HashiCorp Vault

Centralized Secrets Management

Network Security Course
Work Assignment

Author: Matteo Amagliani
Professor: José Luis Segura Lucas

January 27, 2026

Universidad de Castilla-La Mancha
Department of Computer Science

Abstract

In modern distributed systems, managing credentials and sensitive data presents a critical security challenge. Organizations face issues such as credential sprawl, lack of audit trails, static long-lived secrets, and difficulty implementing the principle of least privilege. HashiCorp Vault addresses these challenges by providing a centralized platform for secrets management, dynamic credential generation, encryption-as-a-service, and comprehensive access control.

This assignment explores Vault's architecture, core concepts, security model, and a simple practical implementation. Through hands-on demonstrations, we examine how Vault solves real-world problems in credential management, how it integrates with existing infrastructure, and the security benefits it provides. The demonstrations cover authentication methods, static and dynamic secrets, encryption services, and application integration patterns.

Key findings show that Vault significantly reduces security risks by eliminating hardcoded credentials, providing time-bound access, enabling automatic secret rotation, and maintaining complete audit trails. The system's architecture, based on encryption by default and strict authorization policies, makes it suitable for organizations requiring robust security and compliance capabilities.

Contents

1 Introduction

Modern software systems are increasingly distributed: applications are composed of microservices, run across multiple environments (local development, staging, production), and integrate external services (databases, cloud APIs, identity providers). This complexity amplifies a persistent security issue: ***credentials sprawl***. Secrets are often duplicated across repositories, CI variables, deployment manifests, and machine environments. These practices increase the attack surface and complicate incident response.

2 Credentials and Secrets Management

2.1 What is a secret?

A *secret* is any piece of sensitive information whose disclosure or misuse can lead to unauthorized access or loss of integrity. Typical examples include:

- API keys and tokens for third-party services
- Database usernames and passwords
- TLS private keys and certificates
- Encryption keys and signing keys
- Cloud credentials (access keys, service principals)

From a security engineering perspective, the main challenge is not only storing secrets, but also managing their ***lifecycle***: provisioning, distribution, rotation, revocation, and auditing.

2.2 Why credential management is difficult?

Credential management is difficult for both technical and organizational reasons:

- **Sprawl**: as mentioned before, credentials are copied into multiple systems.
- **Longevity**: static credentials persist for months or years, increasing the probability of exposure.
- **Weak accountability**: shared credentials prevent accurate attribution of actions.
- **Rotation costs**: manual rotations are disruptive and often delayed.
- **Least privilege violations**: credentials frequently grant excessive permissions.

2.3 Security requirements for a credential service

A well-designed credentials storage service should satisfy:

- **Confidentiality**: secrets should not be exposed at rest or in transit.
- **Integrity**: unauthorized modification must be prevented.
- **Availability**: authorized clients should reliably retrieve required secrets.
- **Least privilege**: access should be granted only to necessary paths and operations.
- **Auditability**: access and administrative actions should be logged.
- **Rotation and revocation**: secrets should be rotated and invalidated when needed.

Vault is positioned to address these requirements using an identity-based authorization model, secret engines, and audit devices [8].

3 HashiCorp Vault Overview

3.1 What is HashiCorp Vault?

HashiCorp Vault is a centralized platform for secrets management and privileged access workflows. It provides a single control plane to securely store, access, and manage sensitive data (e.g., passwords, API keys, encryption keys, and certificates) across on-premises, cloud, and hybrid environments [10].

At its core, Vault is built around a **plugin-based architecture** that defines how data flows through the system and how clients interact with protected resources [9]. Plugins act as modular building blocks and are mounted at specific paths, allowing multiple versions of the same plugin to coexist within a single Vault instance.

Vault supports several categories of plugins:

- **Authentication plugins**, which implement authentication workflows and allow clients to establish an identity with Vault.
- **Secrets plugins**, which function as secrets engines responsible for generating, storing, encrypting, and managing sensitive data.
- **Database plugins**, which provide dynamically generated credentials to access database systems.

Key idea

Vault models secrets access as a combination of identity and control mechanisms: every request must be authenticated via plugins, authorized through policies, and auditable, ensuring secure access to sensitive data.

3.2 Why use Vault?

Vault addresses secret sprawl by replacing scattered credentials with a centralized and auditable workflow. It can rotate long-lived credentials, issue short-lived credentials on demand, and enforce least privilege with deny-by-default policies [11, 16]. Vault can also provide "encryption as a service" so applications can encrypt/decrypt data without directly handling key material [26].

In practice, Vault enables:

1. managing static secrets (KV, Cubbyhole)
2. managing certificates (PKI, external CA integrations)
3. managing identities and authentication workflows (identity engine, OIDC provider)
4. managing third-party secrets (dynamic database and cloud credentials)
5. managing sensitive data (encrypt/tokenize without exposing keys)
6. supporting regulatory and compliance-driven security controls (audit, key custody, HSM/KMS)

3.2.1 Manage static secrets

Store and rotate arbitrary secrets in Vault with the Key/Value and Cubbyhole engines. Vault encrypts data before writing out to persistent storage, so accessing the raw storage backend is insufficient to reveal plaintext.

