Vault & neonCLUSTERs

# Introduction

Neon Clusters use the open source HashiCorp [Vault](https://www.vaultproject.io/) project for managing secrets. Vault provides the following capabilities out of the box:

* Secure secret storage (encrypted key/value store)
* Access tokens, policies and roles
* Secret leases, rolling, revocation, and access auditing
* Encryption as a service
* Dynamic secret generation (e.g. temporary AWS, SSH or MS-SQL credentials)
* Simple PKI infrastructure

Vault is quite flexible and can be deployed in many different ways. This document describes how we’ll be using Vault in Neon Clusters.

# Deployment

The neon-cli tool deploys Vault and Consul to the cluster manager nodes where one, three, or five managers are supported. One manager node is suitable for development and testing purposes. Production clusters should deploy three or five manager nodes for high-availability.

Vault persists all of its secrets to Consul, encrypting any information before saving it. The key Vault uses for this encryption is not persisted anywhere within a cluster. This key is generated during Vault initialization and must be manually provided whenever a Vault cluster is restarted. This process is termed unsealing the Vault using unseal keys. By default, neon-cli will generate a single unseal key during cluster setup. You can modify the Vault.KeyCount and Vault.KeyThreshold cluster definition settings to break the encryption key up into multiple parts (KeyCount) and also specify the minimum number or unseal keys required to unseal Vault (KeyThreshold). Multipart keys are used to minimize the trust required of any individual operators (e.g. the two-key rule for nuclear missile launches).

Operators can unseal a cluster’s Vault remotely using the neon vault unseal <key> command or on a manager node via the vault unseal <key> command. Note that you and/or other operators will need to run this command at least KeyThreshold times with valid key parts. Note that you only need to do this for one Vault instance in a cluster; any others will be unsealed automatically.

Vault and Consul are both deployed as host node level services, not as Docker containers. Both services expose services locally using their standard HTTP ports (Vault=8200 and Consul=8500). neon-cli deploys the neon-proxy-vault load-balancer/fail-over service as the cluster wide mechanism to access Vault on the reserved cluster port 5003. The VAULT\_ADDR environment variable on each cluster host will be set to access Vault through this load-balancer and VAULT\_ADDR will also be initialized in the /etc/neoncluster/env-host**[[1]](#footnote-1)** script file that can be mounted to Docker containers and executed by a container’s startup script.

Vault internal network traffic is secured out-of-the-box. Client traffic can be encrypted by specifying a TLS certificate and private key in the cluster configuration. The **neon-cli** tool will write these files to the host root account’s **$HOME/secrets** folder.

Programs will require a token to access the Vault. These are GUIDs created by Vault commands, using another (parent) token. The **neon-cli** tool retrieves and saves the original root token created when Vault is initialized. This and is saved to the operator’s remote machine and has the unlimited right to perform any Vault operations (including creating tokens with potentially fewer rights). The root token is included in the client file along with the one or more keys required to unseal the Vault.

We’ll discuss tokens in detail below, but it’s important to note that the neon-cli tool creates a Vault token for each Docker host and writes it to the root account’s **$HOME/.vault-token** file. This is default place where the vault CLI looks for a token. The combination of this token and the VAULT\_ADDR environment variables means that operators or processes logged in to the Docker host can use the vault CLI without specifying a token or a Vault URL.

# Brief Vault Overview

This section briefly describes some basic Vault capabilities. For more information, please review the [Introduction to Vault](https://www.vaultproject.io/intro/index.html) and the [Vault Documentation](https://www.vaultproject.io/docs/index.html).

Vault is a tool for securely accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, and more. Vault provides a unified interface to any secret, while providing tight access control and recording a detailed audit log.

A modern system requires access to a multitude of secrets: database credentials, API keys for external services, credentials for service-oriented architecture communication, etc. Understanding who is accessing what secrets is already very difficult and platform-specific. Adding on key rolling, secure storage, and detailed audit logs is almost impossible without a custom solution. This is where Vault steps in.

Here are some key Vault features:

**Secure Secret Storage**: Arbitrary key/value secrets can be stored in Vault. Vault encrypts these secrets prior to writing them to persistent storage, so gaining access to the raw storage isn't enough to access your secrets. Vault can write to disk, [Consul](https://www.consul.io), and more.

**Dynamic Secrets**: Vault can generate secrets on-demand for some systems, such as AWS or SQL databases. For example, when an application needs to access an S3 bucket, it asks Vault for credentials, and Vault will generate an AWS keypair with valid permissions on demand. After creating these dynamic secrets, Vault will also automatically revoke them after the lease is up.

Data Encryption: Vault can encrypt and decrypt data without storing it. This allows security teams to define encryption parameters and developers to store encrypted data in a location such as SQL without having to design their own encryption methods.

