Protecting EdgeX Secrets (Edinburgh release)

As of the California release, EdgeX has a standalone micro service secret store (provided by Hashicorp Vault) for centralized management of credentials and sensitive data across EdgeX. While the secret store has been in place, it has not yet been used to store the secrets of the other EdgeX micro services. With the Edinburgh release, that will change as EdgeX secrets will begin to be stored in Vault. Secrets include usernames, passwords, certificates, tokens, etc. used by the micro services.

This sounds straightforward, but can actually be a complex task. How do the secrets get into the secret store? How does the micro service know where/how to get to its secrets and what permission needs to be granted to the micro service so that it has access to retrieve its own secrets? How are secrets organized inside of the secret store?

In this document, we outline the overall design of the EdgeX Edinburgh secure storage usage.

Threat and Purpose

The intention of the work for this release is to avoid having any EdgeX micro service secret stored in plain sight in source control. For example, today, the MongoDB username and password are shown clearly in files like init_mongo.js and clean_mongo.js (see https://github.com/edgexfoundry/developer-scripts/blob/master/init_mongo.js). Usernames, passwords, tokens, etc. could also be found in configuration files stored in source control for many services – importantly for those using the database (see https://github.com/edgexfoundry/edgex-go/blob/master/cmd/core-data/res/docker/configuration.toml as an example).

Inventory of EdgeX Secrets

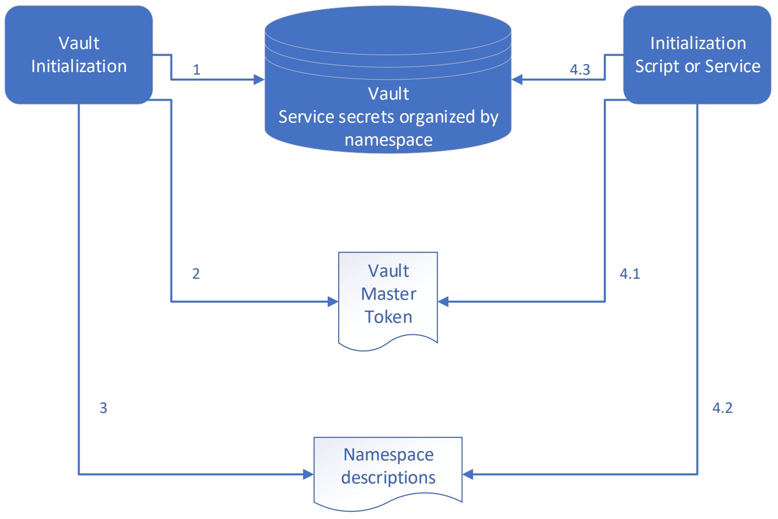
This section includes known and forecast secrets employed by EdgeX.

- Database secrets (username/password) to get access to MongoDB used by core data, metadata, scheduler, notifications, logging, export client and other services that need to store or retrieve data from the persistence store. Redis, today, operates without security access and assumes the database is inaccessible behind the firewall of the application
- Export service (client and distribution) tokens, usernames/passwords to send data to secure endpoints such as Azure IoT, Google IoT Core, a secure HTTPS endpoint, a secure MQTTS topic, etc. In the future, it is presumed that application services will need something similar.
- In the future, it is anticipated that some device services may need to make a secure connection with their associated devices. Such is the case with some BACNet communications. In this case, device services may also require tokens, usernames/passwords, etc. to be securely stored.

Secret Storage Architecture

This section covers the general design of how the secret store is setup, how its credentials are made available to the other EdgeX micro services and how the other services are able to access the secrets given the credentials. The numbered steps below correspond to the steps in the accompanying diagram. Details for each step are provided later in this document.

1. A program stores the secret store access token (currently the master token) in a file on the file system (or shared volume when using containers). So that it can be used by the initialization programs and micro services later (see 4.1 below).
2. The secret store is seeded by an initialization program (this could be the same as the program of #1). This program will create different scopes/namespace for the different EdgeX micro services (see the Organization of Secrets section for more details on this organization). It will also populate the credentials and sensitive data for each micro service – storing the data into the appropriate namespace for each service.
3. The secret store (Vault) offers a REST API interface that allows client micro services or other initialization programs to get secrets from the secret store with an understanding of the applicable namespace (established in #2) and the access token (stored with #1). The namespace is used as part of the REST call URLs to access the appropriate secret from the store. The namespaces for available secrets for each service is shared with each client service (or initialization program) via configuration.
4. When an EdgeX micro service (or initialization script) needs to obtain a credential from the secret store, it needs to perform following steps:
    4.1 Get the access token from file system or volume
    4.2 Get the applicable namespace for the credential
    4.3 Make a REST call to the secret store using the appropriate URL derived from the namespace for the credential along with access token to retrieve the credential.