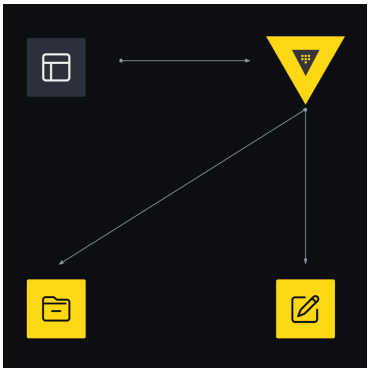

Figure 1: Static secrets management (conceptual).

Cubbyhole The Cubbyhole secrets engine is used to store arbitrary secrets namespaced to a token: paths are scoped per token, and no token can access another token's Cubbyhole. When the token expires, its Cubbyhole is destroyed [21].

KV secrets engine The KV secrets engine is a generic key-value store. It can run in two modes [20]:

- **KV v1 (non-versioned)**: stores only the most recent value for a key; updates overwrite previous values.
- **KV v2 (versioned)**: maintains a configurable number of versions (default commonly 10), supporting safer updates and controlled rollback.

3.2.2 Manage certificates

Vault can manage certificate lifecycles (e.g., PKI issuance, short-lived certs). This supports stronger authentication and reduces manual handling of private keys. While not the focus of the live demo, certificates are relevant because TLS and mutual TLS can be built into credential workflows.

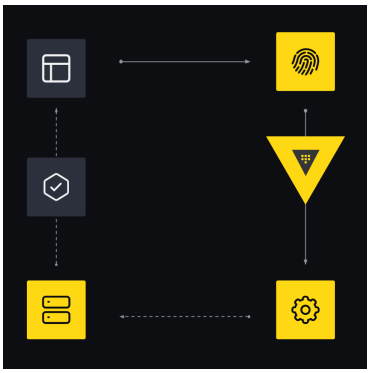

Figure 2: Certificate management (conceptual).

3.2.3 Manage identities and authentication

Vault manages identities and controls client access with entities/aliases, tokens, roles, and policies. It can also support OIDC workflows and act as an identity provider for some client scenarios [24, 25].

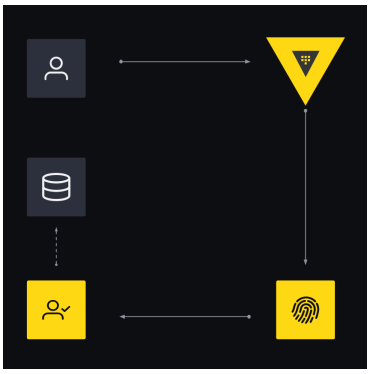

Figure 3: Identity and authentication (conceptual).

3.2.4 Manage third-party secrets (dynamic credentials)

Vault can generate and revoke on-demand credentials for databases and cloud providers. For databases, Vault can manage both static role rotation and dynamic user issuance. Dynamic credentials are one of Vault's strongest security contributions because they reduce the lifetime and reuse of credentials [22].

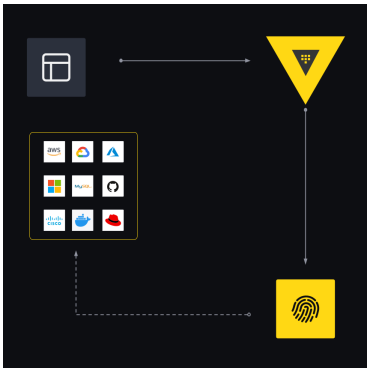

Figure 4: Third-party secrets (conceptual).

3.2.5 Manage sensitive data

Vault can offer encryption-as-a-service (Transit) so applications can encrypt/decrypt or sign/verify without embedding key material in application code. This is relevant when "secrets" include not only credentials, but also regulated data that must be protected [26].

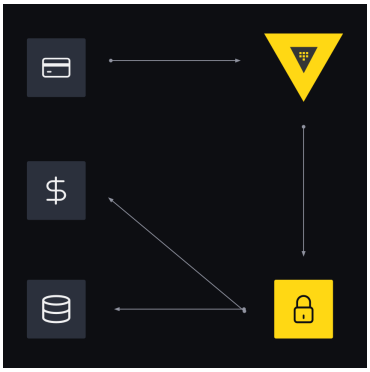

Figure 5: Secure sensitive data (conceptual).

3.2.6 Support regulatory compliance (high-level)

Vault's auditability, access control, and key custody patterns can help satisfy compliance objectives. Some compliance-focused features (e.g., HSM integration, certain FIPS constraints) may depend on deployment mode and edition, but the conceptual controls (audit, least privilege, key management discipline) remain relevant [12, 31].

