EDGE X FOUNDRY™

Vault Master Key Protection Proposal v0.5

DRAFT

Change History

Date	Author/Editor	List of Changes
Nov-29-2018	David Ferriera (ForgeRock)	Initial Draft Release
Dec-05-2018	James White (Dell)	Comments/Design details
Dec-06-2018	Alain Pulluelo (ForgeRock)	Comments/Design details
Dec-10-2018	David Ferriera (ForgeRock)	• Consolidate comments • Changes to API specification section • Changes to plugin architecture section
Dec-11-2018	David Ferriera (ForgeRock)	Incorporated comments from Jim and Tingyu
Jan-16-2019	David Ferriera (ForgeRock)	Incorporated comments from the working group sessions (Nov/Dec 2018)
Aug-02-2019	Bryon Nevis (Intel)	Revamped implementation proposal (0.4)
Apr-10-2020	Bryon Nevis (Intel)	Updates to proposal (0.5)

Introduction

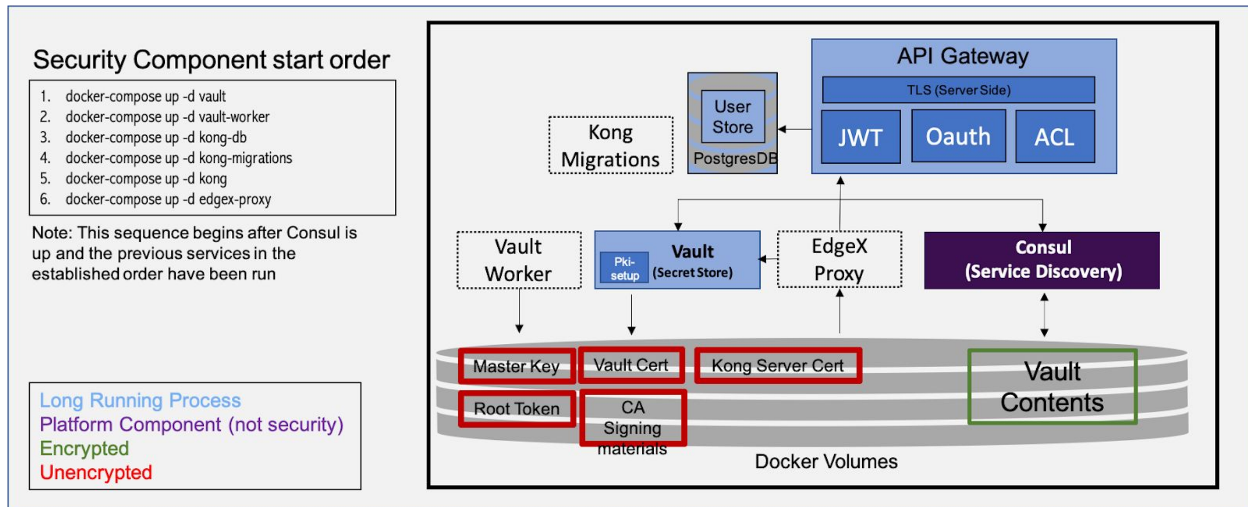
This document is a revamp of EdgeX Hardware-based Secure Storage Design with the specific aim of protecting the Vault Master Key that underlies the EdgeX Secret Store. This document does not seek to standardize an interface to hardware secret storage, but rather creates an extension point that allows for EdgeX to retrieve an encryption key from the platform via a software- or hardware-defined mechanism, and use that key to secure Vault.

Overview

The EdgeX platform contains (or will contain) many secrets such as database credentials, various encryption/signature keys, X.509 certificates, authentication tokens, username/password pairs, Oauth client credentials, IOT cloud platform credentials/keys (e.g., AWS IOT keys). EdgeX has chosen to use HashiCorp Vault for managing secrets. Currently, several of these "initial" secrets are stored on the file system in the clear. The goal is to enable protection of these initial secrets in a portable way that can be extended to hardware-backed implementations.

The EdgeX platform is intended for use on multiple hardware platforms. These platforms include Intel, ARM64 and ARM32. Within these platforms, there are different technologies to solve hardware-based trust, storage and related problems.

