Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020, 7(2):73-76

Research Article ISSN: 2394 - 658X

Secure Secret Management in DevOps CI/CD Pipelines

Kamalakar Reddy Ponaka

kamalakar.ponaka@gmail.com

ABSTRACT

As organizations increasingly adopt DevOps practices, the security of sensitive data such as secrets, API keys, and credentials becomes a significant concern. Continuous Integration and Continuous Deployment (CI/CD) pipelines automate software development but also require robust secret management. This paper presents an approach for securely managing secrets using JWT authentication between GitLab CI and HashiCorp Vault, providing a scalable, fine-grained, and secure solution for modern DevOps environments.

Keywords: JWT authentication, GitLab CI, HashiCorp Vault, secret management, CI/CD Security

INTRODUCTION

With the widespread adoption of DevOps and CI/CD pipelines, organizations face increasing challenges related to the secure management of secrets. Secrets such as API keys, database credentials, and cloud service access tokens are essential for CI/CD processes but must be handled securely to prevent unauthorized access and leaks. Insecure handling of secrets can result in severe security breaches, leading to data loss, unauthorized access, and system vulnerabilities.

Traditional approaches such as hard-coding secrets in source code or passing them via plaintext environment variables expose organizations to substantial risk. To address this, JWT authentication between GitLab CI and HashiCorp Vault offers a secure and scalable method for managing secrets, providing dynamic and temporary access to sensitive information during the pipeline execution.

THE CHALLENGE OF SECRET MANAGEMENT

Managing secrets in a CI/CD pipeline presents several challenges:

a) Hard-coded secrets: Embedding sensitive data in source code leads to potential exposure, especially in public or shared repositories.

b) Insecure storage and transmission: Secrets may be mishandled if stored insecurely or transmitted in plaintext, leading to unauthorized access.

c) Lack of auditing: Without centralized secret management, tracking secret usage and access is difficult, increasing the risk of undetected leaks.

d) Complex secret rotation: Manually rotating secrets across environments can lead to configuration errors or delays in updates.

e) Issue with Environment Variables: While using environment variables to manage secrets in CI/CD pipelines is a common approach, it also introduces significant risks if not handled properly. Below are some of the bad practices associated with storing secrets as CI environment variables:

A. Storing Secrets in Plaintext Environment Variables

One of the most common mistakes is storing sensitive information such as API keys, passwords, or tokens as plaintext environment variables. This practice exposes secrets to several risks:

• **Accidental Exposure**: If environment variables are not properly masked, they can be exposed in CI logs, which are often publicly accessible or shared among teams.

• **Ease of Extraction:** Plaintext environment variables can be easily extracted by any script or command running in the CI pipeline, increasing the attack surface if an attacker gains access to the CI environment.

• **Lack of Encryption:** Storing secrets as plaintext means they are not encrypted at rest or in transit, making them vulnerable to interception or unauthorized access.

B. Hard-Coding Secrets in Pipeline Definitions

Hard-coding secrets directly in the CI configuration file (.gitlab-ci.yml or similar) is a dangerous practice. This can lead to:

• **Version Control Exposure:** When CI configuration files are committed to version control systems (VCS), any hard-coded secrets are stored along with the code, potentially exposing them to anyone with access to the repository.

• **Difficulty in Secret Rotation:** Hard-coded secrets make it challenging to rotate credentials or keys without modifying the codebase and re-deploying, which can lead to delays and potential security lapses.

C. Insufficient Access Controls on Environment Variables

Improperly configured access controls can result in unauthorized access to secrets stored in environment variables:

• **Over-Privileged Access:** Giving all CI jobs or users access to all environment variables, regardless of necessity, violates the principle of least privilege and increases the risk of exposure.

• **Lack of Protected Variables:** CI platforms often provide mechanisms to protect sensitive environment variables, ensuring they are only accessible in specific branches or by authorized users. Failing to utilize these protections can lead to unintended exposure.

D. Environment Variable Sprawl

Storing too many secrets as environment variables leads to "environment variable sprawl," where the management and auditing of secrets become difficult:

• **Increased Complexity:** Managing numerous environment variables across multiple CI jobs or pipelines becomes complex, leading to configuration errors or inconsistent application of security policies.

• **Audit and Monitoring Challenges:** Tracking access and usage of secrets becomes more difficult as the number of environment variables grows, making it harder to detect unauthorized access or misuse.

E. Lack of Rotation and Expiry Mechanisms

Secrets stored as environment variables are often static, meaning they do not change unless manually updated. This practice introduces several security risks:

• **Long-Term Exposure:** Static secrets are more likely to be compromised over time, especially if they are not regularly rotated.