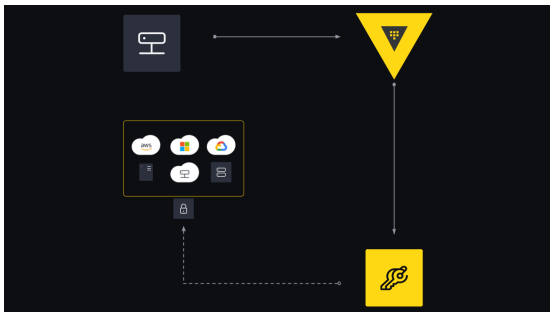

Figure 6: Support regulatory compliance (conceptual).

3.3 How Vault works: request-flow overview

Vault centralizes secret management, rotates credentials, generates credentials on demand, audits client interactions, and supports compliance workflows [11].

A simplified request flow:

1. **Authenticate**: the client supplies credentials; Vault validates identity via an auth method.
2. **Validate**: validates clients identity against trusted authorized third-party.
3. **Authorize**: evaluates policies attached to the resulting token.
4. **Access**: routes the request to the relevant secrets engine.
5. **Audit**: audit devices record request/response metadata.

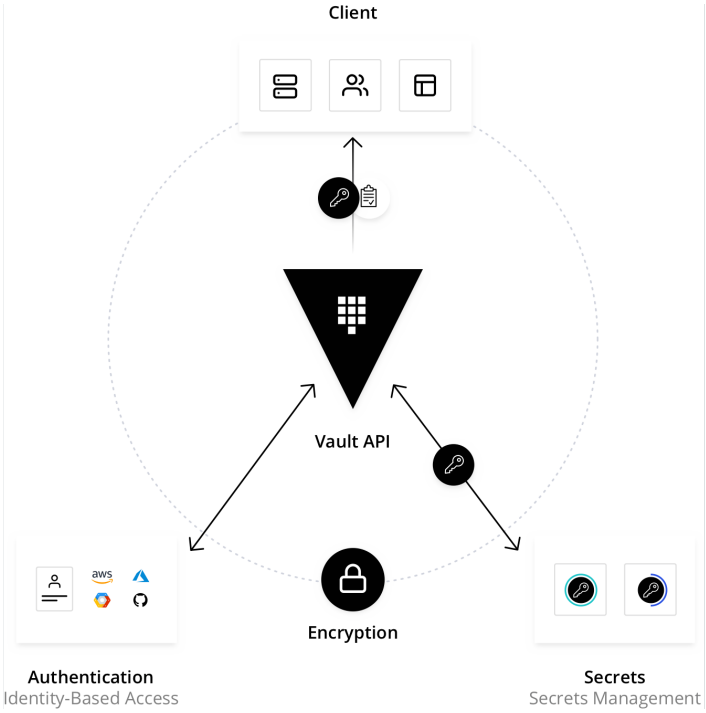

Figure 7: Vault request flow at a high level.

4 Core Concepts

4.1 Static vs dynamic secrets

Vault supports two fundamentally different secret models: *static* and *dynamic* secrets, which mainly differ in how they are created, shared, and revoked [32], [33].

Static secrets are long-lived credentials that are created externally and stored securely in Vault, typically using the key/value (KV) secrets engine. These secrets exist independently of client requests and remain valid until they are manually rotated or revoked. As a result, static secrets are often reused across multiple applications or services, increasing the risk associated with leakage or compromise over time.

Dynamic secrets in contrast, are generated by Vault on demand when a client requests access. They do not exist prior to the request and are issued with a limited lifetime defined by a time-to-live (TTL). Each request produces a unique set of credentials, which are automatically expired and revoked by Vault. This significantly reduces the attack surface by minimizing secret reuse and exposure time.

4.2 Authentication methods

Authentication in Vault establishes **identity**. Vault supports a wide range of auth methods intended for different operational contexts [17]. Representative examples include:

- **Human-oriented**: LDAP, OIDC, GitHub, userpass.
- **Machine-oriented**: AppRole, Kubernetes, cloud auth.

Auth methods are mounted at paths; multiple auth backends can coexist. A successful authentication results in a **token** associated with one or more policies.

Design principle

Authentication proves ***who*** the client is; authorization policies define *what* the client can do.

4.3 Authorization and policies

Vault access control is path-based and deny-by-default: access is disallowed unless explicitly granted by policy [16]. Policies specify capabilities (read, list, update, delete,...) on paths.

```
path "secret/data/myapp/*" {
  capabilities = ["read", "list"]
}
```

Listing 1: Example policy granting read-only access to application secrets

4.4 Tokens

Tokens are the core mechanism by which authenticated clients access Vault [14]. Tokens are associated with:

- One or more policies,
- TTL and renewal behavior,
- Metadata used for auditing and identity mapping.

Root tokens

Root tokens are tokens that have the root policy attached to them. Root tokens can do anything in Vault. Anything. In addition, they are the only type of token within Vault that can be set to never expire without any renewal needed.

4.5 Storage backends

Vault requires a physical storage backend to persist data. The storage backend is treated as untrusted because Vault encrypts data before writing it [12]. Storage choice impacts availability and operational complexity, especially for high availability deployments [18, 27].

