root token, without overwriting with "VAULT_TOKEN=$(cat approleToken)", the agent uses the root token.

Lookup current token.

Execute

```
$ vault token lookup
```

Output should look similar to:

```
Key                 Value
---                 -----
...
display_name        root
...
path                auth/token/root
policies            [root]
...
```

Vault Agent caches leased secret that were requested using tokens that are already managed by the agent.

Task 3: Evict Cached Leases

While agent observes requests and evicts cached entries automatically, you can trigger a cache eviction by invoking the `/agent/v1/cache-clear` endpoint.

Step 7.3.1

To evict a lease, invoke the `/agent/v1/cache-clear` endpoint along with the ID of the lease you wish to evict.

Execute

```
$ curl -X POST -d '{"type": "lease", "value": "<lease_id>"}' \
       $VAULT_AGENT_ADDR/agent/v1/cache-clear
```

For example, if the `lease_id` from ***Step 7.2.6*** is `database/creds/readonly/2TW9uVXkMB5oBw1Dh`, the command would look as follow:

Execute

```
$ curl -X POST \
    -d '{"type": "lease", "value": "database/creds/readonly/2TW9uVXkMB5oBw1Dh"}' \
    $VAULT_AGENT_ADDR/agent/v1/cache-clear
```

In the agent log, you find the following:

```
[DEBUG] cache.leasecache: received cache-clear request: type=lease namespace= value=database/creds/readonly/2TW9uVXkMB5oBw1Dh
[DEBUG] cache.leasecache: canceling context of index attached to accessor
[DEBUG] cache.leasecache: successfully cleared matching cache entries
[DEBUG] cache.leasecache: context cancelled; stopping renewer: path=/v1/database/creds/readonly
[DEBUG] cache.leasecache: evicting index from cache: id=d05ef71852c49d7390f7a2fd76de7e5f85d668e7ffc9c2cb76a602dcf9bb7a11 path=/v1/database/creds/readonly method=GET
```

Step 7.3.2

If a situation requires you to clear all cached tokens and leases (e.g. reset after a number of testing), set the type to all.

Execute

```
$ curl -X POST -d '{"type": "all"}' $VAULT_AGENT_ADDR/agent/v1/cache-clear
```

In the agent log, you find the following:

```
[DEBUG] cache.leasecache: received cache-clear request: type=all namespace= value=
[DEBUG] cache.leasecache: canceling base context
[DEBUG] cache.leasecache: successfully cleared matching cache entries
```

Step 7.3.3

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

End of Lab 7