Leasing and Renewal: All secrets in Vault have a *lease* associated with them. At the end of the lease, Vault will automatically revoke that secret. Clients are able to renew leases via built-in renew APIs.

Revocation: Vault has built-in support for secret revocation. Vault can revoke not only single secrets, but a tree of secrets, for example all secrets read by a specific user, or all secrets of a particular type. Revocation assists in key rolling as well as locking down systems in the case of an intrusion

Vault secrets and services are accessed by referencing a point in a hierarchy of nodes, something like a file system path. At first level of this hierarchy are what Vault calls backends. Vault currently implements three basic backend types:

Secret Backends: These are used to manage secrets like database passwords, API keys, and TLS certificates and private keys. Vault provides several backends for transparently persisting secrets to MySQL, MSSQL, MongoDB, Consul, etc. neonHIVEs currently persist to Consul. Vault also provides the [Generic](https://www.vaultproject.io/docs/secrets/generic/index.html), [CubbyHole](https://www.vaultproject.io/docs/secrets/cubbyhole/index.html), [Transit](https://www.vaultproject.io/docs/secrets/transit/index.html'), and [PKI](https://www.vaultproject.io/docs/secrets/pki/index.html) backends:

GenericUsed to save arbitrary secrets as key/value pairs. By default, this all keys are rooted at /secret/\* but it is possible to mount additional generic backends with different roots. Keys are paths like /secret/my-service/db-credentials. Values are strings. Binary data must be encoded as Base-64, hexadecimal or something.

CubbyHoleUsed to create a limited use secret that wraps another secret. This can be used to help secure the delivery of a secret from one entity to another. This is mounted at cubbyhole.

TransitProvides encryption as a service. It can encrypt and decrypt textual data, sign and verify data, generate hashes and HMACs, and act as a random source of bytes. Transit backends manage the encryption keys internally and also support key rotation by maintaining a keyring such that data encrypted with older keys can still be decrypted. Transit backends are manually mounted at specific root points and multiple Transit backends can be provisioned so different services won’t share encryption keys.

PKIThis implements a simple PKI setup using self-signed certificates. This is rooted at pki and is designed to issue dynamic client certificates and private keys based on configured roles.

Auth Backends: Used to authenticate people or services using Vault. There are several of these, including:

AppRole Role based service authentication rooted at /auth/approle.

Token Token based authentication rooted at /auth/token.

UID/PWD User ID and password authentication rooted at /auth/userpass. Vault maintains the data internally and also maps users to roles. This wouldn’t be suitable for managing end-user credentials but is suitable for managing accounts for system operators.

Others Vault can also integrate with GitHub & AWS accounts, LDAP, and can also verify client TLS certificates.

Audit Backends: Vault can be configured to send audit logs to syslog or files.

# Vault and neonHIVE

Vault is very flexible and neonHIVEs currently takes advantage of only a limited set of features out-of-the-box. Note that this doesn’t limit more sophisticated cluster and service deployments from using additional features. This section describes how Vault is configured for neonHIVE.

As described above Vault is deployed to the cluster manager nodes in a high-availability configuration using Consul as the persistence backend (also deployed to the manager nodes). The neon-vault-lb Docker service provides cluster wide load-balancing/fail-over on all Docker hosts on port **5003**. The VAULT\_ADDR environment variable can be used on both Docker hosts as well as containers that mount and execute /etc/neoncluster/env-host within their entry point script.

The neon setup command initializes Vault, retrieving one or more unseal keys as well as the initial root token. These are displayed in the tool’s output and can also be saved to a JSON file. By default, Vault is configured to such that the maximum and default secret leases will be essentially infinite (290 years). It is possible to change this in the cluster definition but this is not recommended at this time.

The neon-cli automatically mounts or enables the following backends:

/neon-secret **generic** secrets backend reserved for internal use by neonHIVE related hosts and services. Application services should use the default secret backend or mount a custom one.

/secret generic secrets backend available for application services.

/transit Encryption-as-a-service backend that can be used by neonHIVE and application services. Encryption keys with names like neon-\* are reserved for use by neonHIVE related services and the neon-cli creates a key called neon during cluster setup. Application services should create their own keys.

/auth/token Root path for authentication tokens.

/auth/approle Role based authentication.

/audit Auditing is enabled for delivery to syslog as the AUTH facility.

## Vault Policies and Roles

Vault uses policies to restrict or grant tokens from performing specific operations on specific paths. These are Vault’s standard way of managing authorization. Policies are specified as HCL or JSON as described [here](https://www.vaultproject.io/docs/concepts/policies.html). Vault defines two build-in policies named root and default. root grants tokens unlimited access to the Vault and default allows a token to manage itself (e.g. renew, revoke, access its cubbyhole…).

neonHIVE reserves policy names like neon\*. The neon-cli tool creates these policies:

neon-manager This currently grants unlimited access to the entire Vault. This may be more restrictive in the future.

neon-worker This also currently grants unlimited access to the entire Vault. This will definitely be more restrictive in the future.