4.6 Secrets engines

Secrets engines are Vault's core functionality: components that store, generate, or encrypt sensitive data. Each engine is mounted at a specific path and operates independently, allowing multiple engines of the same type to coexist with different configurations [19].

1. **Key-Value (KV) secrets engine**: The Key-Value (KV) secrets engine stores arbitrary key-value pairs such as API keys, passwords, or configuration data. Version 2 of the KV engine supports versioning, soft deletion, and secret metadata, making it suitable for safe updates and rollback in credential management workflows [20].
2. **Cubbyhole secrets engine**: The Cubbyhole secrets engine is a per-token, private storage space used to store sensitive data temporarily. Unlike other secrets engines, data stored in Cubbyhole is accessible only by the token that created it and is automatically deleted when the token expires. This engine is commonly used during authentication workflows to securely pass secrets between processes [21].
3. **Transit secrets engine**:The Transit secrets engine provides encryption-as-a-service. It allows applications to encrypt and decrypt data using Vault-managed keys without storing the data itself in Vault. This engine is commonly used to protect sensitive data in transit or at rest within external systems [26].

4. **Database secrets engine**:The Database secrets engine generates dynamic, short-lived credentials for databases such as PostgreSQL, MySQL, and MongoDB. Vault automatically creates and revokes database users, reducing the risk associated with long-lived static credentials and improving overall security [22].

5. **PKI secrets engine**:The Public Key Infrastructure (PKI) secrets engine manages X.509 certificates. It can act as a root or intermediate certificate authority (CA) and dynamically issue, renew, and revoke certificates, making it ideal for service-to-service authentication and TLS automation.

6. **Totp secrets engine**:The Time-based One-Time Password (TOTP) secrets engine generates one-time passwords compliant with RFC 6238. It is typically used to implement multi-factor authentication (MFA) for users or systems requiring an additional security layer.

7. **Identity: entities and aliases**:Vault's identity system allows mapping multiple authentication sources to a single logical identity. An *entity* represents a client in Vault; *aliases* represent accounts in specific auth backends. Identity metadata improves traceability because entity identifiers can be recorded in audit logs [24].

8. **OIDC provider (high-level)**:Vault can act as an OpenID Connect (OIDC) identity provider, allowing OIDC-capable applications to authenticate users via Vault identity and its supported auth methods [25]. In credential service designs, OIDC may unify user authentication across platforms while retaining centralized authorization.

5 Internal Architecture and How Vault Works

This chapter focuses on the internal mechanisms that underpin Vault's security properties: encryption by default, the seal/unseal boundary, policy enforcement, leasing, auditing, and high availability concepts.

5.1 Vault Internal Architecture and Component Interactions

HashiCorp Vault is designed around a layered architecture that *separates untrusted storage from trusted operations, enforces access control, and manages the lifecycle of secrets.* Figure 8 shows the main components and their relationships.

Vault's core encryption layer is often described as the **barrier**. Because the storage backend resides outside the barrier, it is treated as untrusted. Vault encrypts data before persisting it to storage; compromising the raw storage backend should not directly reveal plaintext secrets [12].

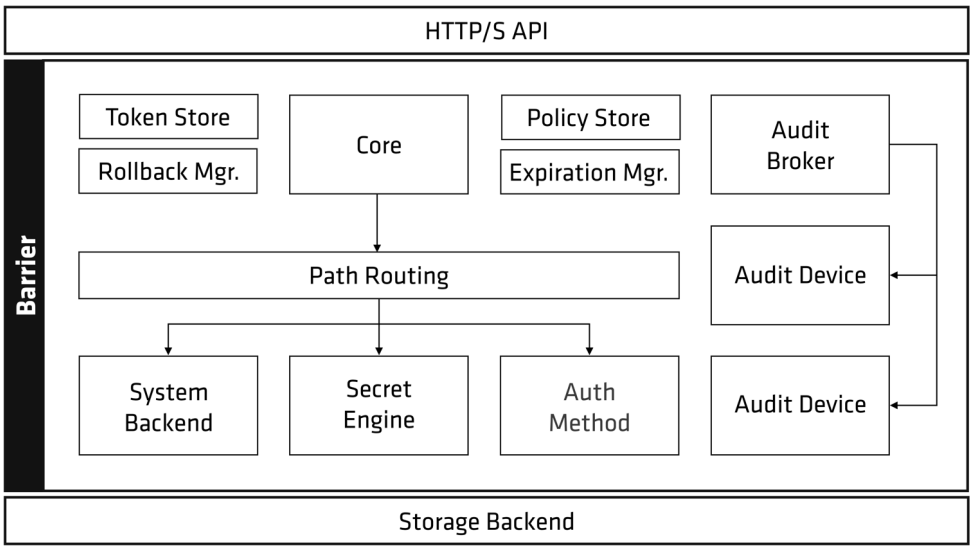

Figure 8: Vault high-level architecture and component interactions.

Top Layer **HTTP/S API** Vault exposes all functionality through a REST-like HTTPS API. This layer is the entry point for clients, CLI, UI, and agents. Requests are stateless and passed to the core for authorization and processing.

