HashiCorp

Generate dynamic database secrets with HashiCorp Vault Enterprise

Using a Vault Enterprise with Integrated Storage reference architecture on AWS

Mara Hammond (she/her)
HashiCorp CSA Candidate

Agenda

- Vault recap
- Brief on dynamic secrets engine
- Review Vault Enterprise reference architecture design
- Demo Vault Enterprise deployment
- Demo Vault Enterprise to write database dynamic secrets

Who’s in the room?

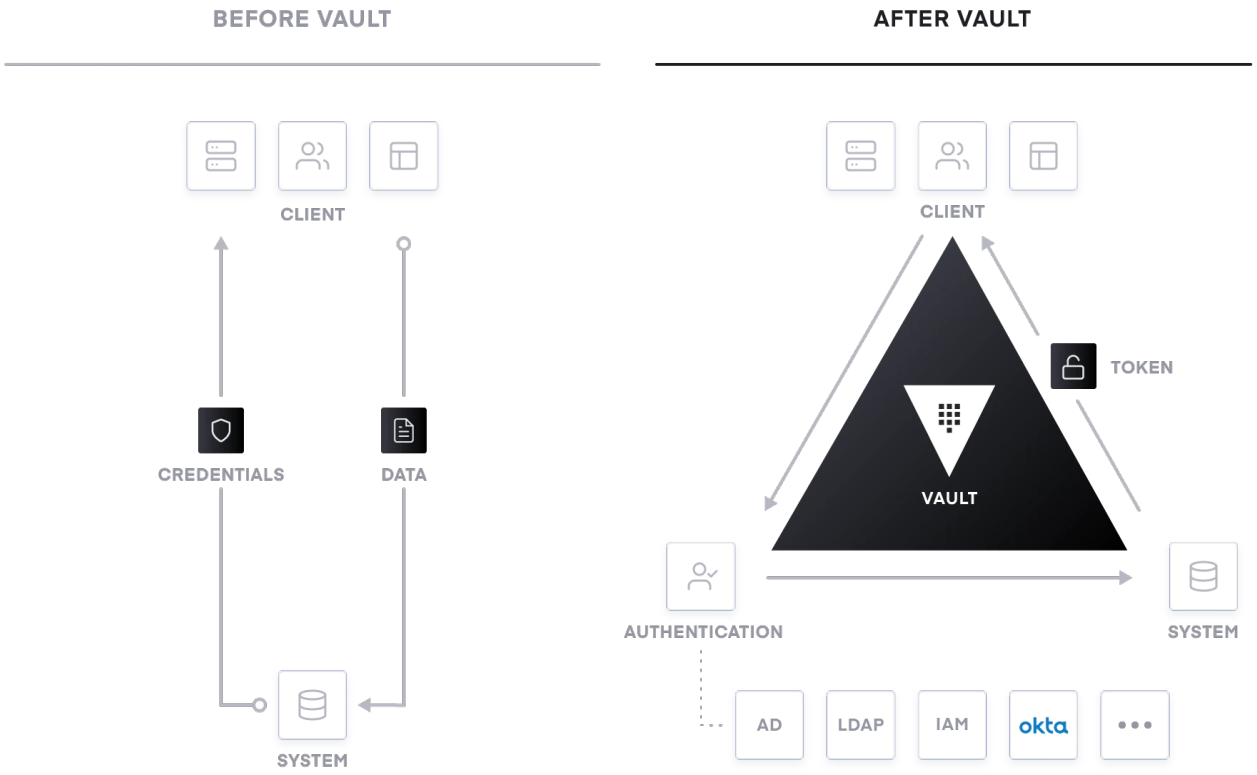
Mara

- 16-year IT veteran
- USC Alum
- Editing credits include Apex Legends, Star Wars
- Cat aficionado
- Long distance Mara-thon runner
- Proud auntie

