PUBLIC DIFFUSION

Almond

Evaluation Technical Report

Vault version 1.21.0

2026/02/23
Version 1.00
ETR-PUBLIC-Vault-1.00
Internal Code: AGR003-40

Almond
7 avenue de la Cristallerie
92310 Sèvres
SAS au capital de 17 977 770,00 €
SIREN 841 059 553
TVA FR24 841 059 553

EVOLUTION SHEET

Version	Description	Role	Name	Date
1.00	First public release, based on internal report (ETR-Vault-1.00)	**Author**	ALMOND	2025/10/22
		Reviewer	ALMOND ANSSI	2026/02/12
		Approver	ANSSI	2026/02/23

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Document purpose

This document is produced as part of the evaluation of the project **Vault** in version **1.21.0**, developed by **HashiCorp**. It constitutes the Evaluation Technical Report (ETR) presenting the result of the evaluation work conducted by the **ALMOND** ITSEF.

This audit has been funded by French cybersecurity agency (ANSSI) as part of its efforts to support the security assessment of open-source software (more info at https://cyber.gouv.fr/open-source-lanssi). This audit aim was to assess the security level of Vault, focusing on code source analysis and dynamic penetration testing (configuration handling, authentication and ACL, secure storage of secrets, API robustness).

This document is a synthesized version of the initial report and is subject to technical and quality control by **ALMOND**. Updates to this document are made by the **ALMOND** project team.

1.2 Technical report identification

Table 1 – ETR Identification

Evaluation project name	Vault
Document reference	PUBLIC-Vault-1.00
Based on	ETR-PUBLIC-Vault-1.00
Evaluation date	2025/10/22 to 2025/mm/dd
Workload	60 man days

1.3 Glossary

Table 2 – Glossary

Acronym	Definition
ANSSI	*Agence Nationale de la Sécurité des Systèmes d'Information*
ITSEF	*IT Security Evaluation Facility*
NA	*Not Applicable*
TOE	*Target Of Evaluation*
CA	*Certification Authority*
ACME	*Automated Certificate Management Environment*
CN	*Common Name*
CRL	*Certificate Revocation List*
CSR	*Certificate Signing Request*
HSM	*Hardware Security Module*
MitM	*Man in the Middle*
OCSP	*Online Certificate Status Protocol*
PKI	*Public Key Infrastructure*
RA	*Registration Authority*
(m)TLS	*(mutual) Transport Layer Security*
UI	*User Interface*
VA	*Validation Authority*
PQC/PQ	*Post-quantum Cryptography*
CLI	*Command Line Interface*

1.4 Layout

In what follows, the evaluator marked obtained results using the following styles.

Technical fact: a bad practice, a minor flaw or a lack of defense in depth that does not lead to an attack. The style of a technical fact is as follows.

TF. X. Technical fact title
Technical fact description.

It is not required to fix technical facts in order to obtain a certification. Nevertheless, it is encouraged to fix them.

Non-compliance: a non-compliance in the behavior of the TOE regarding the security target, the specifications of the TOE or the state of the art. The style of a non-compliance is as follows.

NC. X. Non-compliance title
Non-compliance description.

Non-compliances, according to their seriousness, can be critical to obtain a certification. It is highly recommended to fix them.

Vulnerability: a flaw leading to an attack. The style of a vulnerability is as follows.

VUL. X. Vulnerability title
Vulnerability description.

Each vulnerability is analyzed in order to estimate its impact in comparison with the TOE use context. Following such an analysis, a vulnerability can be classified in different ways:

- **Exploited:**
  - The evaluator successfully implemented the vulnerability during the time allotted to the evaluation,
  - An exploited vulnerability is critical to obtain a certification.
- **Exploitable:**
  - The evaluator did not succeed in implementing the vulnerability during the time allotted to the evaluation, but considers that it is possible to implement it,
  - An exploitable vulnerability is critical to obtain a certification.
- **Non-exploitable:**
  - The vulnerability exists and is exploitable, but it cannot be implemented according to the TOE use context, especially due to assumptions of the security target,
  - A non-exploitable vulnerability is not critical to obtain a certification.
- **Residual:**
  - The vulnerability exists and is exploitable, but it requires too high an attacker level in order to be implemented,
  - A residual vulnerability is not critical to obtain a certification.

Recommendation: a suggestion for the configuration of the TOE or its usage for a secure. The style of a recommendation is as follows.

REC. X. Recommendation title
Recommendation description

2. PRODUCT DESCRIPTION AND SETUP

2.1 Referential and TOE version number

Table 3 - Identification of the evaluated product

Editor	HashiCorp
Product name	Vault
Version number	1.21.0

2.2 Product identification

In order to check the product version installed, one can run this CLI command unauthenticated:

```
vault -version
Vault v1.21.0 (818ca8b3575ea937ca48b640baf35e1b2ede1833), built 2025-10-21T19:33:18Z
```

2.3 Product description

Vault is a tool for securely storing, managing, and controlling access to secrets. It provides:

- Secure storage of secrets (with encryption);
- Creation and revocation of ephemeral secrets;
- Encryption of data as a service;
- Leasing of stored secrets, and their renewal;
- Parametrized revocation of secrets.

It ensures an access control on those secrets and provides various means of authentication (token, password, certificates, etc.). It is composed of a server (or a cluster of servers) accessible through an HTTP API (which can also be used through a CLI or a web UI).

2.4 Installation

2.4.1 Evaluation platform

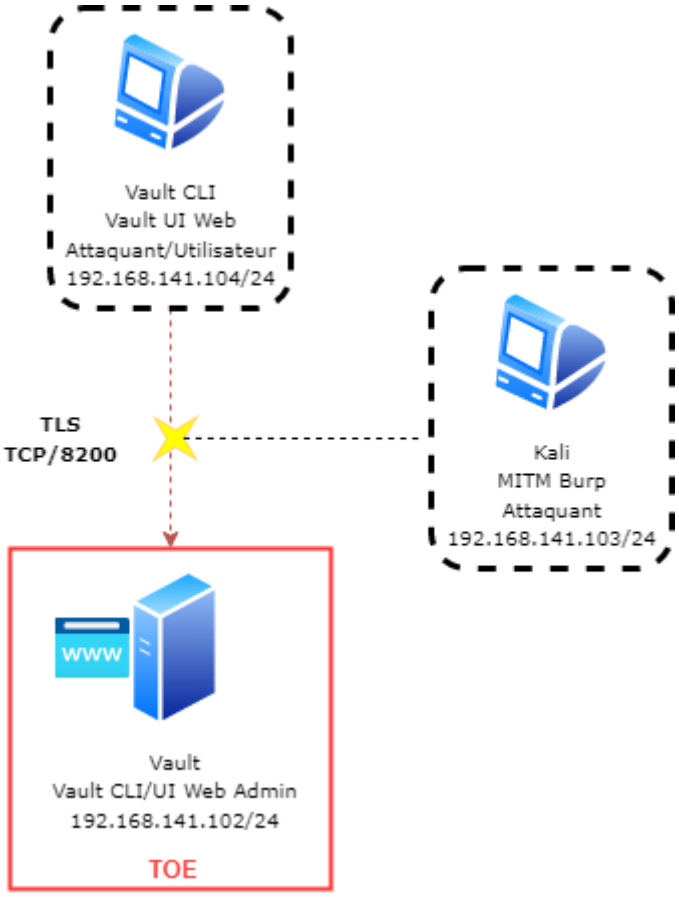

Figure 1 – Evaluation platform with the TOE in red

The evaluation platform includes several machines:

- The server machine:
  - OS: Ubuntu 24.04.3;
  - IP: 192.168.141.102/24.
- Man-in-the-Middle machine with Burp:
  - OS: Kali;
  - IP: 192.168.141.103.
- The client machine:
  - OS: Ubuntu 24.04.3;
  - IP: 192.168.141.104/24

2.4.2 Hardening

Hardening measures are listed in the following page of the documentation:

https://developer.hashicorp.com/vault/docs/concepts/production-hardening

The measures are listed in the following table. Some were applied when it is appropriate.

Table 4 – Hardening measures proposed by the documentation

Hardening measure	Applied	Remark
Baseline recommendations		
Do not run as root	Default	A dedicated user and group are automatically created with the APT installation
Allow minimal write privileges	Default	The APT installation creates a dedicated user and group, and only writes in a folder restricted to them.
Use end-to-end TLS	Default	Self-signed certificate by default, changed to certificate signed by and AC
Disable swap	Not applied	-
Disable core dumps	Not applied	Outside of scope
Use single tenancy	Not applied	-
Firewall traffic	Not applied	Outside of scope
Avoid root tokens	Not applied	Not applied for the evaluation purpose, but recommended
Configure user lockout	Default	User lockout applied by default for AppRole, LDAP and userpass
Enable audit device logs	Applied manually	-
Disable shell command history	Not applied	Outside of scope
Keep a frequent upgrade cadence	Not applied	-
Synchronize clocks	Not applied	-
Restrict storage access	Not applied	Avoid storage access from external attacker
Do not use clear text credentials	Not applied	Related to cloud credentials and HSM PIN
Use the safest algorithms available	Applied manually	TLS 1.3 only
Follow best practices for plugins	Default	No plugin used (outside of scope)
Be aware of non-deterministic configuration file merging	Not applied	-
Use correct filesystem permissions	Default	File permissions were not tampered with after the installation through APT which is appropriate.
Use standard input for vault secrets	-	Concerns the Vault client
Develop an off-boarding process	Not applied	-
Extended recommendations		
Disable SSH / remote desktop	-	No remote session
Use systemd security features	Default	Default with installation through APT
Perform immutable upgrades	-	No upgrade tested
Configure SELinux / AppArmor	Not applied	Outside of scope

Hardening measure	Applied	Remark
Adjust user limits	Not applied	Outside of scope
Be aware of special container considerations	-	Not installed as a container.
Consider memory usage when configuring disable_mlock when integrated storage	Not applied	Outside of scope

REC. 1. Applied as much as possible the hardening from developers

Several recommendations from the developers should be put in place, for example enforce TLS 1.3, disable swap, disable core dumps, use single tenancy, avoid root tokens, etc. Those recommendations, with appropriate explanations, can be found at https://developer.hashicorp.com/vault/docs/concepts/production-hardening.