Barrier

- **Core**: The core orchestrates all internal operations: validating tokens, enforcing policies, routing requests, managing leases, and sending events to the audit system.
- **Barrier**: The barrier is Vault's encryption boundary. All data written to storage is encrypted; the storage backend is untrusted. The barrier protects secrets, tokens, policies, and configuration.
- **Token Store and Policy Store**: The token store maintains active tokens representing authenticated clients, including their TTLs and renewal policies. The policy store contains all ACL policies that define allowed actions.
- **Expiration Manager and Rollback Manager**: The expiration manager tracks leases for tokens and secrets, automatically revoking them when they expire. The rollback manager ensures consistency during storage operations, preventing partial failures from leaving the system in an invalid state.
- **Audit Broker and Audit Devices**: The audit broker receives events from the core and distributes them to configured audit devices. Audit logging provides traceability for all requests and responses, supporting compliance and incident response.
- **Path Routing**: Routing maps validated requests to the appropriate backend based on API paths. It directs requests to the system backend, secret engines, or authentication methods.

- **System Backend**: The system backend exposes administrative operations such as enabling auth methods, secrets engines, and audit devices, as well as managing policies and configuration.
- **Secret Engines**: Secrets engines handle storage or generation of secrets, including KV storage, dynamic database credentials, and cryptographic services. They enforce secret lifecycles, integrate with the expiration manager, and generate audit events.
- **Authentication Methods**: Auth methods validate client identity and issue tokens. Supported methods include username/password, AppRole, cloud IAM, and OIDC.

Bottom-level **Storage Backend**: The storage backend provides durable persistence for all Vault data. All writes pass through the barrier and are encrypted. The core, rollback manager, and expiration manager interact with storage to ensure consistency and proper lifecycle management.

To sum app the interactions: client requests enter via the API and flow through the core, which validates tokens and policies, routes to the appropriate backend, and manages leases and audit events. Secrets and tokens are encrypted by the barrier before reaching storage, while audit devices record all operations. This layered approach ensures secure, consistent, and auditable secret management.

5.2 Seal and unseal: operational security boundary

When a Vault server starts, it begins in a **sealed** state. In this state, Vault can access physical storage but cannot decrypt the data required to serve requests. Before any operational functionality is available, Vault must be **unsealed** [13].

Why sealing exists? Sealing ensures that on startup (or after a seal event), decryption capability is not present in memory until authorized operators explicitly provide the necessary material. This reduces the risk of secrets being exposed simply because the service is running.

5.2.1 Shamir seal and key shares

By default, Vault uses Shamir's Secret Sharing to split the unseal key into multiple key shares [29, 13]. A threshold number of shares is required to reconstruct the unseal capability.

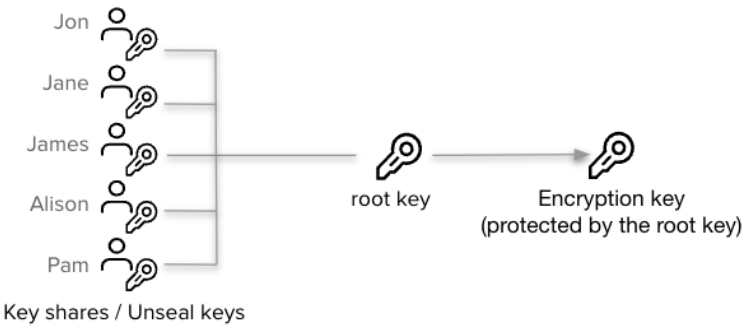

Figure 9: Overview of Shamir-based unseal.

5.2.2 Auto-unseal and recovery keys

Auto-unseal delegates unseal operations to a trusted external system (e.g., cloud KMS or HSM). This reduces manual key-share handling but introduces a dependency on the security and availability of the external key management service. When auto-unseal is used, some administrative operations may require *recovery keys* instead of unseal keys [13].

Trade-off

Auto-unseal simplifies operations but shifts trust to the external KMS/HSM.

5.3 Request processing and policy enforcement

After unsealing, Vault loads configured audit devices, auth methods, and secrets engines. Requests flow through Vault core, which:

- verifies token validity,
- loads relevant policies,
- evaluates authorization (deny-by-default),
- routes the request to the relevant backend,
- emits audit events.

5.4 Token lifecycle and leases

Token lifetime and secret leases are critical to Vault's risk reduction model:

- Tokens may have TTLs and renewal policies [14].
- Secrets issued by engines (e.g., database credentials) are typically returned with a lease ID and TTL [15].

- Vault can revoke secrets automatically upon lease expiration, enabling time-bounded access and automated cleanup [22].

5.5 Audit logging and accountability

Audit devices record security-relevant information about requests and responses. Auditing supports accountability, compliance evidence, and incident response by providing traceability for access and administrative actions [23].

6 Designing a Credentials Storage Service with Vault

This chapter translates Vault primitives into a coherent service design. The objective is to define an operational workflow satisfying least privilege, auditability, and secret lifecycle control.