Let’s Get Into It

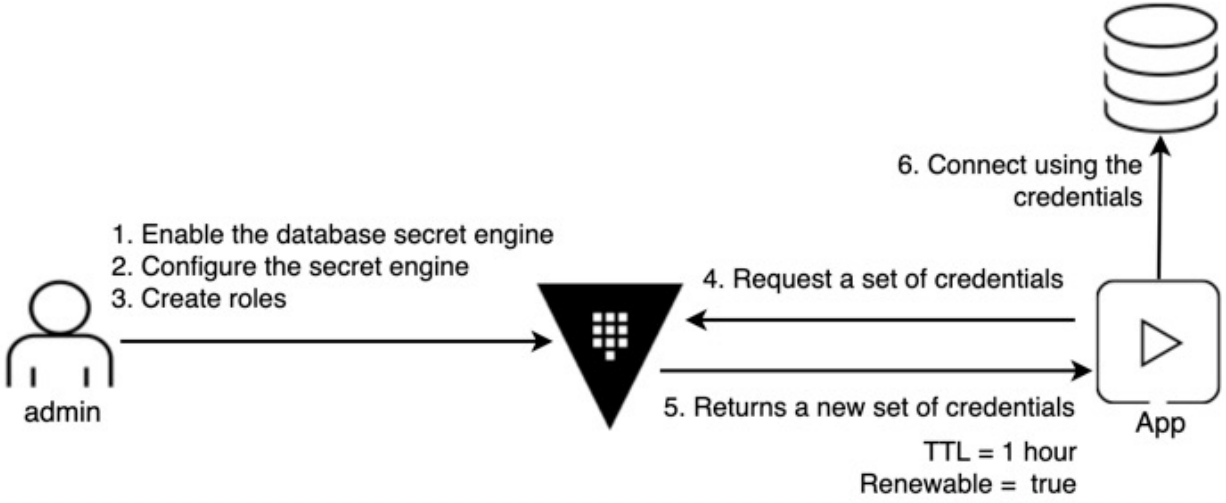
Recap: Security with HashiCorp Vault

- Identity-based security: govern access with policies, trusted identities
- Highly configurable with 100+ integrations, 20+ providers, 30+ secrets engines
- Unified API enables management across providers
- Encryption as a Service delivered just in time with dynamic secrets engines



(Database) Dynamic Secrets

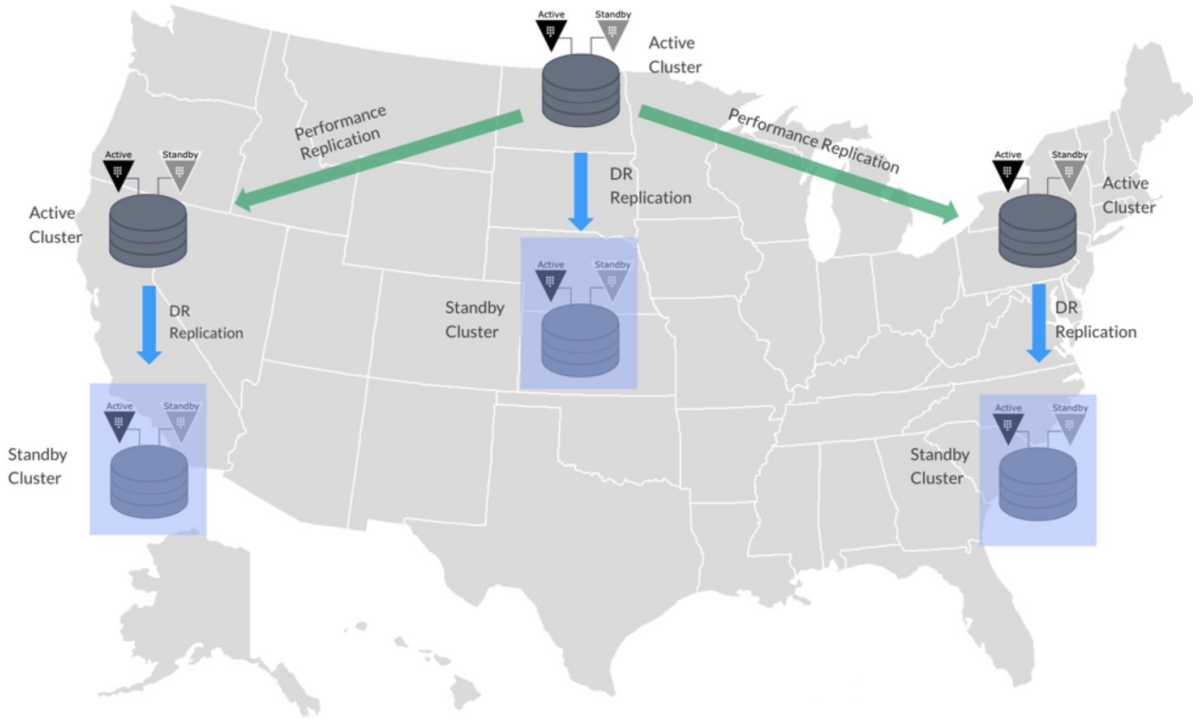
- Primarily used for machine automation, credential vending
- Replaces long-lived, hard-coded secrets
- Short TTL ensures secrets are purpose-built, reduces exposure time
- Vault audits all requests and can be configured to confirm audit write before authorization
- Help contain secrets sprawl across systems – centralize, manage, audit