Current State (Delhi and Edinburgh)



The initial startup flow for secret management in Deli and Edinburgh is as follows:

1. A utility, security-secret-setup, is run that generates a certificate authority and TLS end-entity certificates for Vault and Kong. (These are unencrypted.)
2. Vault is started with the generated PKI

3. A vault-worker service is started that performs the following actions:
   a. Initialize Vault and store the initialization response which includes a Vault Master Key (VMK) as well as a Vault root token into the file system. (Alternatively, if the VMK already exists, use to to unseal the Vault.). The VMK is also unencrypted.
   b. Import the Kong certificate and place it into Vault
   c. Configure Vault policies and generate additional non-root tokens used by other services to access Vault.
4. After Vault setup is completed, Kong and PostgresDB are started and configured. As Kong starts, it requires a TLS certificate and its corresponding private key (as noted above) for the admin API. If they don't exist in the configured location on the file system, Kong generates and installs a cert and places it in the expected path on the file system.
5. The Edgex Proxy service configures Kong. As part of this configuration, the root token for Vault is retrieved from the file system and used to call Vault to retrieve and install the Kong external cert.

A Quick Vault Overview

Vault is a software based secret management service. Vault supports a number of pluggable "secrets" engines that provide, among other functionality:

- Secrets storage – encrypted key/value pairs with secure key access, wrapping and management included (rotations, TTL, usage frequency, etc.)
- Provisioning of database usernames and passwords
- Encryption as a service
- Generation of OIDC-connect compliant OAuth tokens
- Generation of public/private key hierarchies

While EdgeX by default runs a single Vault (and Consul for that matter) server, Vault also has the ability to run in a fault tolerant mode in conjunction with Consul. In this mode multiple vault servers and multiple consul servers are run simultaneously.

In EdgeX, Vault will initially be used for secrets storage. Eventually, almost (more on this in the next section) all EdgeX secrets will be stored in Vault including:

- Certificates
- Database credentials
- Cloud/IOT Platform credentials
- Encryption/Signature keys

There are several constraints / limitations to how Vault can be used in EdgeX:

- Vault pre-assumes a PKI has been deployed and that the private key for the TLS end-entity certificate is available in plaintext; Vault cannot easily self-seed its own PKI.
- The Vault master key is generated by Vault itself. This means we can't "give" Vault the encryption key that we want it to use. Instead, we must take the key that Vault gives us and store it securely.

- The preferred configuration for Vault is to have the Vault on physically separate hardware that only runs Vault. As an IoT framework, EdgeX must be able to work when all components are located on a single processing node.
- The preferred configuration for Vault is that the Vault is manually unsealed by human operators. As an IoT framework, EdgeX must be able to unseal the Vault when no human is present.
- Although Vault has an enterprise version that has a PKCS#11 seal plugin, EdgeX desires to provide a solution to protect the Vault master key that does not depend on enterprise features and has a reasonable software fallback.

Initial List of Assets and Proposed Disposition

Asset	**Proposed Disposition**
TLS CA private key	Unique per device. Shred and dispose after TLS end-entity certificate generation is complete (*1)
TLS Vault end-entity private key	Unique per device. Leave unencrypted for Fuji; encrypt at-rest in future release
TLS Vault end-entity private key	Unique per device. Leave unencrypted for Fuji; encrypt at-rest in future release
Vault master key	Encrypt according to this proposal
Vault root token	Revoke (recreate from VMK if needed)
Non-root tokens	Revoke all previous tokens from previous runs and re-issue at startup. (*2)