2.5 Initial state and basic usage

When installed, the TOE is in a sealed state. It cannot decrypt data it stores:

```
vault seal status
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          true
Total Shares    5
Threshold       3
Version         1.21.0
Build Date      2025-10-21T19:33:18Z
Storage Type    file
Cluster Name    vault-cluster-0046aadb
Cluster ID      f81d83e3-fb0e-31e6-85fa-ca92ba9f735f
HA Enabled      false
```

One can connect to Vault from the client machine using the CLI binary and unseal it using the Shamir shares given by the TOE when initialized:

```
vault operator unseal
Unseal Key (will be hidden):
```

One can repeat this operation until the total threshold is reached (3 here). Once done, the TOE is unsealed:

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    5
Threshold       3
Version         1.21.0
Build Date      2025-10-21T19:33:18Z
Storage Type    file
Cluster Name    vault-cluster-0046aadb
Cluster ID      f81d83e3-fb0e-31e6-85fa-ca92ba9f735f
```

```
HA Enabled      false
```

Then, one can connect using the only token generated by the TOE during the initialization namely the **root** one:

```
vault login
Token (will be hidden):
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                  Value
---                  -----
token                hvs.l5cfLCdFdlMSfHzogNb6wPDO
token_accessor       vcFRApQ2JV4BOTnyGluOHA0d
token_duration       \u221e # infini
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
```

At this step, the elements activated by default are:

- Authentication: userpass;
- Secret engine: none;
- Policy: the default one, and the root one;
- Account: only a "root" one, associated to the root token generated during installation.

Root token can enable a KV2 secret engine mounted on a dedicated path namely `secret/` (the path is customizable):

```
vault secrets enable -path=secret kv-v2
```

and then write a basic policy for a specific path:

```
nano user1.hcl

path "secret/data/user1/mysecret" {
        capabilities = ["read", "update", "create", "list"]
}
path "secret/metadata/user1/mysecret" {
  capabilities = ["list"]
}
```

and push it into the TOE:

```
vault policy write user1_policy user1.hcl
```

Once this is done, root token can enable the *userpass* method and create an end user `user1` associated to this method and the policy `user1_policy`:

```
vault auth enable userpass
Success! Enabled userpass auth method at: userpass/
vault write auth/userpass/users/user1 password="user1pwd" policies="user1_policy"
```

and then connect with this account:

```
vault login -method=userpass username=user1 password=user1pwd

Success! You are now authenticated. The token information displayed below
```

```
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                  Value
---                  -----
token
hvs.CAESIHIHnv8vAxuq9mDAczSzyuS1m4Op_v9DYVfgi7lOZMR9Gh4KHGh2cy5ZcTNOZ0c4aTBia0prNDRwMU
w2djdQWXU
token_accessor       gutUP7lfRO26WDDXpmvdoNPD
token_duration       768h
token_renewable      true
token_policies       ["default" "user1"]
identity_policies    []
policies             ["default" "user1"]
token_meta_username  user1
```

One can notice that this automatically disconnect the user from the root "account". Indeed, the CLI uses the current token which is written in plaintext in the file `~/.vault-token` to build authenticated request to the API. When one switches to another user, the file `~/.vault-token` is overwritten with the new token.

REC. 2. Do not use shared account on client machine

The token of a Vault final user is written in plaintext in the file `~/.vault-token`. Thus, if the account on the client machine is shared, the token can leak to an unauthorized person just by reading it or by using the command `vault token lookup`.

`user1` can store his first secret in the path assigned by his policy:

```
vault kv put -mount=secret user1/mysecret name=foo
```

He can check the secret using the command

```
vault kv get -mount=secret user1/mysecret
======= Secret Path =======
secret/data/user1/mysecret

======= Metadata =======
Key                Value
---                -----
created_time       2025-12-03T10:19:35.197460944Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

==== Data ====
Key     Value
---     -----
name    foo
```

2.6 Ease of use

The installation is simple: it is a basic download and install from an APT repository. Moreover, in this case the TOE is configured with a default (and reasonably secure) configuration and ready to go. The solution is very intuitive to use. The product is well documented as well as the API and CLI agent used to interact with the TOE. Indeed, once the TOE and its web API are deployed, users simply have to use the CLI agent (same binary as the TOE) to interact with the

TOE via a REST API. Users can also use the Web UI, which is also neat and clear. Using the basic download and install from an APT repository, the product default installation **cannot lead to an insecure set up**.

3. IMPACT ASSESSMENT

3.1 File system

The impact on the file system can be summarized:

- Repository sources (of HashiCorp) added to APT configuration (with GPG key added to the keyring);
- Group and user `vault` created (no shell access for this account);
- `/etc/vault.d/`: configuration folder (write access only to user and group `vault`);
- `/opt/vault/tls/`: TLS certificate and private key (read/write access exclusively to user `vault`);
- `/opt/vault/data/`: operative data for Vault (all subfolders have read/write access exclusively to user `vault`);
- `/usr/bin/vault`: executable binary of the product;
- `/usr/lib/systemd/system/vault.service`: configuration to run the program as a service.

The Vault data stored in `/opt/vault/data` are as follows:

- `/opt/vault/data/audit`: related to the audit;
- `/opt/vault/data/auth`: related to the authentication methods;
- `/opt/vault/data/core`: related to the current configuration (such as keyrings);
- `/opt/vault/data/logical`: related to the user data (for each secret engine);
- `/opt/vault/data/sys`: related to tokens.

The impact on the system is minimal and consistent and the file permissions are restrictive.

3.2 Network

Only the port TCP/8200 is opened and corresponds to the TLS communication for the web server (that exposes all API endpoints). A network scan from the outside targeting the TOE has been done. Nothing sensitive was revealed by this scan. Vault redirects requests on URL `/` to the URL `/ui/` which is the connection webpage for the UI.

3.3 Processes

Only one process is related to Vault, and it is consistent with the *systemd* service that was configured during the installation.

```
vault vault 27795 /usr/bin/vault server -config=/etc/vauld.d/vault.hcl
```

In particular, it runs through a dedicated user account.

The process is automatically started using a *systemd* service with the following configuration.

```
[Unit]
Description="HashiCorp Vault - A tool for managing secrets"
Documentation=https://developer.hashicorp.com/vault/docs
Requires=network-online.target
After=network-online.target
ConditionFileNotEmpty=/etc/vault.d/vault.hcl
StartLimitIntervalSec=60
StartLimitBurst=3
```

```
[Service]
Type=notify
EnvironmentFile=/etc/vault.d/vault.env
User=vault
Group=vault
ProtectSystem=full
ProtectHome=read-only
PrivateTmp=yes
PrivateDevices=yes
SecureBits=keep-caps
AmbientCapabilities=CAP_IPC_LOCK
CapabilityBoundingSet=CAP_SYSLOG CAP_IPC_LOCK
NoNewPrivileges=yes
ExecStart=/usr/bin/vault server -config=/etc/vault.d/vault.hcl
ExecReload=/bin/kill --signal HUP $MAINPID
KillMode=process
KillSignal=SIGINT
Restart=on-failure
RestartSec=5
TimeoutStopSec=30
LimitNOFILE=65536
LimitMEMLOCK=infinity
LimitCORE=0

[Install]
WantedBy=multi-user.target
```

Figure 2 – Configuration of the *systemd* service for Vault

3.4 API REST endpoints

The endpoints of the API can be retrieved with the API `/v1/sys/internal/specs/openapi`. Using a root token gives a list of 302 endpoints with a description, HTTP methods, list of parameters and responses. Some of the API endpoints require "sudo" privileges, meaning that it must be present in the policy. Some do not require to be authenticated to perform actions on the TOE. At least, the great majority of API endpoints require a token in the request to authenticate the action. The OpenAPI list is updated every time the TOE opens new endpoints (for instance when a new secret engine is mounted on a path).

3.5 Evaluator judgment

The design of the product is minimalistic. The impact on the system follows this observation, which reduces the attack surface. Only one port is used by the TOE. A consequent web API is deployed. This web API is the only way for a remote attacker to disturb the TOE. While there is an OpenAPI available, the complete list of endpoints is not provided (it depends in particular of secret engine that create new endpoints).

4. PUBLIC VULNERABILITIES

4.1 Internal dependencies (COTS) used by the TOE

COTS used by the TOE and considered in the scope of this analysis are shown in the table hereafter (those are listed in the file `go.mod`). The analysis of the current versions of COTS and the last versions available at the date of the checking have been done by sampling.

The last versions available have been checked on 2025/11/17.

TF. 1. Many COTS are not up to date.

Many COTS are not up to date. However, due to the high number of internal dependencies, this fact is normal. Moreover, versions outdated are close to the latest ones. The editor releases minor versions regularly. This allows the update of the COTS on a regular basis and balance the issue.

4.2 External dependencies

An external dependency is a dependency that is not embedded directly by the TOE. It can be supplied by the host system or installed separately. The host system administrator is responsible for maintaining it in a secure condition.

The product is compiled statically and does not require an external dependency to run.

4.3 Public vulnerabilities on the TOE

A public vulnerability search leads to think that no one exists on the current version.

Research of public vulnerabilities on previous versions with the European Union Vulnerability database was done. These results were filtered on the last two years: 23 CVEs were found and are summarized in Table 5. Our analysis shows that the ones following are still applicable.

Table 5 – Recent CVE on previous versions of Vault

CVE	Description	CVSS v3.1	Version
EUVD-2025-28064 CVE-2025-4656	Vault Community Edition rekey and recovery key operations can cause denial of service	3.1	1.14.8 <1.20.0
EUVD-2025-23395 CVE-2025-6011	Hashicorp Vault has an Observable Discrepancy on Existing and Non-Existing Users	3.7	0 <1.20.1

4.4 Evaluator judgement

The product has many Go dependencies, including several from the developer. They are frequently updated since the product is actively developed. As a consequence, no vulnerabilities are present in the dependencies. However, numerous CVE were found over the past year, several with a high severity score. Two of them were found to be insufficiently fixed and are still exploitable on the evaluated version.

5. SOURCE CODE ANALYSIS

The source code analysis was focused on the module *github.com/hashicorp/vault* version 1.21.0 written in Go.

The number of lines in the Go language is large (around 500000 lines without blank lines and comments), but it includes test files. So, a second count by removing all files ending with `_test.go` gives has been executed.

The source code is voluminous and static analysis tools were used. The tool *gosec* found main issues that are justified in their context. The tool *staticcheck* was used to complete the analysis and several issues were found including unused values. In particular, it has revealed that HMAC values in token are ignored, with no impact on the security of the authentication. The code is well organized, clean, and so appears maintainable.

TF. 2. Several variables are not used in the source code

The static analysis of the source code has shown that some values are set and never used. Examples: a bit length ignored, certificate fields ignored, HMAC in service token ignored.

6. DYNAMIC TESTING

This section will present all the work and results done for this project regarding dynamic testing.

6.1 Tokens

Several tokens are manipulated by Vault and are analyzed in this section:

- Service token (formatted with prefix "hvs");
- Batch token (formatted with prefix "hvb").

Service tokens are persistent in storage, while batch tokens are not.

6.1.1 Service token

An authentication token can be acquired by several means:

- through a successful authentication from another method (such as *userpass*);
- provided by an operator that has the right to create token manually:
- when wrapping a secret using the API `/sys/wrapping/wrap`.