Vault Enterprise Features

- Replication
  - Extend architecture cross-region with performance and DR replication patterns
- Key namespaces
  - Multi-tenant secrets management for operational, compliance requirements
- Advanced Data Protection
  - Use Vault to transform existing data through TDE, tokenization

HashiCorp
Vault Enterprise



Expectations for Vault Deployment

- Resource provisioning, installation can be completed in minutes
- Requires buy-in from NetOps, InfoSec to provision network, SSL certificates
- App teams will need time to refactor applications
- Load testing is strongly recommended
- Plan strategies for DR, performance replication
- Other considerations like key namespaces, audit requirements, etc.

Prepare for Launch

Technical Prerequisites

- AWS account
- AWS IAM user with necessary permissions to deploy resources and access Session Manager
- Terraform CLI installed locally
- The cloned GitHub repository
- A valid Vault Enterprise license stored as `vault.hclic`

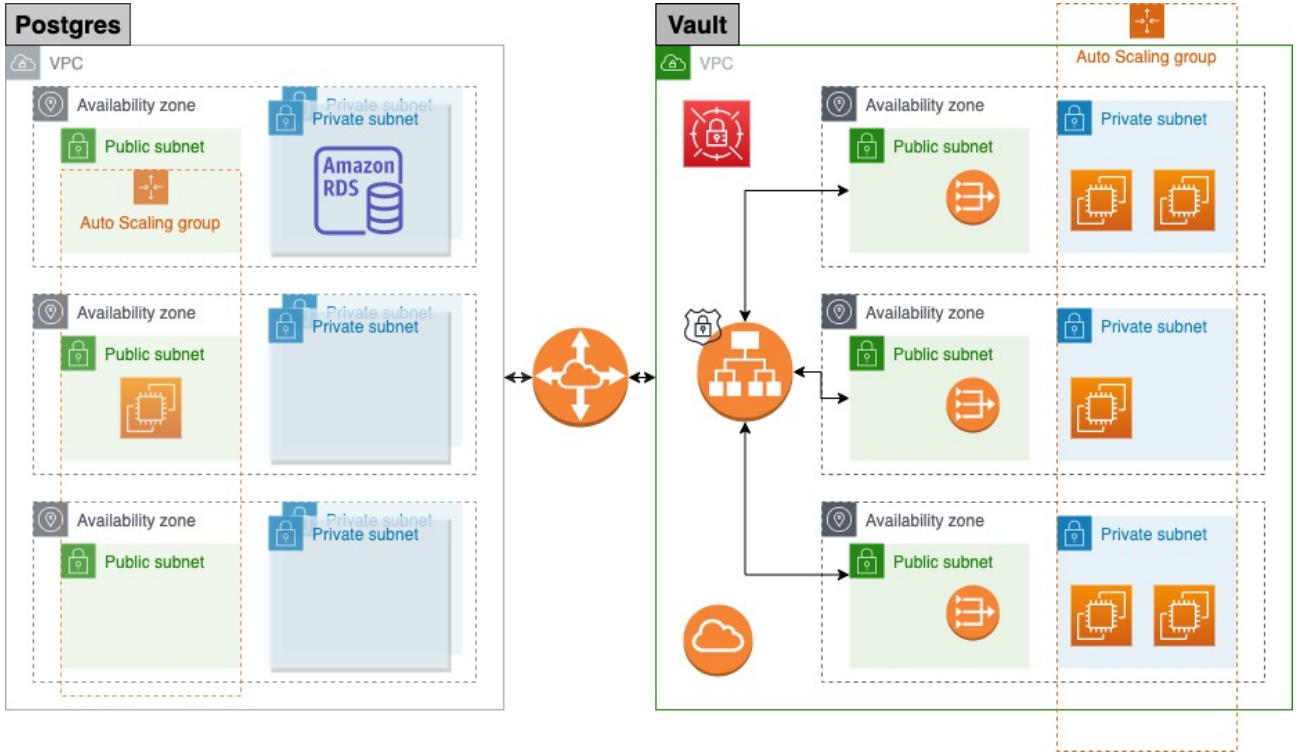
IAM Requirements

Ensure your AWS credentials are configured correctly and have permission to use the following AWS services:

Amazon Certificate Manager (ACM)
Amazon EC2