# Security

This section discusses current security related gaps and issues.

## Trust Assumptions

Every real-world service deployment needs to make certain assumptions about trustworthiness. neonHIVE deployments are no different. The list below outlines the current trust assumptions:

1. Physical: Servers are assumed to be physically secure such that attackers cannot use BIOS to boot another image or remove disk drives.
2. Memory & Filesystem: Operating system security prevents unauthorized access to memory and files.
3. Root account on all machines has a secure password and cannot be hacked.
4. Docker Overlay Networks are effectively isolated from the physical datacenter network.
5. Firewall blocks all traffic except for specific externally facing ports (like 80/443) to cluster nodes.
6. Secure VPN is used by operators to manage services using SSH, SCP and other mechanisms.
7. Operators keep the SSH, Vault tokens & unseal keys, and other secrets safe.
8. Vault really is secure.

## Security Issues

Ideally we’d like to be able to relax physical trust assumptions. It would be wonderful if an attacker would have access to no secrets on a machine. We’re not there yet. I’ve spent some time trying to address this by disabling memory swapping for key services (like Vault), deployed a service to purge the Bash history, and attempting to encrypt parts of the file system using [ecryptfs](http://ecryptfs.org/)).

Unfortunately, I couldn’t get ecryptfs to work. I wanted to use it to encrypt the root account’s **$HOME** folder which for manager nodes stores the Vault TLS certificate and private key and for all nodes, the machine’s Vault token. The basic problem was that ecryptfs requires that an operator provides a password by logging into the machine. This won’t happen for services that start automatically.

The Vault TLS private key is the main vulnerability here. With this, an attacker could act as a man-in-the-middle and view all secrets as they are written and read from Vault. One possible mitigation is to move Vault off cluster into a separate, isolated, more secure set of servers.

Vault 0.6 implemented the cool CubbyHole backend. This provides an easy way to wrap one secret into another limited use secret. A key motivation for this was to provide a way to pass a token or other secret to a Docker container when it is started. For example, an operator scheduler service could:

1. Create a token for a service with access to the secrets the service requires.
2. Create a temporary token with a limited lifespan (say 2 minutes) and that limits itself to two read/write accesses.
3. Stores the service token in the temporary token’s cubbyhole (using one access).
4. Starts the container, passing the temporary token (e.g. as an environment variable).
5. The container reads the service token from the temporary token (using the second access).

Note that Vault will automatically delete the temporary token after the second access or when it expires. If someone managed to intercept and read the temporary token before the container, the container won’t be able to read it and an audit event will be logged, which should trigger an investigation.

This is very cool, but it doesn’t really help for Docker swarm mode service orchestration. The basic problem is that Docker is managing the container lifecycle and there’s currently no place where the temporary token can be inserted into the process.

Docker API endpoints are not currently secured. These are currently exposed on the hosts using the standard port and are not secured by TLS. I believe it could be possible to have Docker expose this only internally using Unix sockets[[2]](#footnote-2). The alternative would be to use Vault PKI.

Consul API is also not currently secured. We’d want to do this if services directly stored important information there. I don’t believe this would be necessary if services use Vault to store the secrets because Vault handles authentication and encryption. Vault will be much less scalable though so it may be worth configuring PKI for this at some point.

neonHIVE currently creates tokens or secrets with effectively infinite (290 year) leases. We’re doing this to avoid the complexity of explicitly having to renew leases and the potential service disruptions that might occur if we don’t get this right all the time. It could be possible to mitigate this by explicitly creating new tokens and then revoking the old ones using the neon-cli tool or other mechanisms. We’ll revisit this at a future date.

neonHIVE currently uses the same credentials for the root account for all host machines and also somewhat encourages these credentials to remain static over time. The neon-cli tool should include features that encourage very secure passwords at a minimum. We should also look into using Vault PKI to authenticate client TLS certificates and/or support temporary SSH credentials. We should also consider having different credentials for manager vs. worker nodes since Vault runs on the managers and that’s also where Docker Swarm orchestration happens. This would help prevent a rouge service on a worker from guessing manager credentials opening the managers and the entire cluster to attack.

neonHIVEs currently grant unlimited Vault access to the neon-manager and neon-worker policies. This are just placeholders. The idea here get AppRole based authentication working for the cluster hosts. Once this works, it’ll be easy to modify the policies. We’ll need to design more restrictive policies before going into production.

1. This script also adds the Vault service’s DNS host name to the container’s **/etc/hosts** file. The **neon-cli** does this same for the Docker host node. For the container, the Vault service host will resolve to the host’s IP address and for the host, this will resolve to the loopback address. This is required for HTTPS. [↑](#footnote-ref-1)
2. The only thing this would break is neon-cli code that directly manages Docker volumes in a few places. [↑](#footnote-ref-2)