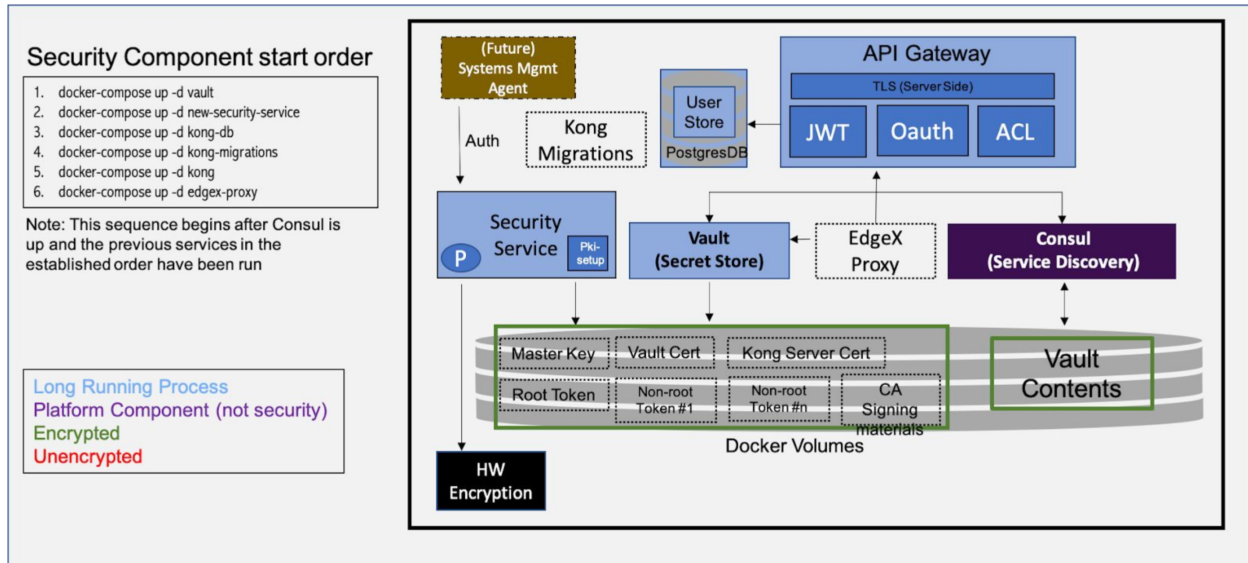
(*1) In the future, Vault can be enabled as a delegated certificate authority with the ability to generate additional certificates at runtime for other needs, such as service-to-service encryption and authentication.
(*2) This ensures that at startup every service has a fresh valid token and also addresses potential disk usage issues due to token leakage.

Usage of Secrets

Initial secrets are only retrieved at start time (both initial and subsequent). This will include, in the future, any individual service that stores secrets in Vault. Anytime a service starts, it must be able to retrieve/store secrets from Vault using a valid token. Therefore, any Vault-consuming service must have a vault token at startup time. Non-root tokens have a Time to Live (TTL)

barrier that implies a service to handle the refresh process, TTL cycles are also limited. These features cannot be disabled by Vault configuration, therefore have to be automated within the future usage procedures.

Future State



Hardware based Encryption

Cryptographic services are a requirement of the Basic Endpoint Security Level as defined by the Industrial Internet Consortium. The IIC Endpoint Security Best Practices document also refers to other industry security standard documents such as IEC 62443-33 and NIST-800-53r5. Hardware Based encryption is an important part of meeting these standards. Toward that end, EdgeX will enable hardware based secure storage utilizing hardware based encryption. Although not recommended, EdgeX may also be run without hardware based secure storage. There are multiple existing systems for Hardware based encryption with the most widely used being:

- TPM – The Trusted Platform Module 2.0 specification is provided by the Trusted Computing Group.
  - TPM boards can be manufactured by 3rd party vendors for inclusion on a system board. A list can be found here. Or the TPM can be manufactured and included on the system board by the same vendor such as Intel.
- TEE – Trusted Execution Environment – An implementation that provides a secure enclave or secure OS to execute trusted code in isolation from user space within the CPU.
  - ARM TrustZone is an architecture of the Hardware layer to support this implementation. TrustZone introduced a special CPU mode called "secure mode"

in addition to the regular normal mode, the architecture includes the SoC and peripherals that are connected with the SoC.