```
$ vault login -method=userpass username=user1 password=user1pwd

Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                    Value
---                    -----
token
hvs.CAESIHIHnv8vAxuq9mDAczSzyuS1m4Op_v9DYVfgi7lOZMR9Gh4KHGh2cy5ZcTNOZ0c4aTBia0prNDRwMUw2djdQWXU
token_accessor         gutUP7lfRO26WDDXpmvdoNPD
token_duration         768h
token_renewable        true
token_policies         ["default" "user1"]
identity_policies      []
policies               ["default" "user1"]
token_meta_username    user1
```

Figure 3 – Service token obtained after a userpass authentication

The token is mandatory to authenticate requests to the REST API (if authentication needed). The token is present in the header `X-Vault-Token` (see Figure 4).

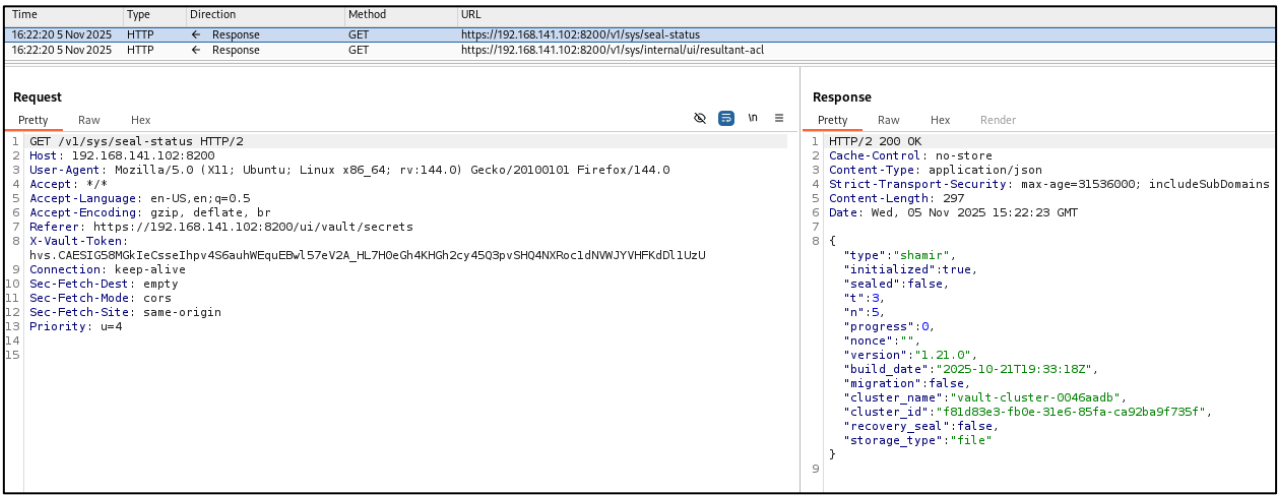

Figure 4 – API REST request (with token in X-Vault-Token header)

Finally, when a token is created, it is associated to a token accessor. This value can be used to refer to the token to perform some actions without knowing the token value: look up token properties and capabilities, but also renew or revoke the token itself.

By default, the token time-to-live is 32 days. It can be specified otherwise using the `ttl` parameter. A token is 24 characters in base62 (secret part), and some metadata. Here is the structure (see Figure 5).

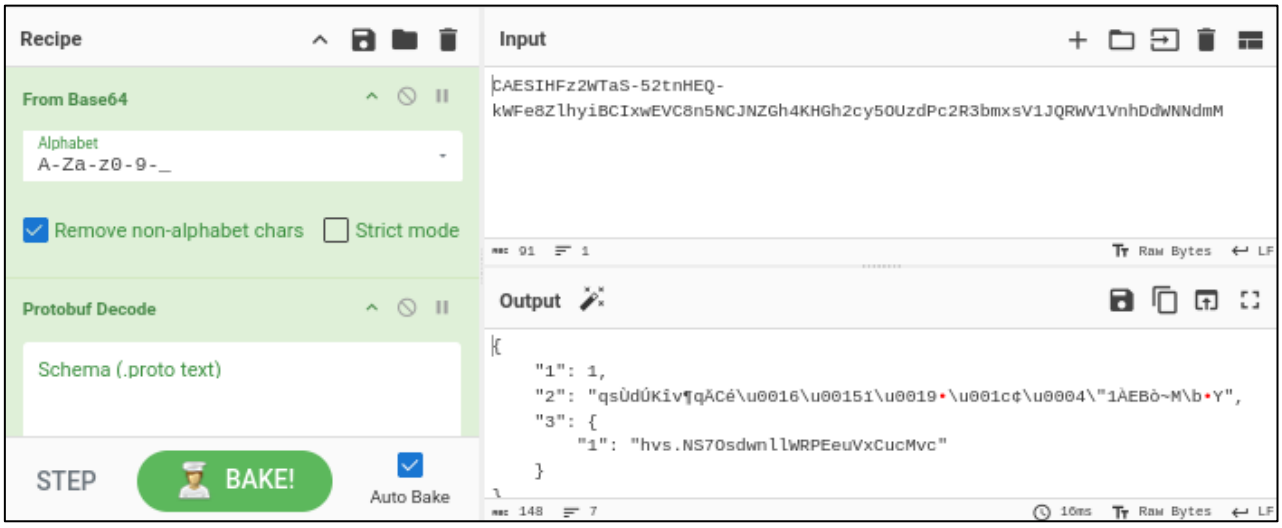

Figure 5 – Format of a service token

6.1.1.1 Response-wrapping

It is possible for a user to wrap the response from the TOE to a request on a specific endpoint by using the CLI option `-wrap-ttl` which corresponds to a HTTP header `X-Vault-Wrap-Ttl`. This action creates an ephemeral token with a cubbyhole associated (namespace specific to each token that can contain secrets, only the token has right on this path `/v1/cubbyhole/`):

```
vault write -wrap-ttl=5m auth/approle/login role_id=7dbab143-02dc-a477-92f5-
4806fc3f352f secret_id=532a4cac-32c1-90f2-f0f2-6b91d2724c57
Key                              Value
---                              -----
wrapping_token:                  hvs.CAESIM6OsBD5Qz2RYXhAfuFsNXzfchR67e7r-E-
lX9vfvxq8Gh4KHGh2cy5JUmRWNXM5VkVEMFdGb2ZNWUNsRHh5QVM
```

```
wrapping_accessor:              1ryAFffeGg05j0KpbTWT3Kj9
wrapping_token_ttl:             5m
wrapping_token_creation_time:   2026-01-12 11:24:51.852696278 +0100 CET
wrapping_token_creation_path:   auth/approle/login
wrapped_accessor:               epgg65EjJQCr8kTTfD2gulLk
```

To extract the ephemeral token, one can decode the Base64 URL wrapping token and then deserialize with Protobuf:

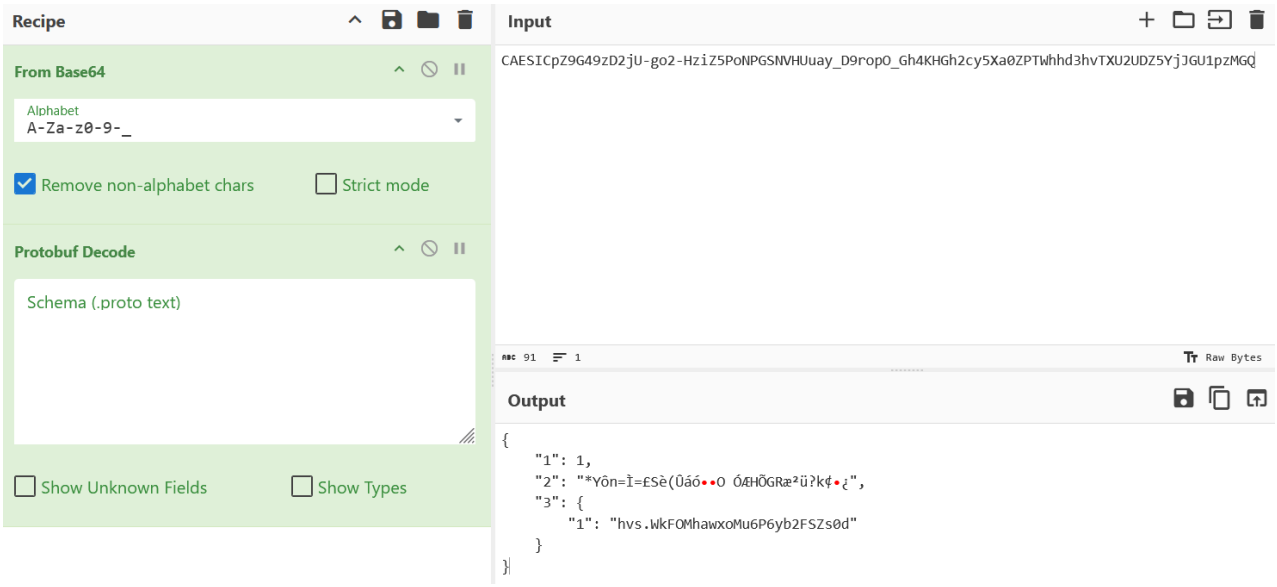

Figure 6 – Wrap token decoding and extract

This ephemeral token is different than the one that generated it. The response is stored inside the ephemeral cubbyhole:

Figure 7 - Wrapped response inside ephemeral cubbyhole

Concerning the security of this feature, **everyone possessing a wrapping token can unwrap it**, even non-authenticated user:

```
vault unwrap <token>
```

This means that a wrapping token must be carefully diffused. This fact is known by the editor, so this is a nominal behavior. When a wrapping token has been unwrapped by someone, an error message is returned when trying to unwrap it again. This can indicate that a wrapping token has been compromised before it came to its target. This fact is also mentioned in the documentation.

REC. 3. Alert administrator if a wrap token is invalid when opened

When a wrapping token has been unwrapped by someone, an error message is returned when trying to unwrap it again. This can indicate that a wrapping token has been compromised before it came to its target.

Once the token has been unwrapped, it is revoked by the TOE. Indeed, if we look at the ephemeral token, we can notice that this a one-time-token. Thus, if the wrapping token is compromised, the attacker can only get the response wrapped. Nothing else is exposed. Moreover, once the response is read, no more action can be done using the compromised token. The policy associated to the token is the following one:

```
vault policy read response-wrapping
path "cubbyhole/response" {
    capabilities = ["create", "read"]
}

path "sys/wrapping/unwrap" {
    capabilities = ["update"]
}
```

This policy allows no critical actions on the TOE.

6.1.2 Batch token

Since this type of token is not stored, its format is different and contains an encrypted blob of data using the latest encryption key of the keyring:

- it has `hvb.` as a prefix;
- the content of the remaining of the token is serialized with Protobuf, then encrypted and encoded in base64 (using the latest key from the keyring).

Since a batch token is encrypted using AES-256-GCM, it cannot be modified. Indeed, a slight modification results with an "invalid token" error.

Several attempts have been made to make requests with a token starting with `hvb` followed by a various number of characters. For some sizes of tokens, an internal error occurs. When the beginning of the token corresponds to an inexistent key in the keyring, then this error never occurs. No explanation was found, but no leak was apparent.

6.1.3 Conclusion

Tokens are random strings used to authenticate a REST API request on a restricted endpoint. Each token is associated to a policy allowing the owner of the token to request endpoints specified in its policy. A token is a random string of 24 base62 characters with more than 128 bits of entropy. The generation is done using a secure cryptographic DRBG through the function `rand.Reader()`. The use of wrapping-response tokens must be used with precautions. Indeed, anyone intercepting a wrapping-response token can unwrap it and access the data protected. No security issue was observed during the analysis: tokens play their role correctly.

6.2 Authentication

Several authentication methods are supported. Those analyzed were:

- *userpass*: authentication with username and password;
- *approle*;
- TLS authentication.

Each authentication method must be enabled first by an authorized user with the following command:

```
vault auth enable METHOD
```

The method can be `userpass`, `approle` or `cert`.

Upon successful authentication with such method, each request from the client to the vault server is token based authenticated (see section 6.1).

6.2.1 Userpass authentication

A user can be created (by authorized admin) with the authentication method *userpass* by specifying a username and a password (or password hash) using a token with sufficient privileges. The endpoint that manages this authentication is `POST /v1/auth/userpass/login/<USERNAME>`. The content request in JSON has a field `password` and is mandatory. Upon successful authentication, the server responds with a token for subsequent requests.

In case of failure, the response is "invalid username or password". This error message does not permit to distinguish if the username is wrong or the password. A fake Bcrypt hash is hardcoded for a comparison in constant-time (whether the user exists or not) to prevent the vulnerability from CVE-2025-6011. However, the repetition of a login request for a same username can end up in a user lockout only if the username exists; in this case, the error message is "permission denied".

TF. 3. Account enumeration using the lockout feature for userpass

The default configuration set up a user lockout after five authentication failures. This allows an attacker to search for valid usernames by repeating authentication attempts.

The Bcrypt parameters used to calculate the hash of the password are the default ones from *crypto/bcrypt*.

According to the default policy, a password cannot be changed. Updating it requires to apply the a specific policy first.

6.2.2 AppRole authentication

This authentication method is designed for apps, services or machines. A token with sufficient privileges creates a role attached to a policy:

```
vault write auth/approle/role/cuisinier token_policies="user1"
Success! Data written to: auth/approle/role/cuisinier

vault read auth/approle/role/cuisinier/role-id
Key        Value
---        -----
role_id    7dbab143-02dc-a477-92f5-4806fc3f352f
```

The `role_id` created is not a secret. Now the admin can generate a `secret_id`:

```
vault write -f auth/approle/role/cuisinier/secret-id
Key                   Value
---                   -----
secret_id             7a1e275c-4a1e-40a0-5ffa-9dc8ae3e3775
secret_id_accessor    332b1716-6844-eb45-d369-51fa46277208
secret_id_num_uses    0
secret_id_ttl         0s
```

These two credentials (role and secret identifiers) have to be transmitted to the final user (app, machine or whatsoever) **by external ways out of the scope of the TOE**. Then the final user can use these credentials to obtain a valid token with the policy defined attached to it. If the AppRole credentials are incorrect, the authentication fails as expected. The `secret_id` value is generated randomly using the following function in the file `/credential/approle/path_role.go`:

```
func (b *backend) pathRoleSecretIDUpdate(ctx context.Context, req *logical.Request,
data *framework.FieldData) (*logical.Response, error) {
    secretID, err := uuid.GenerateUUID()
    if err != nil {
        return nil, fmt.Errorf("failed to generate secret_id: %w", err)
    }
    return b.handleRoleSecretIDCommon(ctx, req, data, secretID)
}
```

The function `GenerateUUID()` calls the secure cryptographic DRBG `rand.Reader()`:

```
// GenerateUUID is used to generate a random UUID
func GenerateUUID() (string, error) {
    return GenerateUUIDWithReader(rand.Reader)
}
```

This returns a random 128-bit value identifier (see `uuid.go` file), which is secure and unpredictable for an attacker.

As expected, the token delivered respects the policy attached to it.

6.2.3 TLS authentication

The TLS authentication can be done with self-signed or signed by a CA (or an intermediate CA). It must be enabled, then a certificate for a CA or self-signed register must be added (the policy can be added at this moment too), then the user can authenticate.