Vault Initialization
1
Vault Service secrets organized by namespace
4.3
Initialization Script or Service
2
Vault Master Token
4.1
3
Namespace descriptions
4.2

Vault Initialization

The secret store init program (currently written in Go) will be modified to create namespaces for each individual microservice as well as create secrets/credentials used by each service and store them in the namespace. Notably, this includes the creation of the username/password pairs for MongoDB (or another database like Redis).

The namespaces will be defined by configuration to the initialization program. Therefore, the current program will be modified to take command line arguments to specify the needed secrets. The secrets can be passed as environment variables, or command line parameters when the init program starts up. For security enhancement, GUIDs or random strings can be generated within the init program and stored into their individual Vault namespaces without leaving the scope of Vault.

Commented [WJ1]: Containerized strategy??

If the secrets in Vault need to be modified or updated, then the user can use the Vault command line or REST API to make changes to the stored secrets.

Vault Master Token File Protection

Today, the Vault master token is stored, without protection, in the file system. In docker deployments this is a shared volume. In snap deployments this this write-able part of the snap. In non-container or non-snap environments (so called "bare metal"), this is a file stored in the native file system. Note: on a traditional Unix/Linux system this file would be only owner readable via standard MAC.

In the future, this file needs to be protected with a hardware security module (HSM) like TPM, TEE or similar mechanism. The security services for Edinburgh will document this as a vulnerability and recommend the end user otherwise secure this secret.

Organization of Secrets (Namespaces)

The credentials and sensitive data will be organized in the secret store by EdgeX micro service. A different namespace path will be used for each micro service. In general, credentials will be organized under a namespace of /v1/secret/edgex/:path where path is a string that represents the individual micro service. For database initialization, specifically Mongo, the initial username/password pair would be accessed through the path of /v1/secret/edgex/mongodbinit, and a REST API request of the secret store would return JSON like this:

```
{
        “auth”: null,
        “data”: {
                “initusername”: “user”,
                “initpasswd”: “passwd”
},
        “lease_duration”: 3600,
        “lease_id”: “”,
        “renewable”: false
}
```

Where "auth" specifies the authentication method for the specific namespace and "lease_duration" specifies how long the will the data last in second.

As additional examples, core secrets (if it has any) would be in v1/secret/edgex/coredata and a modbus device service secrets (if any) would be in v1/secret/edgex/devicemodbus. Assuming core data needs credentials like username/password pair to access core data MongoDB, and need a GUID to identify itself when sending data to another microservice, an HTTP request to {SECRET-STORE-IP}:8200/v1/secret/edgex/coredata will be given response something like this:

```
{
        “auth”: null,
        “data”: {
                “mongousername”: “user”,
                “mongopasswd”: “passwd”,
                “guid”: ”xxxxxxxxx-xxxxxxxx-xxxxxxxx”
},
        “lease_duration”: 3600,
        “lease_id”: “”,
        “renewable”: false
}
```

Vault provides the means to generate “password” tokens. In order to avoid having to put pre-arranged password secrets in some other config or open file to be used by the initialization program, the initialization program will generate passwords to be store in Vault where required.

Categories of Access Levels

Ideally, and in future releases, the access for each service will be different so that access control and policies can be enforced per service. In the Edinburgh release, we will not distinguish the roles and provide each service with access to master token stored in the file system or volume that is accessed equally by all micro services. Today, all secrets stored and access in secure storage assume the same level of access.