Amazon Elastic Load Balancing (ALB)
AWS Identity & Access Management (IAM)
AWS Key Management System (KMS)
Amazon Simple Storage Service (S3)
Amazon Secrets Manager
AWS Systems Manager Session Manager Amazon VPC

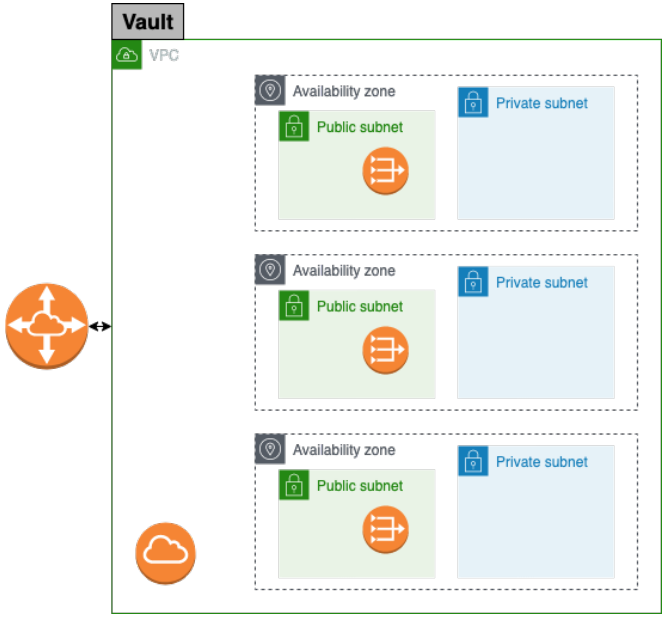
Demo Launch Sequence

- A new AWS VPC for Vault with subnets and NAT gateways deployed across 3 AZs;
- A VPC peering connection between Vault and application environments (Postgres, pre-launched);
- Dev/test: self-signed TLS certificates to secure recovery key and application load balancer;
- A reference architecture cluster deployment of Vault Enterprise 1.10.0, using Integrated Storage



Building a Vault-friendly Network

- New VPC with CIDR range unique to other networks (no overlap)
- 6 subnets deployed across 3 AZs in a region
- 3 NAT gateways deployed across 3 AZs, and 1 internet gateway
- VPC peering connection with application network
- Security Group rules in application environment to permit Postgres, HTTPS traffic



Terraform Note
The `provision-vault-vpc` Terraform configuration will output 2 values when complete: `private_subnet_tags` and `vpc_id`. Save these values for use later by the other demo modules and the Vault installer.