```
vault auth enable cert
vault write auth/cert/certs/almond-auth certificate=@cert.pem ttl=3600
vault login -method=cert -client-cert=cert.com -client-key=cert.key name=almond-auth
```

Figure 8 – Commands to enable and authenticate with TLS

Upon successful authentication, the result contains the authentication token.

```
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                             Value
---                             -----
token                           hvs.CAESIPclWSZRm4q1lzXifLPHaKcIwJHlIZhQtslnzVBZLXiLGh4
KHGh2cy5FS2ZNdURzamxaUzJHZ3hsT3BRVU4wWGc
token_accessor                  Z7K3IeBynzi83WwSiHb9aNRa
token_duration                  1h
token_renewable                 true
token_policies                  ["default"]
identity_policies               []
policies                        ["default"]
token_meta_subject_key_id       n/a
token_meta_authority_key_id     n/a
token_meta_cert_name            almond-auth
token_meta_common_name          ARS
```

```
token_meta_serial_number        101724657544631011477072259529487274290302149100
```

Figure 9 – Authentication token received after TLS authentication

Certificates with malformations were generated under a CA that was registered such as explained above. The following bad certificates were as follows:

- bad signature;
- not yet valid;
- expired certificate;
- valid for server authentication;
- self-signed.

An authentication error occurs for each certificate.

The same method has been used for self-signed certificates that must be uploaded manually into the TOE.

TF. 4. [TLS authentication] It is possible to upload invalid certificates to the server
Certificates are not verified when uploaded to the server and are only verified when used.

However, the execution of the authentication request fails (see Figure 10).

```
$ vault login -tls-skip-verify -method=cert -client-cert=cert2.pem -client-
key=cert1.key name=almond-auth
Error authenticating: Error making API request.

URL: PUT https://vault.amossys.test:8200/v1/auth/cert/login
Code: 400. Errors:

* certificate mismatch of a trusted leaf certificate
```

Figure 10 – Request failure if the certificate has an invalid signature

The certificate is not validated when registered, but only when it is verified.

6.2.4 Conclusion

All authenticated operation on the TOE requires a valid token. To obtain a valid token associated to a policy, final user can use several authentication methods.

The userpass method is still vulnerable to an enumeration account (when user lockout is active, which is the default), despite the implementation of a fix.

Approle authentication method is designed for apps, services or machines unauthenticated that needs valid tokens. To do so, an administrator (that has privileges for these tasks) creates a role associated to a policy and generates a `secret_id`. This `secret_id` is used by the app/service to obtain a valid token associated to the policy attached to the role. The `secret_id` is a 128-bit value generated randomly by a secure cryptographic DRBG.

Finally, certificate authentication fails for an invalid certificate. However, the server does not check certificates when uploaded, but only when a user authenticates.

6.3 Policies

A policy is a list of several endpoints each associated with a set of capacities. Each token is linked to one or more policy. All the security of the TOE relies on this concept. Policies enforce access control and authentication. The possibilities offered by policies are wide and complex.

Moreover, a bad policy configuration or any misuse can compromise the whole security of the TOE.

It is advised to define basic policies for administrators, and then for the end users.

REC. 4. Stick to basic policy

The possibilities offered by policies are wide and complex. Moreover, a bad policy configuration or any misuse can compromise the whole security of the TOE. One can use the CLI option `-output-policy` to see the minimum policy requirements to perform the targeted action.

The documentation about policies is well written and exhaustive. Several examples are documented by the editor.

Furthermore, each path can be specified with the wildcard character "*" (supported only as the last character of a path) and the character "+" for any number of characters between two path segments.

A policy works like a whitelist: every endpoint not appearing on the list is unauthorized. This is more secure than a blacklist: every endpoint appearing on the list is unauthorized, thus an attack could be to bypass a non-authorized endpoint with encoding for instance. Whitelisting prevent bypass and thus is more secure.

Every token associated to a role in approle method or a user in userpass is attached to the default policy. This policy allows a set of non-critical actions useful for the token management. Indeed, all actions allowed by default policy are legitimate for a token and non-critical. No issue concerning the security was observed.

Several root tokens can be created and each one is associated to the root policy. A root token has basically every capability on every endpoint (except deny), even the sudo one for endpoints that requires this special capability (see section 6.8.1). Indeed, some critical endpoints (`/v1/sys/seal` for instance) requires the extra authorization capability (or flag) named sudo. Thus, root policy allows the access to every secret stored by final users in Vault using kv2 secret engine.

REC. 5. Use root tokens for exceptional necessity and partition the administration of the TOE

Use root tokens for exceptional necessity and under supervision of trusted operators. Indeed, root policy allows the access to every secret stored by final users in Vault using kv2 secret engine. Thus, a recommendation is to partition the administration of the TOE with several "admin policy". For instance, one can make an administrator policy for kv2 secret engine configuration with no access to final user's data. One can make another admin policy to manage the PKI secret engine. The root token with its root policy could be use only to manage policies (create, update, delete). This prevents any administrator from escalating its privileges.

However, a root token has no access to a token value, but only to its accessor. This is a good security practice.

6.3.1 Conclusion

All the security of the TOE relies on policies: they enforce access control and authentication. The possibilities offered by policies are wide and complex. Moreover, a bad policy configuration or any misuse can compromise the whole security of the TOE. A recommendation is to stick to basic policies, and fully understood ones. The default policy cannot lead to security issue. Whitelisting concept is better for security than blacklist for this purpose. Potential conflicts are

well handled. Management of policies is a critical security feature. It should be done under supervision of trusted operators.

Finally, the use of wildcards at the end of Kv2 path can cause path traversal attacks allowing an authenticated attacker to deny the access to other data. Moreover, the prohibition of the wildcard * in Kv2 related policies makes the product usage inconvenient. This vulnerability is depicted in section 6.8.2.

6.4 Logging

It is recommended to enable audit logging. To do so, admin can use this command on the TOE with **sudo** capability on the endpoint `/v1/sys/audit`:

```
vault login # with root token
vault audit enable file file_path=/opt/vault/audit-logs.log
tail -f audit-logs.log | jq
```

REC. 6. Enable audit logging
Audit logging is not enabled by default. It is recommended to activate it for security purpose.

The vault service is executed with a dedicated service account created by default at installation namely `vault` user on the server machine. Thus, the audit log file is protected with appropriate ACL (`-rw-------- vault:vault`).

Each entry of the logs is in JSON format. Information about requests on endpoints and their responses are logged. Logs are readable and exhaustive.

By default, all sensitive values are obfuscated using HMAC-SHA-256, like for instance the client token and its accessor. Thus, no sensitive value has been observed in the logs. For audit purpose, keeping the client token accessor in plaintext can help to manage and track suspect client token with unusual behaviors. To do so, the admin can use the command:

```
vault audit disable file/
vault audit enable file file_path=/opt/vault/audit-logs.log hmac_accessor=false
```

REC. 7. Disable obfuscation of token accessor in logs
For audit purpose, keeping the client token accessor in plaintext can help to manage and track suspect client token with unusual behaviors.

By default, no rotation nor compression are set up. The documentation mentions log rotation but does not recommend it.

REC. 8. Set up log rotation and compression for logs file.
Using cronjob, set up a rotation and compression for logs files. Indeed, logs are detailed and the volume of the audit file increase rapidly.

6.5 Secured communications

The CLI agent is the used to interact with the TOE Vault. The CLI build the correct requests to communicate with the different REST API endpoints at https://192.168.141.102:8200/. Otherwise, the web UI also forges correct requests to communicate with REST API. Both use the same HTTP channel on port TCP/8200.

REC. 9. Replace default certificate by a trusted certificate

Use an external trusted PKI to generate the X509 server certificate. The AC must be trusted by the web navigator or the client machine OS. The certificate must be compliant to [ANSSI-TLS] recommendations.

Network analysis showed the use of TLS to encapsulate communications between client (Vault CLI or web UI) and Vault server (TOE). The ciphersuites proposed by the TOE server are robust according to [ANSSI-TLS] except for two tolerated but not recommended: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA and TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA. The default backend certificate is self-signed certificate. For a safer use, set up a trusted certificate for the backend [REC. 9].

TF. 5. Server supports deprecated ciphersuites

Two TLS ciphersuites supported by the server use SHA-1 function for the HMAC part. Even though SHA1 function is not a robust hash function, HMAC-SHA-1 is tolerated because the robustness of the HMAC scheme does not rely on robustness of the hash function used. However, the use of HMAC-SHA-1 is not recommended.

6.6 Sealing

The unseal operation is as follows.

```
vault operator unseal
```

Figure 11 – Unsealing command

This operation requires a single unseal key share, and it responds with the current status of the unsealing with its progress.

```
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       3
Threshold          2
Unseal Progress    1/2
Unseal Nonce       cd969423-cc4b-a932-5f02-0cc87c821369
Version            1.21.0
Build Date         2025-10-21T19:33:18Z
Storage Type       file
HA Enabled         false
```

Figure 12 – Unsealing in progress

Once all required key shares, the “sealed” state is either “false” or “true” (in case of a wrong key share). The seal operation is as follows.

```
vault operator seal
```

Figure 13 – Sealing command

This required “sudo” privileges.

During the analysis, the team inspected the presence of AES key in memory, in particular to recover the unseal key and encryption key of secrets. After unsealing, several 256-bit keys were found in memory. Then the Vault server was explicitly sealed. The search of AES keys in memory was repeated and the same keys were still present. One of them is the unseal key

which was confirmed using a program based from the file shamir/shamir.go to recombine the Shamir secret shares.

