EuroPython

Basel, Switzerland, 08-14 July 2019

Protecting Secrets with Oslo.Config and HashiCorp Vault

Moisés Guimarães de Medeiros - Software Engineer - Red Hat

Introduction

Applications and services rely on configuration in order to interact with each other and to function according to a purpose. *Config files* are widely used to separate configuration from code as configuration varies substantially across deployments while code does not.

OpenStack Common Libraries (Oslo) has an alternative to Python's standard module ConfigParser called *oslo.config*. It supports config through command line arguments, environment variables, config files and has an API for configuration source drivers.

There is also *Castellan*, a generic secret manager interface maintained by Oslo, that aims to enable projects to use deployment specific *secret managers* avoiding vendor lock-in.

The motivation of this work is to show how to use oslo.config to fetch sensitive configuration data (secrets) from places that are better equipped to deal with them.

Problem Description

Best practices say that passwords and other secrets should not be stored as plain text in config files and some regulations even enforce this practice as mandatory.

Although we can rely on file system permissions to restrict access to config files, their content can still be accidentally shared, checked into revision control, or printed to a console or log file, without having all sensitive data stripped out of it.

Also, popularization of containers and public clouds increase the risk of data getting compromised as we lose control over what else runs on the same machine.

There are proper solutions for storing this kind of sensitive data called secret managers. They have features to handle access control, password rotation, data encryption and decryption, X.509 certificates and can also interface with Hardware Security Modules (HSM) if the security requirements are set that high.

oslo.config

A config option in oslo.config is identified by its `name` and `group`, when the group is not defined, the option automatically belongs to the `DEFAULT` group. Below we can see a sample config file with three options and the minimal code required to register them:

```
[DEFAULT]
foo=bar
[db]
user=admin
pswd=AyqUJtFu4Zam
```

```
from oslo_config import cfg
conf = cfg.ConfigOpts()
conf.register_cli_opt(cfg.StrOpt("foo"))
conf.register_opt(cfg.StrOpt("user"), "db")
conf.register_opt(cfg.StrOpt("pswd"), "db")
```

In the example above, values can be accessed at `conf.foo`, `conf.db.user` and `conf.db.pswd` after the option values are loaded with a call to `conf` like `conf()`.

Values from environment variables takes precedence over values from config files. Oslo.config looks for variables in `os.environ` named like `OS_{GROUP}__{NAME}`.

To enable options to be defined via command line arguments, they must register with `register_cli_opt()` so they are exposed to the `cli` as `--{group}-{name}`. Values from the command line arguments have the highest precedence in oslo.config.

For configuration source drivers, a config option DEFAULT.config_source is used to define extra sources. Source drivers are responsible for providing config values when they are not present in: `cli args`, `env variables` and `config files`.

Oslo.config will check for defined option values using the following order of precedence:

1. Checks the `command line arguments` in case the option was exposed to the `cli`.
   - For `conf.foo`, looks for an argument named `--foo` as `DEFAULT` is omitted for `cli`.
2. Checks for an `environment variable` with the pattern `OS_{GROUP}__{NAME}`.
   - For `conf.foo`, it will look for a variable named `OS_DEFAULT__FOO`.
3. Checks if the option was defined in one of the loaded `config files`.
4. Asks the `configuration sources` in the order they appear in `config_source`.
5. Returns `None` or the `default` value in case the config option has one.

Proposed Solution

To minimize impact on codebases, we used the `source drivers API` to implement the `castellan` driver. This driver is responsible for retrieving config values from secret managers supported by Castellan and has the following options:

`driver`=castellan

The name of the driver that can load this configuration source.

`config_file`

A config file required by Castellan to talk to a secret manager.

`mapping_file`

A pseudo config file that maps options to secret_ids instead of values. The driver uses secret_ids in Castellan to fetch option values stored in the secret manager.

The proof of concept (PoC) for this work aimed to fulfill the following criteria:

1. Have a sample application fetching config values through oslo.config;
2. Strip out secrets from its config file;
3. Store those secrets in a `secret manager`;
4. Have the sample application to fetch secrets from the `secret manager` without any change to its code.

The complete code for the PoC can be found at:

https://github.com/moisesguimaraes/ep19

Conclusions

By fetching option values through the castellan driver, one can include extra layers of protection and take advantage of features provided by the secret manager in use.

We were able to generate temporary credentials for multiple nodes being able to revoke access to a single node without compromising the whole deployment. Also, the lifespan of a secret can be fine-tuned to minimize the attack window using compromised keys and credentials.

Having the secrets to live in a single place makes the process of credentials rotation easier as the new credentials only has to be updated in one place instead of in every config files that requires it.

Future Work

With the proper use of policies, we think that orchestrators could trigger secrets generation whenever spawning new nodes, having unique cryptographic keys and credentials per node.

The next step in our roadmap is to integrate this work into *TripleO*, a program aimed at installing, upgrading and operating OpenStack clouds using OpenStack's own cloud facilities as the foundations to automate fleet management at datacenter scale.

Developers interested in using another solution as an extra source of configuration data can implement their own drivers following this spec:

https://specs.openstack.org/openstack/oslo-specs/specs/queens/oslo-config-drivers

PoC Scenarios

BEFORE

The sample app is using a config file to load database credentials. This file lives within the deployment of the application and to scale it needs to run in a large number of machines, making copies of the secrets all over the cloud.

```
file: web_plaintext.conf
   [app]
   port=5001
   [db]
   hostname=localhost
   username=postgres
   password=changeme

cli call:
   python app.py \
   --config-file=web_plaintext.conf
```

AFTER

After identifying the secrets in the configuration file we setup a HashiCorp Vault instance behind Castellan and moved the secrets to it under an arbitrary access token.

The app's config file is updated with the `castellan` driver settings to be able to reach the secrets.

We created both a castellan.conf and a mapping.conf with the secret's ids for the `castellan` driver config.

The app now is launched with the Vault token being provided via `environment variables`, stripping out any kind of secrets in the plaintext config files.

Each instance of the app has its own Vault token and its own credentials to the database, making it possible to instantly revoke database access for a single node.

```
file: web_vault.conf
   [DEFAULT]
   config_source=castellan
   [castellan]
   driver=castellan
   config_file=castellan.conf
   mapping_file=mapping.conf
   [app]
   port=5002
   [db]
   host=localhost

file: castellan.conf
   [key_manager]
   backend=vault
   [vault]
   #token=

file: mapping.conf
   [db]
   username=3846d164d5a146eb695637bec4b3
   password=c2aa215115598549e22dfb52e82a

cli call:
   OS_VAULT__ROOT_TOKEN_ID=\
   s.FSUHTpovzLBhu4ENA5evXWdC \
   python app.py \
   --config-file=web_vault.conf
```

OpenStack

"OpenStack is a set of software tools for building and managing cloud computing platforms for public and private clouds. Backed by some of the biggest companies in software development and hosting, as well as thousands of individual community members"

https://openstack.org

Oslo

"The Oslo project is a collection of over 30 libraries that are designed to reduce the technical debt of code duplication across projects and provide for a greater quality code path due to the frequency of use in OpenStack projects."

https://openstack.org

Barbican

"Barbican is the OpenStack Key Manager service. It provides secure storage, provisioning and management of secret data, such as passwords, encryption keys, X.509 Certificates and raw binary data."

https://www.openstack.org

HashiCorp Vault

"HashiCorp Vault secures, stores, and tightly controls access to tokens, passwords, certificates, API keys, and other secrets in modern computing. Vault handles leasing, key revocation, key rolling, and auditing."

https://www.vaultproject.io