Terraform Note
The `peer-from-vault` Terraform configuration requires the `vpc_id` values for the application network (see Appendix) and Vault network (see above). Specify these IDs in `variables.tf`. Once set, the module will create simple routes between each VPC to properly direct traffic to the opposing network.

Securing a Vault-friendly Network

- In production, SSL certificates need to be signed by domain certificate authority
- Certificate name should match domain name
- Lab demo is creating locally self-signed TLS secrets (not secure)
- Vault recovery keys will be stored in a AWS Secrets Manager secret
- AWS ACM stores load balancer SSL certificate to secure HTTPS listener

AWS Certificate Manager

AWS Secrets Manager

Terraform Note
The `generate-tls` Terraform configuration will output 3 values when complete: `lb_certificate_arn`, `leader_tls_servername`, and `secrets_manager_arn`.
Save these values for use later by Vault installer.

Demo Break

Installing Vault Enterprise

- Configuration pulls in `hashicorp/vault-ent-starter/aws` module
- Instances launched from an autoscaling group, which is triggered by a launch template
- Embedded `user_data` script bootstraps Vault installation on Ubuntu server AMI
- Module deploys KMS key to auto-unseal Vault after initialization
- Vault license must be autoloaded on all nodes – copy `vault.hclic` to subfolder

Terraform Notes

For Vault Enterprise deployment to succeed, use values output from previous modules to replace placeholders for the following variables:

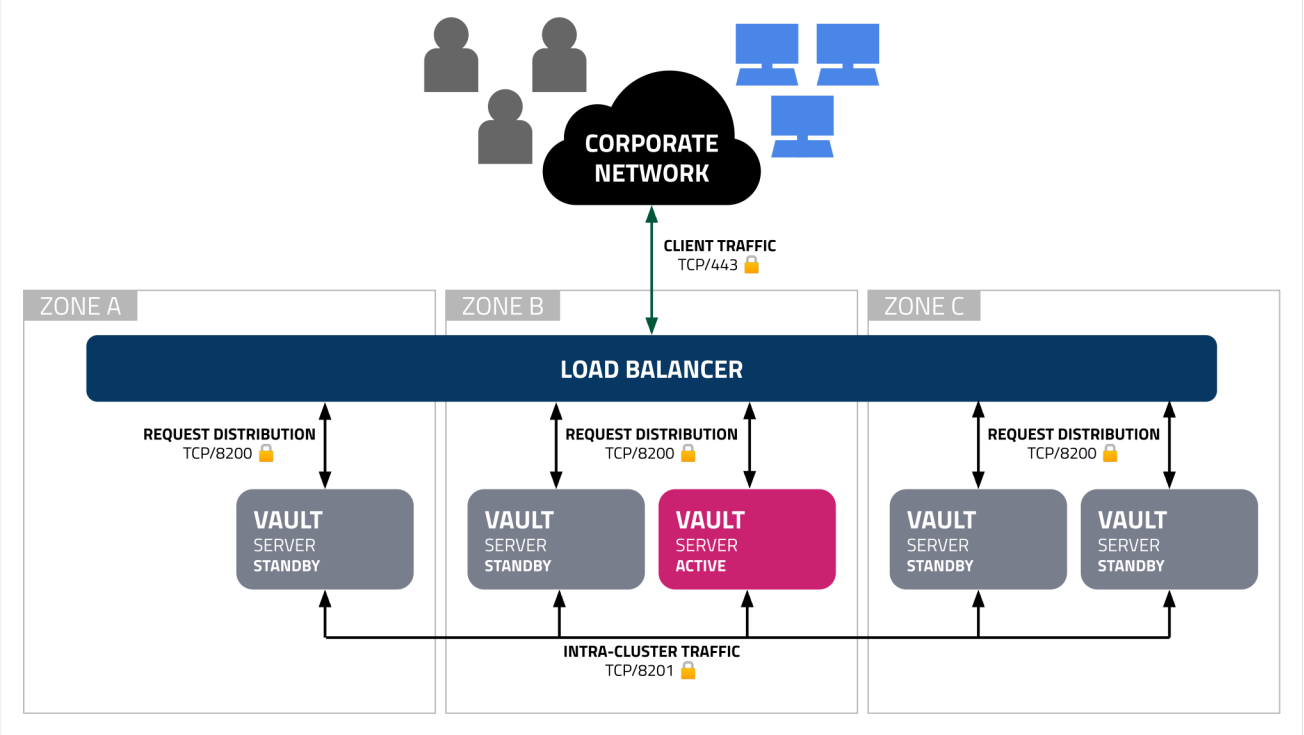
- `lb_certificate_arn`
- `leader_tls_servername`
- `secrets_manager_arn`
- `vpc_id`