6.1 Service model and roles

A credential service must separate responsibilities:

- **Operators (security/infra)** configure auth methods, policies, secret engines, auditing, and perform initialization.
- **Application identities** retrieve only required secrets at runtime; they should not have write privileges to shared secret namespaces.
- **Auditors** review audit outputs and verify policy compliance.

6.2 Requirements mapping

The following mapping connects credential-service requirements (Chapter 2) to Vault controls.

Requirement	Vault control
Confidentiality	Barrier encryption; TLS; access via tokens and policies
Least privilege	Deny-by-default policies; path capabilities
Auditability	Audit devices capturing request metadata
Rotation/revocation	Dynamic secrets (leases); revocation; scheduled rotation for static creds
Accountability	Identity entities/aliases; per-app tokens and roles

Table 1: Mapping credential service requirements to Vault controls.

6.3 Static credentials pattern (KV v2)

Static secrets (e.g., an API key) are stored under a KV v2 mount [20]. Recommended:

1. store secrets under clear namespaces (e.g., `secret/myapp/`)
2. grant application identities `read` and optionally `list` only
3. restrict write/update/delete to operator identities
4. use versioning to support safe updates and controlled rollback

6.4 Dynamic credentials pattern (database engine)

Dynamic database credentials reduce the risk of long-lived shared DB passwords [22]. Recommended:

1. configure Vault with a database connection using privileged credentials stored securely
2. define Vault roles specifying creation statements and granted privileges
3. applications request credentials at runtime; Vault returns unique username/password pairs with TTL
4. credentials are revoked automatically when leases expire, or manually during incident response

6.5 Threat model and mitigations

A simplified threat analysis for credential services:

- **Hardcoded credentials in code**: mitigated by centralized retrieval at runtime.
- **Credential leakage in logs/config**: mitigated by short-lived dynamic secrets and least privilege.
- **Unauthorized access attempts**: mitigated by deny-by-default policies and audit logs.
- **Storage compromise**: mitigated by barrier encryption and seal boundary [12].

Security intuition

Vault shifts the problem from distributing long-lived secrets to controlling access to a protected service that can issue time-bounded credentials and record who did what.

7 Practical Demonstration Plan

This chapter defines some steps of my live demonstration. The demo is intentionally small but realistic: it shows Vault's core security boundaries (seal/unseal, least privilege, short-lived access, auditability) using only Docker, the Vault CLI and Vault UI. Detailed commands are provided in Appendix A.

7.1 Demo goals

The demonstration will show:

- **Static secrets** stored with KV v2 (versioning and safe updates).
- **Least-privilege access control** enforced by policies (allowed vs denied).
- **Non-root access** using a restricted identity (`userpass`, etc.).
- **Time-bounded access** via token TTL and revocation.
- **Audit evidence** looking at many req. /res. logs.
- **(Not yet completed)** dynamic database credentials with TTL + lease revocation.

7.2 Environment and assumptions

The demo runs locally on Windows using Docker:

- Vault in **dev and server mode** with file storage (persistent data) and explicit **init/unseal**.
- A bind-mounted log directory for **audit logs**.
- (Optional) a PostgreSQL container to demonstrate the Database secrets engine.

8 Discussion and Conclusions

This assignment evaluated HashiCorp Vault as a credentials storage service, combining architecture and access-control concepts with a reproducible demonstration. Vault addresses a common security failure mode sprawling, long-lived credentials, by centralizing control and enforcing authorization at request time.

From a credential-management perspective, the most relevant benefits are:

- **Clear authorization boundaries** through policies and deny-by-default behavior.
- **Auditability** of administrative and secret access operations.
- **Reduced blast radius** via token TTLs, revocation, and secret leases (when using dynamic engines).
- **Safer secret handling** through KV v2 versioning and controlled lifecycle operations.

These security gains come with **operational responsibilities**:

- secure handling of init/unseal materials and a disciplined unseal process;
- careful policy design and review to prevent privilege creep;
- protection and monitoring of audit logs as sensitive security telemetry.

Overall, Vault provides a practical security model for modern systems: authentication establishes identity, policies enforce least privilege, and secret engines enable either versioned static secrets or short-lived dynamic credentials, with auditing as the accountability layer.

A Quick Start and CLI (Windows PowerShell)

This appendix collects the minimal commands required to reproduce the lab on Windows using PowerShell, Docker, and the Vault CLI.

My HashiCorp Vault - GitHub Repository

A.1 Baseline environment

Set the server address:

```
$env:VAULT_ADDR = "http://127.0.0.1:8200"
```

Listing 2: PowerShell: set VAULT_ADDR (local lab)

A.2 Dev mode

Dev mode is useful for learning the CLI, but is not the best "realistic" demo because it is in-memory and auto-unsealed.

```
docker compose -f docker\docker-compose.dev.yml up -d
vault login dev-only-token
vault status
```

Listing 3: Dev mode stack (Docker)

Security note (dev mode)

Dev mode is insecure and intended only for local practice.