• **No Expiry:** Without automated expiry or rotation mechanisms, secrets remain valid indefinitely, even if the associated job, user, or application no longer requires them, increasing the risk of unauthorized use.

F. Storing Secrets in Shared or Global Environment Variables

Using shared or global environment variables across multiple pipelines or jobs introduces significant risks:

• **Unnecessary Exposure:** Secrets are accessible to all jobs, even those that do not require them, leading to potential misuse or accidental exposure.

• **Cross-Project Leakage:** If the same environment variables are used across different projects, a breach in one project could potentially expose secrets across multiple projects.

These challenges necessitate a solution that allows centralized, secure, and dynamic management of secrets.

JWT AUTHENTICATION OVERVIEW

JSON Web Token (JWT) is a widely adopted standard for securely transmitting information between parties as a JSON object. JWTs are signed to ensure their integrity and can be used to carry claims such as identity and permissions, making them suitable for authentication and authorization.

In a CI/CD context, GitLab CI generates JWT tokens for each job. These tokens can be validated by HashiCorp Vault to authenticate the pipeline and grant temporary access to secrets. By using JWT authentication, sensitive credentials do not need to be hard-coded or stored as long-lived tokens, improving overall security.

SOLUTION ARCHITECTURE

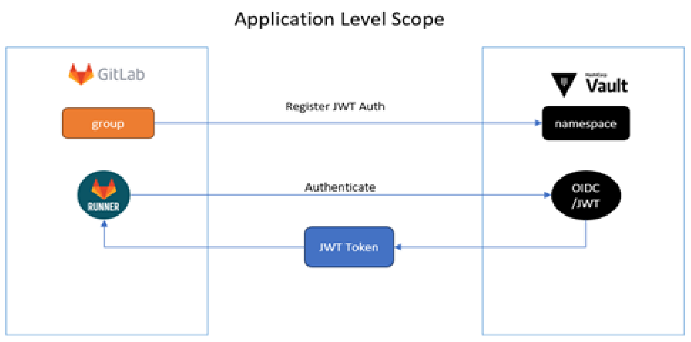
A. GitLab CI Overview

GitLab CI provides a comprehensive platform for automating the build, test, and deployment phases of application development. It offers robust secret management features and can integrate with external secret stores such as HashiCorp Vault.

B. HashiCorp Vault Overview

HashiCorp Vault is a widely used secret management solution that securely stores and manages access to sensitive data. Vault encrypts secrets at rest and provides fine-grained access control through its policy engine. Vault also supports dynamic secrets, which are short-lived and can be automatically revoked after use.

C. JWT Authentication in CI/CD Pipelines

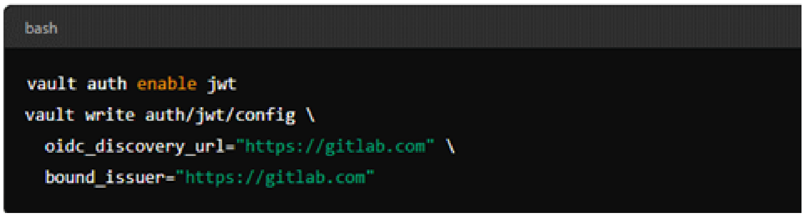


The integration of GitLab CI and HashiCorp Vault using JWT authentication consists of the following steps:
a) Job JWT Token Generation: GitLab CI automatically generates a JWT token for each job, which can be used for authentication with Vault.
b) Vault JWT Authentication: The GitLab CI job presents the JWT token to HashiCorp Vault for validation.
c) Vault Policy Enforcement: Vault applies the appropriate policies to the job based on the claims contained in the JWT token.
d) Secret Access: After successful authentication, Vault issues a short-lived token that the CI job can use to access the necessary secrets.

D. Integration Steps

a) Configuring Vault for JWT Authentication

To enable JWT authentication in Vault, the JWT authentication method must first be enabled, and GitLab's OIDC discovery URL should be configured.

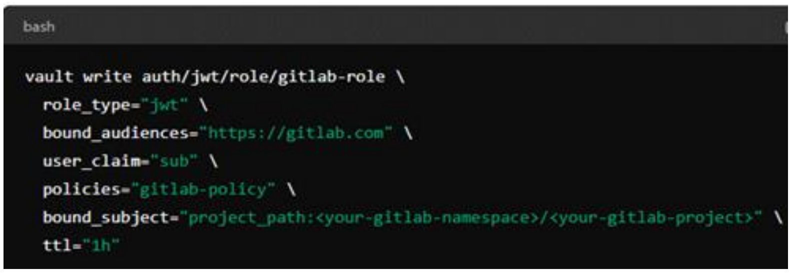

bash

```bash
vault auth enable jwt
vault write auth/jwt/config \
  oidc_discovery_url="https://gitlab.com" \
  bound_issuer="https://gitlab.com"
```