Other variables are customized for this demo deployment:

- `instance_type` (set to `m5.large`, default is `m5.xlarge`)
- `region` (set to `us-west-2`, default is `us-east-1`)
- `resource_name_prefix` (set to `mara`)
- `vault_license_filepath` (path to `./license` subfolder)
- `vault_version` (set to `1.10.0`, default is `1.8.1`)

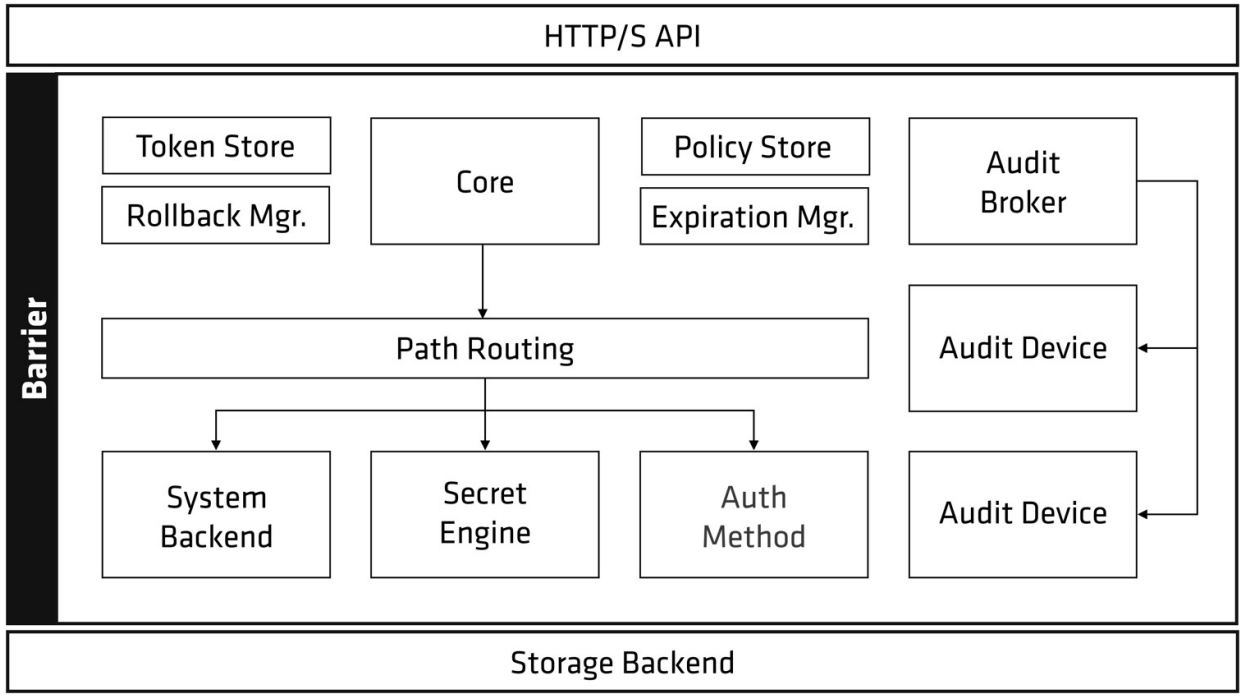
Vault Reference Architecture Design

- Cluster of 5 EC2 instances behind a load balancer (spread placement)
- Instances deployed across 3 AZs to mitigate single point of failure
- Storage integrated on Vault Nodes
- ELB (secured by SSL) checks health of nodes, triggers auto-scaling group to replace Vault nodes if needed
- Sizing: load test, but plan for critical I/O and memory requirements
- The more infrastructure you add, the more important network latency will be (keep it under 8ms)