```
Combined buffer (hex):
c08c989bd7912245b2fe169265732f1f0f00d8bbf987b05ea4c84049b28a57fc
```

Figure 14 – Unseal key recovered with Shamir Secret Sharing

The others were also confirmed to be the barrier key and the keyring keys.

TF. 6. Various AES keys are still present in memory when Vault is sealed

When Vault is sealed, there are still residual data that can be used to reconstruct the various AES keys used by the TOE: unsealing key, barrier key, and keys from the keyring.

The sealing and unsealing operation process are correctly managed. However, the AES keys in memory are still present in the process memory even when Vault is sealed. Residual data on the keys are not

The full details on Shamir's Secret Sharing is given in the cryptographic analysis in section 7.1.3.

6.7 Secrets engines

Secret engines can store, generate or encrypt data. Many engines are available, and the ones that are analyzed are:

- KV version 2;
- KV version 1;
- SSH certificate;
- X.509 certificate.

6.7.1 KV version 2

A basic use case is depicted in section 2.5.

The authorized paths for an end user are defined by its policy. If a final user has no capability on a path, he cannot access or store secrets using this path.

The data stored on the backend using `kv` secret engine are encrypted at rest using the current key of the keyring and AES-256-GCM. To each secret stored in Kv2 secret engine, metadata are associated:

```
vault kv get secret/user1/mysecret
======= Secret Path =======
secret/data/user1/mysecret

======= Metadata =======
Key                Value
---                -----
created_time       2025-11-19T10:24:54.626307488Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            898

==== Data ====
Key     Value
---     -----
```

```
name     foo
```

Figure 15 - KV2 metadata sample

One can delete a secret version:

```
vault kv delete -mount=secret user1/mysecret # use -versions option to specify one or
several versions to delete
```

Then, the secret version deleted is not accessible anymore using the CLI. However, it is not erased from the backend. Indeed, the secret is still present on the disk. To erase it from the disk, the final user can use the command:

```
vault kv destroy -mount=secret -versions=1 user1/mysecret
```

This command reaches the restricted endpoint `/v1/secret/destroy/user1/mysecret`.

If using the CLI, the secrets and credentials can be retrieved using the terminal's commands history (example: when looking for `vault login` command). Then, one must regularly clear its history to prevent data leak or deactivate it.

REC. 10. Deactivate terminal's commands history

If using the CLI, the secrets and credentials can be retrieved using the terminal's commands history.

Attempts of path traversal in order to access to unauthorized data are discussed in section 6.8.2: one vulnerability was found. Moreover, it is possible for an authenticated user of Kv2 to create many secrets in an authorized path to overload the TOE by sending `kv put` requests with a secret of 1MB random data. Secrets bigger than 1MB are refused by the TOE. There is no anti-brute force mechanism. One can notice that the size of the TOE disk increases rapidly. This requires the attacker to be authenticated (possess a valid token with authorizations on a Kv2 path): this is a strong hypothesis.

TF. 7. No anti brute force mechanism

An authenticated user can spam the creation of 1MB secrets using the Kv2 API. Depending on the network flow, the size of the data stored by the TOE can increase rapidly. This may be a risk for small infrastructure.

REC. 11. Dedicate a partition for TOE's data

An authenticated user can spam the creation of 1MB secrets using the Kv2 API. Depending on the network flow, the size of the data stored by the TOE can increase rapidly. Dedicate a partition for TOE's data, separate from the binary, is a good practice to supervise the data.

6.7.2 KV version 1

KV v1 is the former version of KV v2 secret engine. KV v1 is still available in Vault 1.21.0. This is basic key value store. One can only create, update or delete a secret. Versions, creation date, last update date or any metadata present in Kv2 is not available with Kv1.

The data stored on the backend using `kv` secret engine are encrypted at rest using the current key of the keyring and AES-256-GCM. The API is the same than the KV v2 one, but simpler, without the management of metadata, versions or soft delete. In this sense, KV v1 secret engine is more basic than KV v2.

The difference between the two engines is the presence of metadata associated to a secret in KV v2 (version management, possibility for a user to undelete a secret if it's not destroyed, etc.). For simpler use, one can use KV v1 engine:

```
vault secrets enable -path=kv1 -version=1 kv # using root token
```

One must configure the policy of a final user:

```
path "kv1/user1/*" {
    capabilities = ["create", "read", "list", "update", "delete"]
}
```

Then, `user1` can put its first secret:

```
vault kv put kv1/user1/my_secret name=hello
vault kv get kv1/user1/my_secret

==== Data ====
Key     Value
---     -----
name    hello
```

Just like `kv2`, the authorized paths for an end user are defined by its policy. If a final user has no capability on a path, he cannot access or store secrets using this path.

6.7.3 SSH certificates

This secret engine allows the creation of SSH certificate signed by an AC private key stored in the Vault server. A role must be created to sign certificates, this role defines the parameters allowed for a certificate to be issued (keypair, principals). Finally, SSH certificates can be signed by submitting a public key, and the result contains the certificate. To generate a SSH certificate, the overall steps according to the source code are as follows:

- the role is retrieved on the server;
- the public key is retrieved from the request;
- the fields of the certificate are collected (from the request, the role, or a template, or generated);
- the certificate is signed.

REC. 12. [PKI SSH] Restrict the public key parameters

By default, only a small set of algorithms are accepted for SSH certificates. The role defined for signing SSH certificates can further reduce the allowed algorithms: RSA (at least 2048, or 3072 bits), ECDSA (P-256 to P-521), and Ed25519.

REC. 13. [PKI SSH] Set the allowed principals if the role is attributed directly to the user

In the case the role used for generating a certificate is attributed directly to the user, then the `allowed_users` option in the role configuration should be set to principals specific to the user.

The validity period of the certificate is determined with the field `ttl` (time-to-live) from the request. If it is absent, then the default value is retrieved from the role configuration. Then, if the value is 0, the default value from the system configuration is obtained with a call to `backend.System().DefaultLeaseTTL()`. In that case, the duration is 30 minutes. The consistency with the maximal value authorized by the role is checked.

REC. 14. [PKI SSH] Set the default and maximal time-to-live values in the configuration of the role

To ensure that certificates have a reasonable period of validity, the values `ttl` and `max_ttl` in the configuration of the role for generating SSH certificate should be set.

Three points have raised during this audit for this part:

TF. 8. [PKI SSH] An inconsistent algorithm can be set for a role

A role for SSH certificate signing can have a parameter `algorithm_signer`, but it is not verified that it is consistent with the type of asymmetric key for the associated AC.

TF. 9. [PKI SSH] It is not possible to forbid critical options

The role configuration for critical options can only accept a few specified values or allow any values. In the second case, it means that the field `critical_options` in the SSH certificate request can contain any string.

TF. 10. [PKI SSH] Undefined cast for the serial number generation

The serial number for an SSH certificate is generated as a large integer that is casted into a Uint64. The official documentation of Golang indicate that the result is undefined.

6.7.4 X.509

The TOE can be used as a X509 PKI using the PKI secret engine. An authorized user (using a token with sufficient privileges) can generate a root CA and configure the CRL endpoint. Then, the authorized user can define a role, which is a set of parameters concerning the certificates to be issued by the root CA when a user requests a certificate using this role:

```
vault write pki/roles/pki_users allowed_domains=cesti.com allow_subdomains=true
max_ttl=83d
```

An authenticated user to the path `/v1/pki/issue/pki_users` can issue a certificate and its private key with the parameters defined and allowed by the role. The certificate is signed by the root CA, and the validity dates, the extensions, the key algorithm, the usages and so on are in line with the parameters defined and allowed by the role.

Concerning the security, only the authorized users (with a correct policy attached to its token) can request a certificate to this restricted endpoint. No private key associated to certificate issued has been found on the backend, which is a good practice for security.

To revoke a certificate, one can use the restricted endpoint `PUT pki/revoke` with the serial number associated to the certificate to be revoked. To be noticed that no further information than the serial number (which is public data) is required. One must keep this endpoint restricted and reachable only by admin.

REC. 15. [PKI X509] Keep the revoke endpoint restricted

To revoke a certificate, one can use the restricted endpoint `PUT pki/revoke` with the serial number associated to the certificate to be revoked. To be noticed that no further information than the serial number (which is public data) is required. One must keep this endpoint restricted and reachable only by admin.

6.7.4.1 ACME

Another method to obtain a certificate with Vault is via the ACME protocol. ACME protocol allows a remote client to automatically get a valid certificate associated to domain name (SAN) if he can prove that he masters the target domain name.

Administrator can enable ACME protocol using the following commands:

```
vault secrets tune -allowed-response-headers=Location -allowed-response-
headers=Replay-Nonce -allowed-response-headers=Link pki
vault write pki/config/cluster path=https://192.168.141.102:8200/v1/pki
vault write pki/config/acme enabled=true
```

Then, the ACME responder is up and listen to the non-restricted endpoint `/v1/pki/acme/directory`. The challenge tested is `http-01` as described in RFC 8555. The following security features of ACME protocol as defined by RFC 8555 are compliant in Vault case:

- Signature verification
- Challenge format and verification
- Nonce anti-replay
- No redirection followed by the server

Moreover, Vault supports External Account Binding (EAB) feature. Indeed, normally the ACME server lets any client create an ACME account and request certificates. There is no control by the ACME server about the flow of certificates requested. The EAB feature allows the ACME server to control the flow of request by restricting only to the authorized client the right to create an account. The client has to register himself to the TOE outside of the ACME protocol. To do so, admin must generate EAB credentials on the restricted endpoint `/v1/pki/acme/new-eab` that required the capacities `create` and `update`:

```
vault write -force /pki/acme/new-eab
Key               Value
---               -----
acme_directory    acme/directory
created_on        2026-01-05T14:31:32+01:00
id                b5336a61-3418-ec6b-79d2-f5325cac9232
key               vault-eab-0-n105lDAlQhu4e0JB1jWi3LCLda9SN4mbYfw0xBVpAoU
key_type          hs
```

The field `id` is the identifier of the external account, and the `key` is a shared secret. In the source code, the file `path_acme_eab.go` reveals that this key is 32 bytes HMAC key generated with the cryptographic `rand.Reader()` DRBG, which is secure.

The admin must enable the automatic EAB requirement for new ACME account requests:

```
vault write pki/config/acme eab_policy=new-account-required
```

Then, the EAB credentials must be transmitted to the ACME client **by a canal outside of the scope of the TOE**. If an ACME client wants to register an ACME account without any EAB credentials, the request will be refused by the TOE. If the ACME client possesses valid EAB credentials, it can register an ACME account via the ACME protocol and request a certificate providing the EAB credentials:

```
certbot certonly --register-unsafely-without-email --standalone --eab-kid "b5336a61-
3418-ec6b-79d2-f5325cac9232" --eab-hmac-key "vault-eab-0-
n105lDAlQhu4e0JB1jWi3LCLda9SN4mbYfw0xBVpAoU" -d acme.domain.com --server
https://192.168.141.102:8200/v1/pki/acme/directory
```

The request sent contains a JWS produced using the HMAC key in the field `signature`:

```
{
  "termsOfServiceAgreed": true,
  "externalAccountBinding": {
```

```
    "protected":
"eyJhbGciOiAiSFMyNTYiLCAia2lkIjogIjYxZjRmMDY4LTZmMzMtYjg3Mi05NDMwLTk3N2JmMGI4YzgyMyIsI
CJ1cmwiOiAiaHR0cHM6Ly8xOTIuMTY4LjE0MS4xMDI6ODIwMC92MS9wa2kvYWNtZS9uZXctYWNjb3VudCJ9",
    "signature": "rH2U4bVQJG26I7epOVYDhIFoGTCiWp-HWeaX0MaUBmU",
    "payload":
"eyJuIjogImt2SzVia3R5bUJPVEJKU3pFQXFfY2dtR3VNdE1BQjY3RHZmNG9ILUhJVWRvWXI3OEtJdUNpQjYwT
DhwRl9BLVh6eVFlN09XZkxkekNFRG41UTdDRy0yN2NoMEtpSTRYTEQ1QVotd1dMdWR2bXhZblZqR3NkSGdhTUE
0UDlKUzN1SXNxTGhwcFp6NVV6dUtUdEJyMDB5c1ZpSkNmM01nRHd2SmxKU3NQbU5paEN6TXg4M1p1X21HbXVie
XQ3TU1YOVhkdlljbnc4REZLQVZmQmJXTThDRkhYdlo2d1FFdjJ4VmlVNUFPSndyeWVvT1o2bjlmV1NENVpNWXJ
na2Z2YjJrbGJJWWZURWVmcVA4Zm14VVZjRllHdjFoOTNDSnBmTTlFVmlHR1lhamkwbFl6RWlTTTVYWHh1dkw1L
ThIbXdlWVZiQUx1S1dwZmF5NHcxRWtiWF9uUSIsICJlIjogIkFRQUIiLCAia3R5IjogIlJTQSJ9"
  }
}
```


The signature is an HMAC-SHA-256 tag. Indeed, the field `protected` contains:


```
{
  "alg": "HS256",
  "kid": "61f4f068-6f33-b872-9430-977bf0b8c823",
  "url": "https://192.168.141.102:8200/v1/pki/acme/new-account"
}
```


If the signature (tag) or the payload is modified, an error is returned by the TOE "failed to verify signature". If the `kid` or the symmetric key used to produce the tag are incorrect, an error is returned by the TOE "failed to verify eab".

If one registers an ACME account using EAB credentials, those cannot be reused afterward to register another account.

In addition to EAB, for more security, it is possible to configure the ACME responder to issue certificates compliant to a role to control the certificates to be issued by the CA. To do so, one can configure the ACME responder:

```
vault write -force /pki/config/acme -default_directory_policy=<role_name>
```

REC. 16. [PKI X509] Set up EAB with allowed roles, if ACME is used

Normally the ACME server lets any client create an ACME account and request certificates. There is no control by the ACME server about the flow of certificates requested. The EAB feature allows the ACME server to control the flow of requests using the command `vault write pki/config/acme eab_policy=new-account-required` by restricting only to authorized client the right to create an account. In addition to EAB, for more security, it is possible to configure the ACME responder to issue certificates compliant to a PKI role to control the certificates to be issued by the CA.

When an ACME order is authorized by the server, the client gives a CSR to the server in order to generate the valid certificate associated. A security measure is to check if the CSR given correspond to the ACME order passed, namely the domain mastered. For instance, a CSR containing a SAN for domain B cannot be accepted if the ACME order was for domain name A. In this case, the TOE detects the mismatch and returns an error.

6.7.5 Transit secret engine

The transit secret engine encryption service generates new private keys for that task and does not offer an encryption oracle using the barrier key of the TOE (that encrypts all data inside the TOE): that could have led to attacks.

The endpoints linked to this secret engine are restricted (it requires the right policy on the path). Moreover, this engine is a service, not a security functionality of the TOE. The evaluator judge that **no critical attack can be performed** against the TOE or its critical goods using this business functionality.

6.7.6 Conclusion

The KV secrets engine can be used to store secrets on the backend. The data stored on the backend using `kv` secret engine are encrypted at rest using the current key of the keyring and AES-256-GCM. Access to data is restricted to authorized path by the policy associated to the token requesting the engine. However, the use of wildcards at the end of Kv2 path can cause path traversal attacks allowing an authenticated attacker to deny the access to others data. Moreover, the prohibition of the wildcard "*" in KV version 2 related policies makes the product usage inconvenient. This vulnerability is depicted in section 6.8.2.

The TOE can be used to manage the an SSH PKI by providing dedicated roles. While an inconsistency was found (a role can be defined to sign with an algorithm inconsistent with the private key of the AC), and a potential problem with the user `critical_options`, no vulnerability was revealed. However, it requires a correct management of principals (which can be restricted to the roles).

The TOE can be used as a X509 PKI using the PKI secret engine. PKI roles can be defined to control the content of certificate to be issued. The endpoint used to request a certificate via a PKI role is restricted. Basic compliance testing did not reveal any security issue. To revoke a certificate, one can use the restricted endpoint `PUT pki/revoke` with the serial number associated to the certificate to be revoked. No further information than the serial number (which is public data) is required. One must keep this endpoint restricted and reachable only by admin.

The analysis has also shown that the ACME challenge *http-01* is correctly performed by the Vault ACME responder. Moreover, the TOE supports External Account Binding feature improving security and control over the issuance of certificates via ACME protocol. The EAB feature allows the ACME server to control the flow of request by restricting only to the authorized client the right to create an account.

The transit secret engine is a cryptographic service. Using this secret engine, one can encrypt plaintext, sign a document, hash a password, etc. Only the keys are stored on the backend encrypted like all other data with the current key of the keyring. Neither the ciphertext nor the plaintext are stored on the TOE. All algorithms proposed by this cryptography as a service functionality are theoretically robust. The implementations of these primitives come from trusted Go packages. This engine is a service, not a security functionality of the TOE.

6.8 REST API

6.8.1 Authentication and authorization

The TOE was recompiled with modifications to print some information during the handling of HTTP endpoints. Thus, inside the `CheckToken()` function, the following elements were traced:

- The token and the name associated (such as "root" in case of the root token);
- The Boolean `unauth` as parameter of the function `CheckToken()`;
- If the path is a "root path" (as explained below, it corresponds to the "sudo" capability);
- The results of the policy checks (allowed or not).

All paths were tested when the TOE is unsealed, the results were consistent with the expected behavior.

The "sudo" capability indicated in the OpenAPI specifications is verified through a Boolean `rootPath` in the function `CheckToken()`.

http/request_handling.go

```
// Check if this is a root protected path
rootPath := c.router.RootPath(ctx, req.Path)

if rootPath && unauth {
    return nil, nil, errors.New("cannot access root path in unauthenticated request")
}
```

Figure 16 – Verification if the endpoint is a root path ("sudo" capability required)

This value is consistent with the OpenAPI specifications. Furthermore, the correct application of the ACL for these endpoints has been checked using the `admin` policy: a selection of "sudo" endpoints are present in this policy, and only those results in an allowed ACL

Finally, the ACL verification is correctly checked for all endpoints that are not indicated as "unauthenticated", except the endpoint `/sys/internals/ui/feature-flags` that exposes feature flags for use with the UI and does not need to be authenticated[1].

The endpoints `/sys/wrapping/rewrap` and `/sys/wrapping/unwrap` require tokens generated specifically for the wrapping operation, therefore there is no ACL verification, but their existence is checked (see section 6.1.1.1).

Various endpoints were analyzed and all required permissions properly enforced.

6.8.2 Path traversal

A test aimed to perform a path traversal by encoding the special characters `../` in URL encoding. Two users were set up: `user1` and `user2`, each with their own policy to set secrets on the path `/data/user1/*` and `/data/user2/*` respectively with a `kv-v2` engine.

The evaluator tried to reach a secret `mysecret` stored in non-authorized path `/data/user1/` via an authorized path `/data/user2/../user1/mysecret` using `user2`. The TOE returned 500 Internal Server Error, but no 403 Forbidden. This suggests the request was authorized by the server. The logs revealed that the path targeted is indeed `secret/data/user2/../user1/mysecret`. The authorization comes from the fact that the token is authorized to all sub repositories with prefix `/data/user2/*`.

Moreover, **if** the last version of the secret is soft deleted, the evaluator can reach the metadata of it using this process:

[1] https://developer.hashicorp.com/vault/api-docs/system/internal-ui-feature

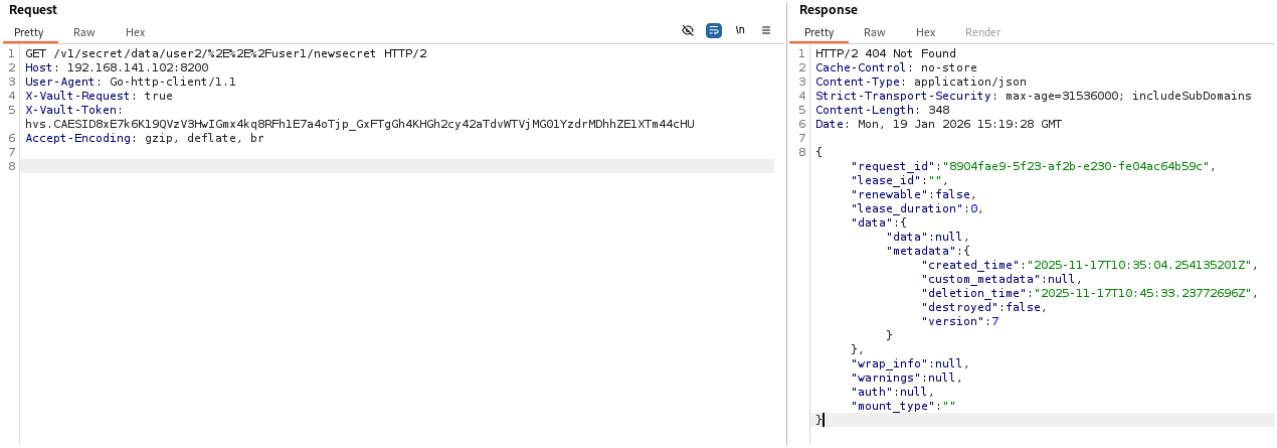

Figure 17 - Unauthorized metadata reach

However, the user is not authorized to get metadata ("deny" capability on `secret/metadata/user1/*` path).

The evaluator tried the same test, but with the DELETE method. **The unauthorized data is indeed deleted**.

A similar test was to use the PUT method to overwrite unauthorized secrets. **This is a success**:

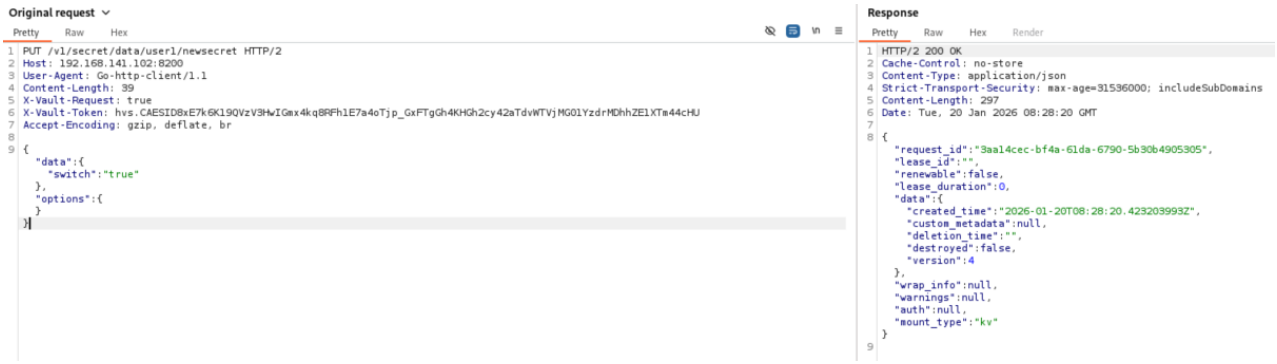

Figure 18 - Unauthorized overwrite

However, a legitimate user cannot access this data anymore and got the same error as the one depicted previously. But the attacker can reach the data he wrote when reaching the endpoint `secret/data/user2/%2E%2E%2F/user1/newsecret`:

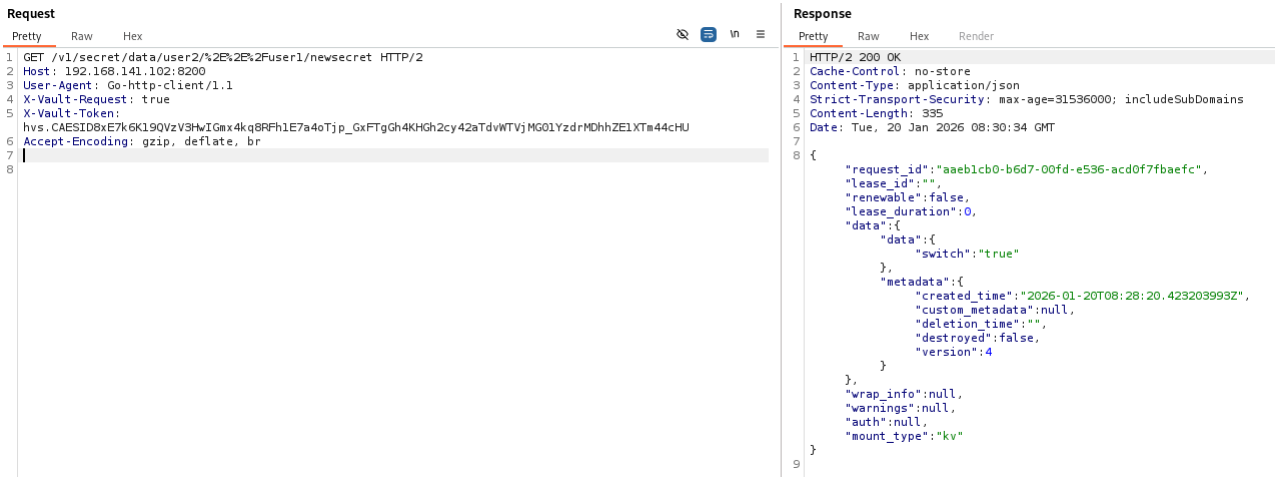

Figure 19 - Successful GET on unauthorized path

Using this method, the evaluator tried to reach the endpoint `secret/metadata/user2/%2E%2E%2F/user1?list=true` with GET method using `user2` token in order to list the names of the secrets of the sub repository `user1`, which `user2` cannot access due to its policy. However, the TOE lists the secrets presents in sub repository `user1`, despite the fact that `user2` is not authorized to access this information. **Thus, an attacker can list secrets of other users**.

The evaluator performed a similar test but using `kv1` secret engine which does not use versions. He tried to reach the secret `kv1/user2/vuln` through the authorized path `kv1/user1/*` with relative path `kv1/user1/%2E%2E%2F/user2/vuln`. An error 400 is returned:

```
Code: 400. Errors:

* 1 error occurred:
      * read failed: relative paths not supported
```

The TOE handles the malformed path.

VUL. 1. Path traversal
Due to the use of a wildcard "*", a non-privileged user of the secret engine KV2 can deny access to data he's not supposed to have access to using the PUT or DELETE method on a malformed path.

REC. 17. Do not use wildcard * in policies
If possible, always prefer absolute path within policy rather than path ending with the wildcard "*". Indeed, this can cause a path traversal attack when using the KV2 secret engine.

6.8.3 Unauthenticated API

From the unauthenticated endpoints, two issues were highlighted during the analysis. Indeed, the endpoint `/v1/sys/rekey/init` is not restricted and its purpose is to generate new Shamir's secret's part. To do so, Shamir's secrets possessors have to authenticate the process by entering each one their former Shamir's secret on this endpoint. The process is the same as the one used to unseal the TOE. A nonce is associated to a in working process.

However, this endpoint allows a DELETE method and can be reached by the CLI command:

```
vault operator rekey -cancel -nonce=<nonce_of_process>
```

This DELETE method aborts the process, if a process has been launched. If the nonce is not correct, the abortion fails. However, this nonce is public and can be retrieved by the command:

```
vault operator rekey
```

Thus, it is possible to do a script that spam the endpoint `GET /v1/sys/rekey/init`, check if a process is launched and extract the nonce associated. Then, the script can abort the procedure by reaching the `DELETE /v1/sys/rekey/init` endpoint using the correct nonce. An unauthenticated attacker can thus prevent any legitimate Shamir's secrets renewal.

TF. 11. DoS preventing Shamir's secrets renewal
An unauthenticated attacker can prevent any legitimate Shamir's secrets renewal using the endpoint `/v1/sys/rekey/init` with the `DELETE` method.

The second issue is similarly from the one above. Indeed, the endpoint `/v1/sys/generate-root/attempt` is not restricted and its purpose is to generate a root token. To do so, Shamir's secrets possessors have to authenticate the process by entering each one their former Shamir's secret on this endpoint. The process is the same as the one used to unseal the TOE. A nonce is also associated to in working process.

However, this endpoint allows a DELETE method. This DELETE method aborts the process, if a process has been launched. If the nonce is not correct, the abortion is still a success: the nonce is not checked. Thus, it is possible to do a script that spam the endpoint `DELETE /v1/sys/generate-root/attempt` using the CLI command:

```
vault operator generate-root -cancel
```

An unauthenticated attacker can thus prevent any legitimate new root token generation.

TF. 12. DoS preventing new root token generation
An unauthenticated attacker can prevent any legitimate new root token generation using the endpoint `DELETE /v1/sys/generate-root/attempt`.

To avoid these two attacks, one cannot restrict the exposure of the endpoints `/v1/sys/generate-root/attempt` and `/v1/sys/rekey/init` to a trusted network using a reverse proxy.

REC. 18. Set up a reverse proxy to protect unauthenticated endpoints
To avoid the two attacks described in section6.8.3, one cannot restrict the exposure of the endpoints `/v1/sys/generate-root/attempt` and `/v1/sys/rekey/init` to a trusted network using a reverse proxy.

6.8.4 Conclusion

The API has been tested to assert the correct verification of the ACL rights. All endpoints that require an authorization (including the "sudo" rights) are checked. The few endpoints that do not require authentication are justified. However, an unauthenticated attacker can prevent legitimate Shamir's secrets renewal using the endpoint `/v1/sys/rekey/init` with the `DELETE` method or any legitimate new root token generation using the endpoint `DELETE /v1/sys/generate-root/attempt`. Thus, a recommendation is to restrict the exposure of the endpoints `/v1/sys/generate-root/attempt` and `/v1/sys/rekey/init` to a trusted network using a reverse proxy [REC. 18]. The tool *dirsearch* has been used to try to reveal some non-documented endpoints. None was revealed. However, the use of wildcards at the end of Kv2 path can cause path traversal attacks allowing an authenticated attacker to deny the access to other data. Moreover, the prohibition of the wildcard "*" in Kv2 related policies makes the product usage inconvenient. Concerning the unauthenticated API, two issues were highlighted during the analysis.

7. CRYPTOGRAPHIC ANALYSIS

7.1 Implementation analysis

7.1.1 Random generation

The random number generation uses the *crypto/rand* package through the `Reader` variable. According to the *crypto/rand* package documentation and source code, it corresponds to a call to the `getrandom()` UNIX function with a flag set to 0. In that case, the system call accesses the ChaCha20 RNG identically to `/dev/random` (section 3.4.2 of [BSI-Linux-RNG]).

7.1.2 Cryptographic primitives

The cryptographic primitives used by the TOE are implemented by packages `crypto/<primitive_name>`. These Go packages (external dependency that must be updated by the TOE admin) are maintained by the Google team, are regularly audited by the international community and thus can be trusted.

7.1.3 Shamir Secret Sharing

This mechanism is implemented entirely in the file `shamir/shamir.go`. It provides two functions (the other functions are internal to this mechanism):

- `Split()`: from a secret, a number of parts and a threshold, it splits the secret in several secret shares;
- `Combine()`: from a number of shares above the threshold, it calculates the shared secret.

The `Split()` function starts with sanity checks to ensure that the threshold t is less than the number of parts n, the number of parts and thresholds are less or equal to 255 (which is dictated by the use of a field with 256 elements), the threshold is at least two (otherwise it makes this mechanism useless), and that the secret to protect is not empty.

The high-level management of splitting the secret is compliant with Shamir Secret Sharing. The correct implementation of the polynomial generation and evaluation, as well as the finite field implementation, are analyzed further.

The function `Combine()` takes as input a list of shares to build the master secret. The overall operations of the combination of shares is compliant with Shamir Secret Sharing.

The compliance of the multiplication, inverse and division operations have been tested using *Sagemath* to compare the results produced with these functions (with an exhaustive calculation on all inputs): the same results were obtained.

A polynomial is generated with the function `makePolynomial()`. All coefficients are randomly generated, except the constant coefficient which corresponds to a secret byte provided as input (it is also randomly generated outside this function since it is a byte of the unseal key).

shamir/shamir.go

```
// makePolynomial constructs a random polynomial of the given
// degree but with the provided intercept value.
func makePolynomial(intercept, degree uint8) (polynomial, error) {
    // Create a wrapper
    p := polynomial{
        coefficients: make([]byte, degree+1),
```

```
    }

    // Ensure the intercept is set
    p.coefficients[0] = intercept

    // Assign random co-efficients to the polynomial
    if _, err := rand.Read(p.coefficients[1:]); err != nil {
        return p, err
    }

    return p, nil
}
```

Figure 20 – Construction of a random polynomial for Shamir's secret sharing

This function is correctly implemented. Moreover, the implementation of the interpolation corresponds exactly to the expected formula for Lagrange polynomial interpolation.

7.2 Key management

Here is a diagram representing the key management of the TOE.

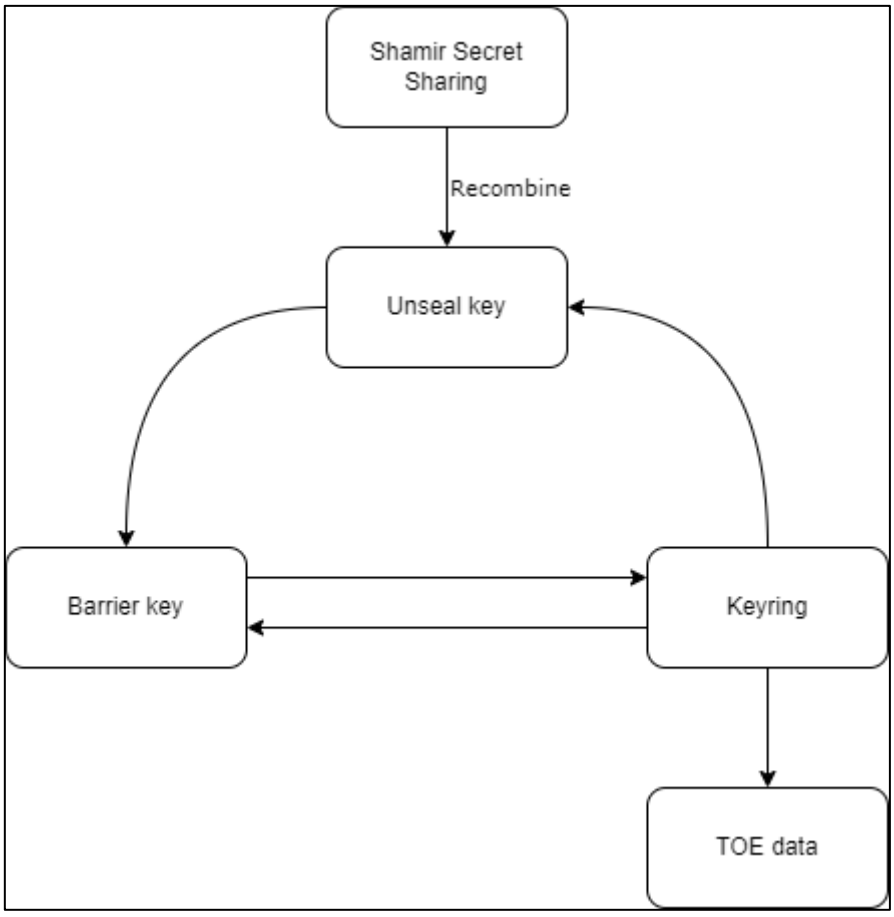

Figure 21 – Relation between AES keys of the TOE

The unseal key is generated first during the initialization. This key stays valid until a reinitialization process is executed. The barrier key follows the same pattern. However, both are newly generated when a rekeying has occurred.

The TOE data is protected with keys from the keyring. Those can encrypt a lot of data, and a rotation is automatically executed every 2^{32} requests, or it can be done manually with the

command `vault rotate`. These are not affected by a rekeying (the keys are still needed to decrypt old data).

The different AES-256-GCM keys were used in Python scripts to decrypt the data stored on the server.

7.3 Conclusion

The cryptographic analysis has shown that the algorithms used as defaults are compliant with [PG-083]. In some cases, obsolete algorithms are present (SHA-1), but are limited in their usage, and do not have an impact on the security. Finally, the implementation of the primitives and protocols come from the Golang cryptographic library which is considered robust[2].

[2] See https://go.dev/blog/tob-crypto-audit for the independent audit on the *crypto* module.

8. VULNERABILITY ANALYSIS

8.1 VULN-01

8.1.1 Exploit scenario

Due to the use of a wildcard "*" in policies, a non-privileged, but authenticated, user of the secret engine KV version 2 can deny access to data he's not supposed to have access to using the PUT or DELETE method on a malformed path

8.1.2 Prerequisite

The attacker must have the following ACL on a KV2 secret engine mount path:

```
path "<kv2_mount_path>/data/<path_authorized>/*" {
        capabilities = ["read", "update", "create", "list", "delete"]
}

path "<kv2_mount_path>/metadata/<path_authorized>/*" {
        capabilities = ["read", "list"]
}
```

It is necessary that this ACL ends with the wildcard *.

8.1.3 Attack path

The attacker can try to reach the endpoint `<kv2_mount_path>/data/<path_authorized>/%2E%2E%2F<unauthorized_path>?list=true` with GET method to list secrets within the unauthorized path. Then, he can reach the endpoint `<kv2_mount_path>/data/<path_authorized>/%2E%2E%2F<unauthorized_path>/<secret_name>` with PUT method or DELETE method. The targeted secret will be deleted using the DELETE method, or not accessible anymore by a legitimate user using the PUT method.

8.1.4 Scoring

VULN-01		
Due to the use of a wildcard "*", a non-privileged user of the secret engine KV2 can deny access to data he's not supposed to have access to using the PUT or DELETE method on a malformed path.		
CVSS v4.0	5.8 Medium	
Attack Vector	Local	
Attack Complexity	Low	
Attack Requirements	Present	
Privileges Required	Low	
User Interaction	None	
Impact on vulnerable system	Confidentiality	Low
	Integrity	None

VULN-01		
	Availability	High
Impact on subsequent system	Confidentiality	None
	Integrity	None
	Availability	None

9. EVALUATION SUMMARY

9.1 Results summary

9.1.1 Summary of technical facts

The following table summarizes the minor flaws uncovered during the evaluation. As a reminder, a minor flaw is a bad practice or a lack of defense in-depth that does not lead to an attack scenario, nor a scoring.

Table 6 – Summary of technical facts

Identifiant	Description
TF. 1	Many COTS are not up to date.
TF. 2	Several variables are not used in the source code
TF. 3	Account enumeration using the lockout feature for userpass
TF. 4	[TLS authentication] It is possible to upload invalid certificates to the server
TF. 5	Server supports deprecated ciphersuites
TF. 6	Various AES keys are still present in memory when Vault is sealed
TF. 7	No anti brute force mechanism
TF. 8	[PKI SSH] An inconsistent algorithm can be set for a role
TF. 9	[PKI SSH] It is not possible to forbid critical options
TF. 10	[PKI SSH] Undefined cast for the serial number generation
TF. 11	DoS preventing Shamir's secrets renewal
TF. 12	DoS preventing new root token generation

9.1.2 Summary of vulnerabilities

The following table summarizes the vulnerabilities uncovered during the evaluation. As a reminder, they are flaws that lead to an attack scenario and a scoring.

Table 7- Summary of vulnerabilities

Identifier	Description
VUL. 1	Path traversal

The details are given in section 8.

Additionally, the public CVE-2025-4656 and CVE-2025-6011 are mostly still applicable:

- CVE-2025-4656: a denial of service preventing Shamir's secrets (see section 6.8.3);
- CVE-2025-6011: account enumeration on userpass method using lockout feature (see section 6.2.1).

9.1.3 Summary of recommendations

The following table summarizes the recommendations made during the evaluation.

Table 8 – Summary of recommendations

Identifier	Description
REC. 1	Applied as much as possible the hardening from developers
REC. 2	Do not use shared account on client machine

Identifier	Description
REC. 3	Alert administrator if a wrap token is invalid when opened
REC. 4	Stick to basic policy
REC. 5	Use root tokens for exceptional necessity and partition the administration of the TOE
REC. 6	Enable audit logging
REC. 7	Disable obfuscation of token accessor in logs
REC. 8	Set up log rotation and compression for logs file.
REC. 9	Replace default certificate by a trusted certificate
REC. 10	Deactivate terminal's commands history
REC. 11	Dedicate a partition for TOE's data
REC. 12	[PKI SSH] Restrict the public key parameters
REC. 13	[PKI SSH] Set the allowed principals if the role is attributed directly to the user
REC. 14	[PKI SSH] Set the default and maximal time-to-live values in the configuration of the role
REC. 15	[PKI X509] Keep the revoke endpoint restricted
REC. 16	[PKI X509] Set up EAB with allowed roles, if ACME is used
REC. 17	Do not use wildcard * in policies
REC. 18	Set up a reverse proxy to protect unauthenticated endpoints

9.2 Evaluator judgment

The installation is simple: it's a basic download and install from an APT repository. Moreover, in this case the TOE is configured with a default configuration and ready to go. The solution is very intuitive to use. The product is well documented as well as the API and CLI agent used to interact with the TOE. Indeed, once the TOE and its web API are deployed, users simply have to use the CLI agent (same binary as the TOE) to interact with the TOE via a REST API. Users can also use the Web UI, which is also neat and clear. Using the basic download and install from an APT repository, the default product installation cannot lead to an insecure set up.

The design of the product is minimalistic. The impact on the system follows this observation, which reduces the attack surface. Only one port is used by the TOE. A consequent web API is deployed. This web API is the only way for a remote attacker to disturb the TOE.

Tokens are random strings used to authenticate a REST API request on a restricted endpoint. Each token is associated to a policy allowing the owner of the token to request endpoints specified in its policy. The generation is done using a secure cryptographic. The use of wrapping-response tokens must though be used with precautions. Indeed, anyone intercepting a wrapping-response token can unwrap it and access the data protected. No security issue was observed during the analysis: tokens play their role correctly.

To obtain a valid token associated to a policy, final user can use several authentication methods. Approle authentication method is designed for apps, services or machines unauthenticated that needs valid tokens. To do so, administrator creates a role associated to a policy and an `id` and generates a `secret_id`. This `secret_id` is used by the app/service to obtain a valid token associated to the policy attached to the role. The `secret_id` is a 128-bit value generated randomly by a secure cryptographic DRBG. Administrator must be careful to external canal of transmission to deliver the roleID and secretID to the target machine/app/service.

All the security of the TOE relies on policies: they ensure access control and authentication. The possibilities offered by policies are wide and complex. Moreover, a bad policy configuration or

any misuse can compromise the whole security of the TOE. A recommendation is to stick to basic policies, and fully understood ones. The default policy cannot lead to security issue. Whitelisting concept is better for security than blacklist for this purpose. Potential conflicts are well handled. Management of policies is a critical security feature. It should be done under supervision of trusted operators.

However, the use of wildcards at the end of Kv2 path can cause path traversal attacks allowing an authenticated attacker to deny the access to others data. A similar issue was revealed in 2021 with CVE-2020-8567 concerning the Vault plugin Kubernetes secret store. This fact puts the trust in the editor into perspective. Moreover, the prohibition of the wildcard * in Kv2 related policies makes the product usage inconvenient.

No undocumented endpoint was revealed. Concerning the unauthenticated API, two issues were highlighted during the analysis. An unauthenticated attacker can prevent legitimate Shamir's secrets renewal using the endpoint `/v1/sys/rekey/init` with the `DELETE` method or any legitimate new root token generation using the endpoint `DELETE /v1/sys/generate-root/attempt`. Thus, a recommendation is to restrict the exposure of the endpoints `/v1/sys/generate-root/attempt` and `/v1/sys/rekey/init` to a trusted network using a reverse proxy.

Network analysis showed the use of TLS to encapsulate communications between client (Vault CLI or web UI) and Vault server (TOE). The ciphersuites proposed by the TOE server are robust according to [ANSSI-TLS] except for two tolerated but not recommended (minor issue). The default backend's certificate is self-signed certificate. For a safer use, set up a trusted certificate for the backend.

Logs are written in a file protected in the backend machine with corrects ACL. Information about requests on endpoints and their responses are logged. Logs are readable and exhaustive. No sensitive value has been observed in the logs. For audit purpose, a recommendation is to deactivate the HMAC-SHA256 obfuscation on the token's accessor value. Another recommendation is to set up rotation and compression for logs files.

The KV secrets engine can be used to store secrets on the backend. The data stored on the backend using `kv` secret engine are encrypted at rest using the current key of the keyring and AES-256-GCM. Access to data is restricted to authorized path by the policy associated to the token requesting the engine. However, the use of wildcards at the end of Kv2 path can cause path traversal attacks allowing an authenticated attacker to deny the access to others data. Moreover, an authenticated user can spam the creation of 1MB secret using the Kv2 API. Depending on the network flow, the size of the data stored by the TOE can increase rapidly. This may be a risk for small infrastructure. Dedicate a partition for TOE's data, separate from the binary, is a good practice to supervise the data.

The transit secret engine is a cryptographic service. Using this secret engine, one can encrypt plaintext, sign a document, hash a password, etc. Only the keys are stored on the backend encrypted like all others data with the current key of the keyring. Neither the ciphertext nor the plaintext are stored on the TOE. All algorithms proposed by this cryptography as a service functionality are theoretically robust. The implementations of these primitives come from trusted Go packages. This engine is a service, not a security functionality of the TOE.

The TOE can be used as a X509 PKI using the PKI secret engine. PKI roles can be defined to control the content of certificate to be issued. The endpoint used to request a certificate via a PKI role is restricted. Basic compliance testing did not reveal any security issue. To revoke a certificate, one can use the restricted endpoint `PUT pki/revoke` with the serial number associated with the certificate to be revoked. No further information than the serial number (which is public data) is required. One must keep this endpoint restricted and reachable only by admin.

The analysis has also shown that the ACME challenge *http-01* is correctly performed by the Vault ACME responder. Moreover, the TOE supports External Account Binding feature improving security and control over the issuance of certificates via ACME protocol. The EAB feature allows the ACME server to control the flow of requests by restricting only to the authorized client the right to create an account.

The cryptographic analysis has shown that the algorithms used as defaults are compliant with [PG-083]. In some cases, obsolete algorithms are present (SHA-1), but are limited in their usage, and do not have an impact on the security. Finally, the implementation of the primitives and protocols come from the Golang cryptographic library which is considered robust.

In view of the work carried out, the ITSEF delivers the following opinion:

The product "Vault" version 1.21.0 is deemed reliable for production. However, the recommendations made in this report must be followed in order to ensure safe use through a hardened configuration.

10. REFERENCES

Ref.	Title
[RGS-B2]	*« Gestion des clés cryptographiques : Règles et recommandations concernant la gestion des clés utilisées dans des mécanismes cryptographiques »*, ANSSI, Annexe B2 version 2.0, 2012/06/08
[RGS-B3]	*« Authentification, Règles et recommandations des mécanismes cryptographiques »*, ANSSI, Annexe B3, version 1.0, 2010/01/13
[PG-083]	*« Guide des mécanismes cryptographiques – Règles et recommandations concernant le choix et de dimensionnement des mécanismes cryptographiques »*, ANSSI, version 2.04, 2020/01/01
[PG-078]	*« Recommandations relatives à l'authentification multifacteur et aux mots de passe »*, ANSSI, version 2.0, 2021/10/08
[BP-106]	*« Automatisation de la gestion des certificats avec ACME »*, ANSSI, version 1.0, 2024/12/24
[PA-079]	*« Guide de sélection d'algorithmes cryptographiques »*, ANSSI, version 1.0, 2021/03/08
[ANSSI-TLS]	*« Recommandations de sécurité relatives à TLS »*, ANSSI, version 1.2, 2020/03/26
[PA-009]	*« Recommandations pour la mise en œuvre d'un site web : maîtriser les standards de sécurité côté navigateur »*, ANSSI, version 2.0, 2021/04/28.
[FKP17]	"On the One-Per-Message Unforgeability of (EC)DSA and Its Variants", M. Fersch, E. Kiltz, and B. Poettering. Theory of Cryptography, 2017
[Bro02]	"Generic Groups, Collision Resistance, and ECDSA", D. Brown. Cryptology ePrint Archive, Report 2002/026, 2002
[BSI-Linux-RNG]	"Documentation and Analysis of the Linux Random Number Generator", BSI, version 6.2, 2025/09/19