This step allows Vault to verify JWT tokens issued by GitLab.

b) Creating a Vault Role for GitLab CI

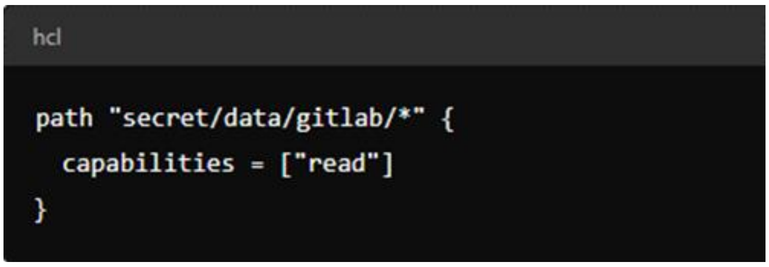
A Vault role is required to define which secrets a GitLab CI job can access based on its JWT claims.


bash

```bash
vault write auth/jwt/role/gitlab-role \
  role_type="jwt" \
  bound_audiences="https://gitlab.com" \
  user_claim="sub" \
  policies="gitlab-policy" \
  bound_subject="project_path:<your-gitlab-namespace>/<your-gitlab-project>" \
  ttl="1h"
```

c) Defining a Vault Policy

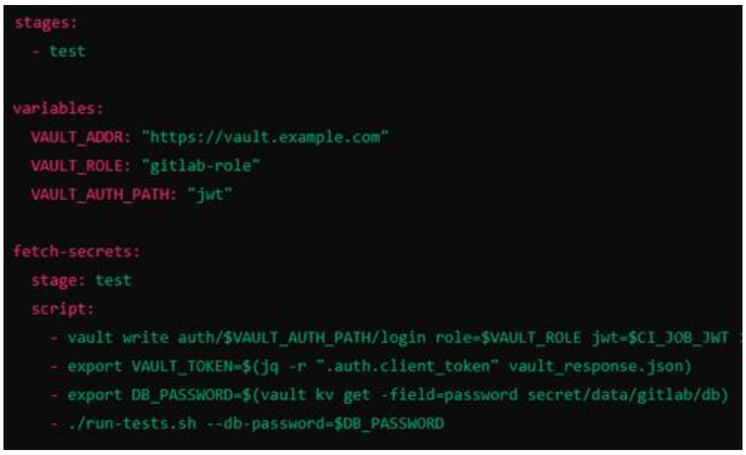
Vault policies control access to secrets. For example, a policy can restrict access to a specific set of secrets:


hcl

```hcl
path "secret/data/gitlab/*" {
  capabilities = ["read"]
}
```

d) Modifying GitLab CI Pipeline

In GitLab CI, the .gitlab-ci.yml file can be configured to retrieve secrets from Vault using the JWT token:



```
stages:
  - test

variables:
  VAULT_ADDR: "https://vault.example.com"
  VAULT_ROLE: "gitlab-role"
  VAULT_AUTH_PATH: "jwt"

fetch-secrets:
  stage: test
  script:
    - vault write auth/$VAULT_AUTH_PATH/login role=$VAULT_ROLE jwt=$CI_JOB_JWT
    - export VAULT_TOKEN=$(jq -r ".auth.client_token" vault_response.json)
    - export DB_PASSWORD=$(vault kv get -field=password secret/data/gitlab/db)
    - ./run-tests.sh --db-password=$DB_PASSWORD
```

e) Security and Best Practices

• **Role-Based Access Control (RBAC):** Vault's RBAC capabilities should be used to ensure that each CI job has access only to the secrets it needs.
• **Secret Masking:** GitLab’s secret masking functionality should be enabled to prevent secrets from being exposed in pipeline logs.
• **Audit Logging:** Vault's audit logs can track secret access, providing visibility into which pipelines accessed specific secrets.
• **Token Expiration:** The Vault tokens issued to GitLab CI jobs are short-lived, expiring shortly after the job completes, reducing the window of potential exposure.

CONCLUSION

JWT authentication between GitLab CI and HashiCorp Vault offers a secure, scalable solution for managing secrets in CI/CD pipelines. By leveraging temporary tokens and fine-grained access controls, organizations can reduce the risks associated with secret management and ensure that sensitive data is protected throughout the software development lifecycle.

This approach centralizes secret management, enhances security, and provides a scalable method for securely handling secrets in dynamic, automated environments.

REFERENCES

[1]. GitLab CI Documentation. Available: https://docs.gitlab.com/ee/ci/
[2]. HashiCorp Vault Documentation. Available: https://www.vaultproject.io/docs
[3]. JWT Introduction. Available: https://jwt.io/introduction/