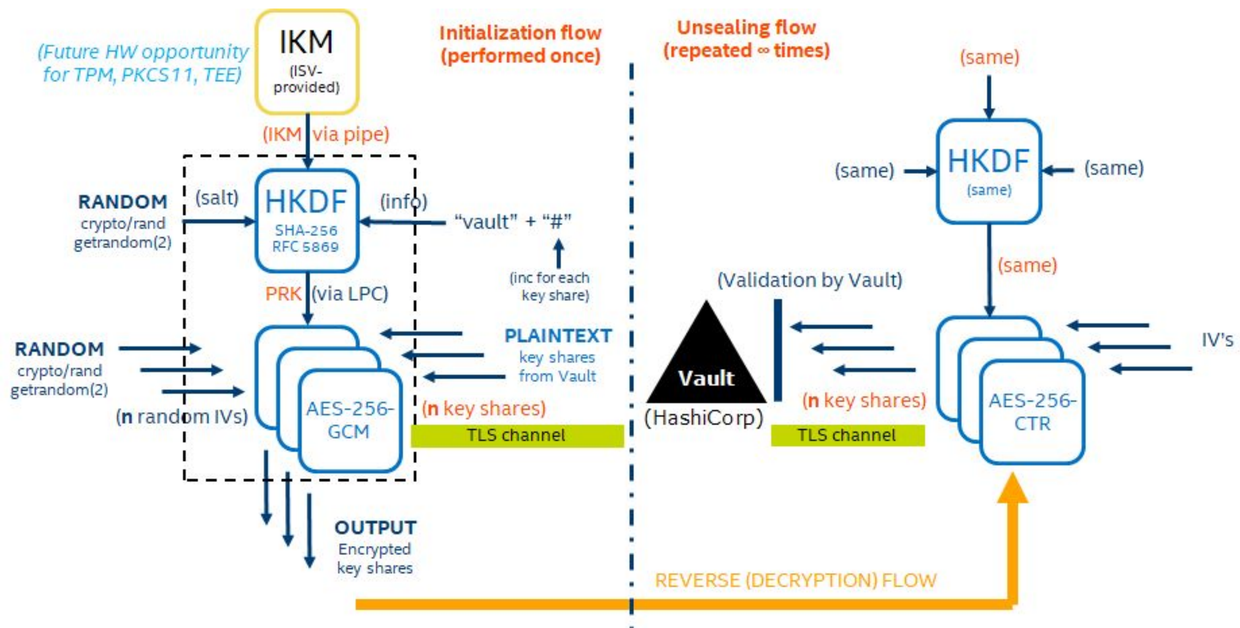
While the architectures differ significantly, the general principle is essentially the same. Both implementations have different ways to perform encryption, hashing, key generation and key storage functions. They also differ in cryptographic inputs like randomness and entropy. Storing a key in hardware provides yet another layer of security over software-based solutions. Simply having access to the filesystem does not provide an attacker an avenue for successfully decrypting data. Although some TEE implementations can provide slices of the filesystem which are themselves isolated and blinded from user space. These zones need supplemental TEE security mechanisms like monotonic clock to avoid alterations on replay and downgrade attacks.

Note: These types of systems frequently have small CPU and small storage capacities. As such, they are meant to be used for encryption and storage of small amounts of data on an infrequent basis.

This document is not intended to be a full explanation of hardware based encryption systems.

Proposal for Vault Master Key protection in EdgeX

This document proposes to protect the Vault master key by introducing a RFC-5869 key derivation function (KDF) to produce a wrapping key that will be used by the vault-worker process to encrypt the Vault master key.



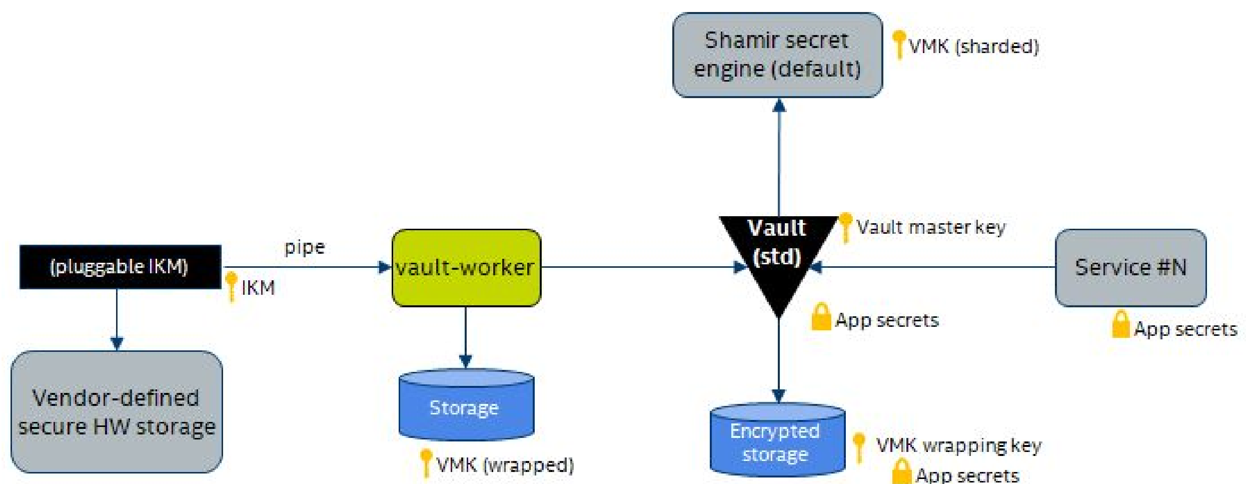
An RFC-5869 KDF requires three inputs. A change to any input results in a different output key:

- Input keying material (IKM). It need not be cryptographically strong, but it is the "secret" part of the KDF.
- A salt. A non-secret random number that adds significantly to the strength of the KDF.
- An "info" argument. The info argument allows multiple keys to be generated from the same IKM and salt. This allows the same KDF to generate multiple keys each used for a different purpose. For instance, the same KDF can be used to generate an encryption key to protect the PKI at-rest.