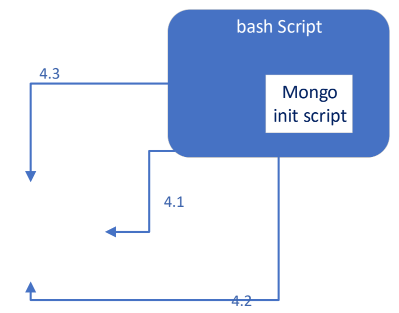
In the future, the following categories of access are projected (with the example of mongodbinit).

```
path "v1/secret/edgex/mongodbinit " {

  capabilities = ["create", "read", "update", "delete", "list"]

}
```

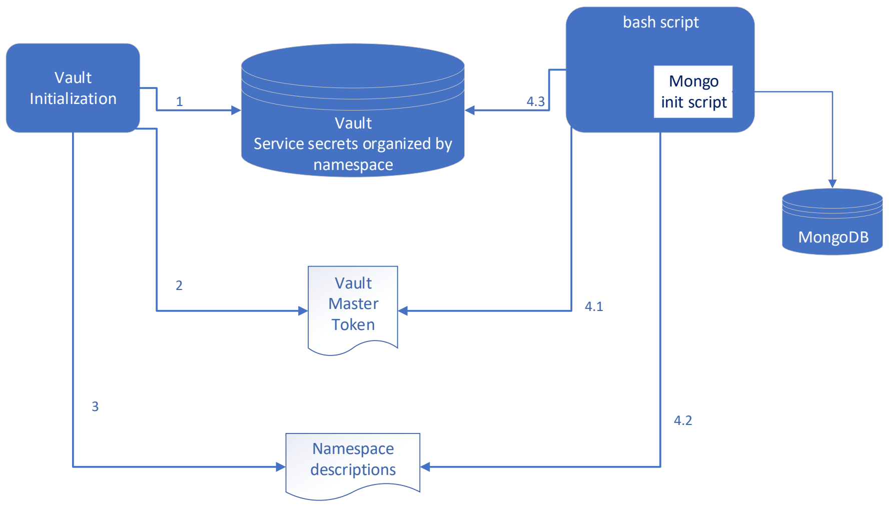
The capabilities will be associated with the access tokens that a client microservice uses to access the credential. A request to the secret will be granted or denied based on the capabilities.

Database Initialization

Mongo initialization is a standalone program and service that creates the MongoDB data structures (and access) for EdgeX micro services (https://github.com/edgexfoundry/docker-edgex-mongo). Similar initialization scripts and services may exist for other database implementations (such as Redis). When the database initialization program executes, it requires an initial access username/password pair to create the database instance. It may, as in the case of MongoDB, also create several collections or structures each requiring other username/password pairs. Typically, there is a separate (and uniquely protected) collection for each EdgeX micro service (such as a collection for coredata, metadata etc.). Currently the initial username/password as well as username/password pairs for micro services are saved in plaintext format in the initialization script. Under the new architecture, the database initialization script will query the secret store to obtain the credentials and use them to perform initialization.



The existing MongoDB initialization program is implemented using JavaScript (mongo_init.js). For Edinburgh, a bash script will get the master token file, use the master token to get the appropriate username/password pairs and the call on a revised initialization script with the pairs to initialize the database and all appropriate collections. In other words, a bash script will perform operations 4.1-4.3 above for retrieving all database username/password pairs and then call to execute the initialization JavaScript with the username/password pairs.



The cleanup script (and any other MongoDB script) will also have to be modified in a similar fashion as it also requires access to the database via username/password.

Micro Service Secrets Retrieval (in Go)

Any credential that is kept in configuration file or other plaintext format currently needs to be moved to secret store, and a code module needs to be created and made available to all micro services needing to retrieve credentials/secrets. The general steps for the Go module will be similar to what is described in the step 4 of “Secret Store Architecture” previously, which is

- get the access token to secret store
- obtain REST API path for the credential
- REST calls to API endpoint to retrieve the credential with access token

Edinburgh Security Work Outlined

To add the features outlined in this document to at least secure the database secrets (username/password pairs) for all consuming micro services, the following tasks will need to be completed:

1. Modify the secret store service initialization (implemented in Golang) to create the initial namespaces and credentials for Mongo.
2. Modify the secret store service initialization to store the master token to a configuration defined file location to the file system or shared volume.
3. Define the secret namespaces and share the namespace definitions with the individual consuming micro service or init scripts. This can be either specified in a configuration file or within Consul (as with other configuration, it should be done by Consul when available or configuration file when Consul is not available).
4. Create a bash script to fetch the master token and access the secret store to fetch the database username/password pairs, and then call the existing database (Mongo) initialization script to properly initialize the databases and their associated collections.
5. Create a "client" Go module to fetch the master token file, get secret namespace information from configuration and access secrets (in particular the database username/password pairs) from the secret store using the token and namespace. The module will serve as a template for other language libraries in the future to access the secret store in the same fashion.