Vault Internal Architecture

- Only HTTP API front-end and storage back-end are insecure
- System loads in sealed state – must be unsealed with root key or key shares
- Client requests include authN methods to prove identity, routed by policies (ACLs)
- Core answers client with token, to be used for subsequent authZ requests
- All requests and responses logged from core to audit broker -> audit devices



Demo Break

Initialize and Configure Vault

- Connect directly to a Vault node via Console/SSM
- Initialization unseals Vault for first time, creates root token and Shamir key shares
- Enable dead server cleanup, dead server health check interval, and quorum minimum for HA enablement
- Create admin policy and generate another root token for subsequent requests
- Enable the database secrets engine

Create Admin Policy Template

```
$ tee admin-policy.hcl <<EOF
# Mount secrets engines
path "sys/mounts/*" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}

# Configure the database secrets engine and create roles
path "database/*" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}

# Manage the leases
path "sys/leases/+/database/creds/readonly/*" {
  capabilities = [ "create", "read", "update", "delete", "list", "sudo" ]
}

path "sys/leases/+/database/creds/readonly" {
  capabilities = [ "create", "read", "update", "delete", "list", "sudo" ]
}

# Write ACL policies
path "sys/policies/acl/*" {
  capabilities = [ "create", "read", "update", "delete", "list" ]
}

# Manage tokens for verification
path "auth/token/create" {
  capabilities = [ "create", "read", "update", "delete", "list", "sudo" ]
}EOF
```

Demo Break

Integrate Vault with PostgresQL

- Connect to "OP-ASG" EC2 instance to create Postgres database roles for Vault
- Provision read-only role for Vault in application PostgresQL database
- Configure Vault to write to the Postgres database
- Configure HOW Vault is going to talk to our Postgres database with a creation statement

Create PostgresQL DB Role (read-only) for Vault

```
$ psql --username=tfPostgresql_op --host=database.server.com --
dbname=tfPostgresql -c "CREATE ROLE \"ro\" NOINHERIT;"

$ psql --username=tfPostgresql_op --host=database.server.com --
dbname=tfPostgresql -c "GRANT SELECT ON ALL TABLES IN SCHEMA public TO \"ro\";"
```

Configure Vault Database Plugin

```
$ vault write database/config/postgresql \
plugin_name=postgresql-database-plugin \
connection_url="postgresql://{{username}}:{{password}}@hostname:5432/postgres?ss
lmode=disable" \
allowed_roles=readonly \
username="root" \
password="rootpassword"
```

Define Dynamic Secrets Creation Statement

```
$ tee readonly.sql <<EOF
CREATE ROLE "{{name}}" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL
'{{expiration}}' INHERIT;
GRANT ro TO "{{name}}";EOF
```

Create Vault Database Secrets Engine Role

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

Demo Break

Test Database Dynamic Secrets

- Read a secret from the database engine and see it populate in Postgres, note TTL
- Switch to the dynamic secret for table operations
- Renew an active dynamic secret (for a specific duration)
- Revoke an active dynamic secret, watch an active secret expire

Common Vault requests for managing database dynamic secrets

```
# Read (Generate) new dynamic secret lease from database engine
$ vault read database/creds/readonly

# List active dynamic secrets leases
$ vault list sys/leases/lookup/database/creds/readonly

# Renew lease (TTL) for an active dynamic secret
$ vault lease renew database/creds/readonly/lease-identifier

# Renew lease (TTL) for custom interval (in seconds)
$ vault lease renew -increment=300 database/creds/readonly/lease-identifier

# Revoke lease for dynamic secret from database engine
$ vault lease revoke database/creds/readonly/lease-identifier

# Revoke all leases for the readonly path
$ vault lease revoke -prefix database/creds/readonly
```