The advantage of the KDF-based solution is that the IKM need not be stored in the file system. For example:

- It could be externally supplied by some management node on the network that has the ability to attest the device.
- It could be stored in NVRAM on a TPM or an HSM.
- It could be a random value that is sealed cryptographically to a TPM or HSM or TEE and released by local or remote attestation.

In the proposed design, the "vault-worker" component that initializes and unseals the secret store takes the IKM from a pipe. This IKM is provided by a vendor-defined mechanism.



To further strengthen the solution, an implementation can choose to engineer a solution whereby the IKM is only released a configurable number of times per boot, so that malware that runs on the system post-boot cannot retrieve it:

- A kernel driver could set a counter in the kernel preventing re-release of the IKM.
- A TPM implementation could extend a PCR once the IKM had been retrieved preventing further retrieval of the IKM in the current boot or block a certain NVRAM index from being re-read.

IKM API Specification

As part of unsealing the secret store, an optional binary will be invoked that is expected to provide the IKM for the KDF:

```
IKM(1)              Input key material for KDF                  IKM(1)
```

NAME

```
ikm - Return input key material for a hash-based KDF.
```

SYNOPSIS

```
ikm
```

DESCRIPTION

```
ikm outputs initial keying material to stdout as a lowercase
hex string to be used for the default EdgeX software
implementation of an RFC-5869 KDF.

The ikm can output any number of octets. Typically, the KDF
will pad the ikm if it is shorter than hashlen, and hash the
ikm if it is longer than hashlen. Thus, if ikm returns
variable-length output it is advantageous to ensure that the
output is always greater than hashlen, where hashlen depends on
the hash function used by the KDF.
```

NOTES

```
If ikm is a shell script, it may be useful to filter the output
through "xxd -p"
The default ikm provided by EdgeX returns an IKM of 32 octets
of zeroes.
```

EXAMPLE

```
ikm
3cb25f25faacd5
```

SEE ALSO

```
kdf(1)
```


```
IKM(1)              Input key material for KDF                  IKM(1)
```

Changes to Vault Worker

The existing vault worker initializes Vault and stores the initialization response (resp-init.json) to a persistent docker volume. The following are the proposed changes to the vault worker:

- Generate a random salt value using golang crypto random function and save/load it from the persistent docker volume.
- Invoke the `$IKM_HOOK` and capture the resulting encryption key.
- Encrypt or decrypt the keyshare using AES-256-GCM using the derived key and generated random salt and store the encrypted keyshare in the persistent docker volume.

Hardware extensibility mechanisms

This section documents how the proposed Vault key protection mechanism could be extended to hardware-based protection mechanisms.

Extensibility to software enclaves

Using a software enclave, the entire KDF computation can be placed inside of an enclave that is bound to hardware. Both the IKM and salt can be generated by using a HWRNG and the entire enclave state can be unique per-machine and sealed per-machine.

Extensibility to PKCS#11-compliant HSMs

Using a PKCS#11 interface, it should be possible to:

- Generate a random number to serve as IKM to the KDF
- Generate a new key persistently stored in the HSM
- Use the persistent key to encrypt or decrypt the IKM at rest

Extensibility to TPM's

It is possible to use a TPM as a PKCS#11 device using a library such as tpm2-pkcs11. It would also be possible to write to a native TPM interface and

- Generate a TPM primary object in the owner hierarchy and store in the TPM NVRAM
- Generate a random number to server as IKM to the KDF, seal it to the TPM, and store the sealed object in the TPM NVRAM
- Store the KDF seed in TPM NVRAM as well.
- Require PCR-based authorization policy to retrieve the IKM.

- Optionally lock the TPM NVRAM from reading the secret another time, or extend a PCR that prevents re-reading the secret.