A.3 Server mode (demo baseline)

A.3.1 Start the stack

```
docker compose -f docker\docker-compose.server.yml up -d
vault status
```

Listing 4: Server mode stack (Docker)

A.3.2 Initialize and unseal (first run)

```
$init = vault operator init -key-shares=1 -key-threshold=1 -format=json
    | ConvertFrom-Json
$unsealKey = $init.unseal_keys_b64[0]
$rootToken = $init.root_token

vault operator unseal $unsealKey
```

```
vault login $rootToken
vault status
```

Listing 5: Initialize + capture unseal key and root token (PowerShell)

Operational note

After a container restart, Vault may return sealed again and require `vault operator unseal` with the same unseal key.

A.4 Credentials storage service: setup and tests

This section operationalizes Vault as a credentials storage service. The operator configures audit, secrets engines, and policies; applications authenticate and retrieve only what they need under least privilege.

Operator setup: audit + KV v2

```
vault audit enable file file_path=/vault/logs/vault-audit.log
vault audit list
```

Listing 6: Enable audit logging to a bind-mounted file

```
vault secrets enable -path=app-secrets -version=2 kv
vault secrets list
```

Listing 7: Enable KV v2 for application credentials

T1 — Store and read a static secret

```
vault kv put -mount=app-secrets payments/stripe api_key="
    sk_test_REPLACE_ME" account="demo-account"
vault kv get -mount=app-secrets payments/stripe
```

Listing 8: T1: write and read (KV v2)

T2 — Versioning and rollback intuition (KV v2)

```
vault kv put -mount=app-secrets payments/stripe api_key="
    sk_test_NEW_VALUE"
vault kv metadata get -mount=app-secrets payments/stripe
vault kv get -mount=app-secrets -version=1 payments/stripe
```

Listing 9: T2: update + inspect metadata + read an older version

T3 — Least privilege: allowed vs denied

Write a policy for a single credential path (KV v2 requires both `/data/` and `/metadata/` paths):

```
vault policy write app-cred-policy - << EOF
path "app-secrets/data/payments/stripe" {
  capabilities = ["read", "create", "update"]
}
path "app-secrets/metadata/payments/stripe" {
  capabilities = ["read"]
}
EOF
```

Listing 10: T3: least-privilege policy (KV v2)

Bind the policy to a restricted identity and test:

```
vault auth enable userpass
vault write auth/userpass/users/payments-app password="Matteo_secret_psw
    " policies="app-cred-policy"

vault login -method=userpass username=payments-app
vault kv get -mount=app-secrets payments/stripe
vault kv get -mount=app-secrets payments/other-service  # expected:
    denied
```

Listing 11: T3: create restricted user and test access

T4 — Token lifecycle (TTL, renew, revoke)

```
vault login $rootToken

$tok = vault token create -ttl=2m -policy=app-cred-policy -format=json |
     ConvertFrom-Json
$short = $tok.auth.client_token

vault token lookup $short
vault token renew  $short
vault token revoke $short
```

Listing 12: T4: short-lived token (PowerShell)

Expected outcome

After revocation, the token stops working immediately, demonstrating time-bounded access and incident-response capability.

T5 — (Work in progress...) Dynamic DB credentials (PostgreSQL)

This optional extension requires the server+DB stack and demonstrates on-demand credentials with TTL + lease revocation.

```
docker compose -f docker\docker-compose.server-db.yml up -d
```

Listing 13: T5: start server+db stack

```
vault secrets enable database

vault write database/config/postgres ‘
  plugin_name=postgresql-database-plugin ‘
  allowed_roles=readonly ‘
  connection_url="postgresql://{{username}}:{{password}}@vault-postgres
   :5432/demo?sslmode=disable" ‘
  username="vaultadmin" ‘
  password="vaultadminpw"
```

Listing 14: T5: enable and configure database secrets engine (example)

```
$creds = vault read -format=json database/creds/readonly | ConvertFrom-
   Json
$lease = $creds.lease_id

vault lease revoke $lease
```

Listing 15: T5: issue credentials + revoke lease (PowerShell)

T6 — Audit evidence

Audit logs are stored in the bind-mounted file.

```
Get-Content -Tail 30 .\docker\vault\logs\vault-audit.log
```

Listing 16: T6: read latest audit lines from host (PowerShell)

Audit log sensitivity

Audit logs contain sensitive metadata (paths, identities, access patterns). Protect them like security logs: restricted access, integrity monitoring, and secure retention.

B Policy Examples

Policies are deny-by-default: access must be explicitly granted [16].

B.1 Read-only access to an application namespace (KV v2)

```
path "app-secrets/data/myapp/*" {
  capabilities = ["read", "list"]
}
path "app-secrets/metadata/myapp/*" {
  capabilities = ["read", "list"]
}
```

Listing 17: Read-only policy for app secrets (KV v2)

B.2 Database dynamic credentials access (example)

```
path "database/creds/readonly" {
  capabilities = ["read"]
}
```

Listing 18: Policy to read dynamic DB credentials

C Audit Log Considerations

Audit logs provide traceability of Vault operations and are critical for compliance and incident response [23].