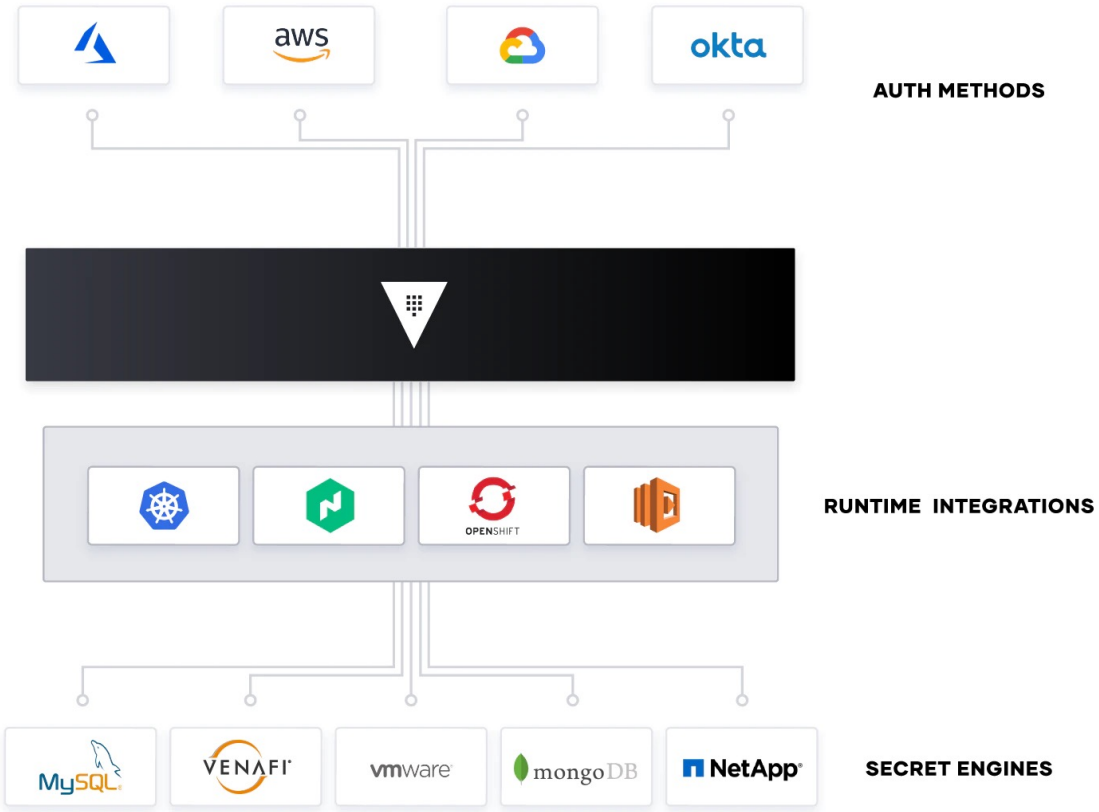
Validate Vault Dynamic Secrets are provisioned in PostgresQL

```
$ psql --username=vault-generated-username --host=database.server.com --
dbname=tfPostgresql -c "SELECT usename, valuntil FROM pg_user;"
```

Demo Break

Next Steps

- So much more to explore and ideate
- Additional dynamic and static secrets engines
- Key namespaces
- App integration with Vault Agent
- Integration with other HashiCorp solutions like Consul and Nomad



Questions?

Thank You

Appendix: Deploying Test Database

- Two separate configurations deploy network and database, EC2 instance
- New VPC with CIDR range unique to other networks (no overlap)
- 9 subnets deployed across 3 AZs in a region
- RDS PostgresQL launched in single AZ
- Launch template and ASG of 1 for EC2
- Bootstrap script pre-installs Postgres client on EC2 instance

Terraform Note
The `provision-postgres-singleaz` Terraform configuration will output 5 values when complete:
`db_instance_endpoint`, `db_instance_name`, `db_instance_password`, `db_instance_username`, and `vpc_id`.
Expose sensitive values with `terraform output`. Save these values for use later in the demo.

Terraform Note
The `deploy-operator-asg` Terraform configuration requires the Postgres VPC `vpc_id` value. Specify this ID in `variables.tf`.