What audit logs typically contain?

- request path and operation type;
- authentication metadata (token accessor; entity ID when identity is enabled);
- request ID and timestamps;
- response status and duration.

Security considerations

- protect audit logs at rest and in transit;
- restrict access to operators/auditors;
- integrate monitoring/alerting for suspicious patterns.

Operational guidance

Enable auditing early (ideally before onboarding applications). Without logs, post-incident analysis is severely limited.

References

[1] HashiCorp. *Install the Vault binary (tutorial)*. Available at: https://developer.hashicorp.com/vault/tutorials/get-started/install-binary

[2] HashiCorp. *Manually install a Vault binary*. Available at: https://developer.hashicorp.com/vault/docs/get-vault/install-binary

[3] HashiCorp. *Operations quick start*. Available at: https://developer.hashicorp.com/vault/docs/get-started/operations-qs

[4] HashiCorp. *Learn to use the Vault CLI*. Available at: https://developer.hashicorp.com/vault/tutorials/get-started/learn-cli

[5] HashiCorp. *status - Vault CLI command*. Available at: https://developer.hashicorp.com/vault/docs/commands/status

[6] HashiCorp. *Learn CLI: status exit codes (sealed/unsealed)*. Available at: https://developer.hashicorp.com/vault/tutorials/get-started/learn-cli

[7] HashiCorp. *Vault configuration parameters*. Available at: https://developer.hashicorp.com/vault/docs/configuration

[8] HashiCorp. *Vault Documentation*. Available at: https://developer.hashicorp.com/vault/docs/internals/security

[9] HashiCorp. *Vault Documentation*. Available at: https://developer.hashicorp.com/vault/docs/plugins

[10] HashiCorp. *What is Vault?* Available at: https://developer.hashicorp.com/vault/docs/about-vault/what-is-vault

[11] HashiCorp. *How Vault Works*. Available at: https://developer.hashicorp.com/vault/docs/about-vault/how-vault-works

[12] HashiCorp. *Vault Security Model (Internals)*. Available at: https://developer.hashicorp.com/vault/docs/internals/security

[13] HashiCorp. *Seal/Unseal Concepts*. Available at: https://developer.hashicorp.com/vault/docs/concepts/seal

[14] HashiCorp. *Tokens Concepts*. Available at: https://developer.hashicorp.com/vault/docs/concepts/tokens

[15] HashiCorp. *Leases and Renewable Secrets (Concepts)*. Available at: https://developer.hashicorp.com/vault/docs/concepts/lease

[16] HashiCorp. *Policies (Concepts)*. Available at: https://developer.hashicorp.com/vault/docs/concepts/policies

[17] HashiCorp. *Authentication Methods*. Available at: https://developer.hashicorp.com/vault/docs/auth

[18] HashiCorp. *Storage Backends*. Available at: https://developer.hashicorp.com/vault/docs/configuration/storage

[19] HashiCorp. *Secrets Engines*. Available at: https://developer.hashicorp.com/vault/docs/secrets

[20] HashiCorp. *KV Secrets Engine*. Available at: https://developer.hashicorp.com/vault/docs/secrets/kv

[21] HashiCorp. *Cubbyhole Secrets Engine*. Available at: https://developer.hashicorp.com/vault/docs/secrets/cubbyhole

[22] HashiCorp. *Database Secrets Engine*. Available at: https://developer.hashicorp.com/vault/docs/secrets/databases

[23] HashiCorp. *Audit Devices*. Available at: https://developer.hashicorp.com/vault/docs/audit

[24] HashiCorp. *Identity (Concepts)*. Available at: https://developer.hashicorp.com/vault/docs/concepts/identity

[25] HashiCorp. *OIDC Identity Provider*. Available at: https://developer.hashicorp.com/vault/docs/secrets/identity/oidc-provider

[26] HashiCorp. *Transit Secrets Engine*. Available at: https://developer.hashicorp.com/vault/docs/secrets/transit

[27] HashiCorp. *High Availability (Concepts)*. Available at: https://developer.hashicorp.com/vault/docs/concepts/ha

[28] HashiCorp. *Integrated Storage (Raft)*. Available at: https://developer.hashicorp.com/vault/docs/configuration/storage/raft

[29] Shamir, A. (1979). *How to Share a Secret*. Communications of the ACM, 22(11), 612–613.

[30] Verizon. *2024 Data Breach Investigations Report*. Available at: https://www.verizon.com/business/resources/reports/dbir/

[31] NIST. *SP 800-57 Part 1 Rev. 5: Recommendation for Key Management*. Available at: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

[32] HashiCorp. *Dynamic and static secrets in CI/CD*. Available at: https://developer.hashicorp.com/well-architected-framework/secure-systems/secure-applications/ci-cd-secrets/dynamic-and-static-secrets

[33] HashiCorp. *Understand static and dynamic secrets* . Available at: https://developer.hashicorp.com/vault/tutorials/get-started/understand-static-dynamic-secrets