castellan Documentation

Release 5.3.0.dev5

OpenStack Foundation

Apr 15, 2025

CONTENTS

Generic Key Manager interface for OpenStack.

- License: Apache License, Version 2.0
- Documentation: https://docs.openstack.org/castellan/latest
- Source: https://opendev.org/openstack/castellan
- Bugs: https://bugs.launchpad.net/castellan
- Release notes: https://docs.openstack.org/releasenotes/castellan
- Wiki: https://wiki.openstack.org/wiki/Castellan

CHAPTER

ONE

TEAM AND REPOSITORY TAGS

CHAPTER

TWO

CONTENTS

2.1 Installation

At the command line:

```
$ pip install castellan
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv castellan
$ pip install castellan
```

2.2 Usage

This document describes some of the common usage patterns for Castellan. When incorporating this package into your applications, care should be taken to consider the key manager behavior you wish to encapsulate and the OpenStack deployments on which your application will run.

2.2.1 Authentication

A fundamental concept to using Castellan is the credential context object. Castellan supports the following credentials for authentication:

- Token
- Password
- Keystone Token
- Keystone Password

In order to use these credentials, valid configuration parameters must be provided.

```
# token credential
# token variable not required, token can be obtained from context
[key_manager]
auth_type = 'token'
token = '5b4de0bb77064f289f7cc58e33bea8c7'

# password credential
[key_manager]
auth_type = 'password'
```

(continues on next page)

(continued from previous page)

```
username = 'admin'
password = 'passw0rd1'

# keystone token credential
[key_manager]
auth_url = 'http://192.169.5.254:5000'
auth_type = 'keystone_token'
token = '5b4de0bb77064f289f7cc58e33bea8c7'
project_id = 'a1e19934af81420d980a5d02b4afe9fb'

# keystone password credential
[key_manager]
auth_url = 'http://192.169.5.254:5000'
auth_type = 'keystone_password'
username = 'admin'
password = 'passw0rd1'
project_id = '1099302ec608486f9879ba2466c60720'
user_domain_name = 'default'
```

Note

Keystone Token and Password authentication is achieved using keystoneauth1.identity Token and Password auth plugins. There are a variety of different variables which can be set for the keystone credential options.

The configuration must be passed to a credential factory which will generate the appropriate context.

```
from castellan.common import utils

CONF = cfg.CONF
CONF(default_config_files=['~/castellan.conf'])
context = utils.credential_factory(conf=CONF)
```

Now you can go ahead and pass the context and use it for authentication.

Note

There is a special case for a token. Since a user may not want to store a token in the configuration, the user can pass a context object containing an auth_token as well as a configuration file with token as the auth type.

An oslo context object can also be used for authentication, it is frequently inherited from `oslo.context.RequestContext`. This object represents information that is contained in the current request, and is usually populated in the WSGI pipeline. The information contained in this object will be used by Castellan to interact with the specific key manager that is being abstracted.

Example. Creating RequestContext from Keystone Client

```
from keystoneauth1 import identity
from keystoneauth1 import session
from oslo_context import context

username = 'admin'
password = 'openstack'
project_name = 'admin'
auth_url = 'http://localhost/identity/v3'
auth = identity.Password(auth_url=auth_url,
                         username=username,
                         password=password,
                         project_name=project_name,
                         default_domain_id='default')
sess = session.Session()

ctxt = context.RequestContext(auth_token=auth.get_token(sess),
                              project_id=auth.get_project_id(sess))
```

ctxt can then be passed into any key_manager api call.

2.2.2 Basic usage

Castellan works on the principle of providing an abstracted key manager based on your configuration. In this manner, several different management services can be supported through a single interface.

In addition to the key manager, Castellan also provides primitives for various types of secrets (for example, asymmetric keys, simple passphrases, and certificates). These primitives are used in conjunction with the key manager to create, store, retrieve, and destroy managed secrets.

Example. Creating and storing a key.

```
import myapp
from castellan.common.objects import passphrase
from castellan import key_manager

key = passphrase.Passphrase('super_secret_password')
manager = key_manager.API()
stored_key_id = manager.store(myapp.context(), key)
```

To begin with, wed like to create a key to manage. We create a simple passphrase key, then instantiate the key manager, and finally store it to the manager service. We record the key identifier for later usage.

Example. Retrieving a key and checking the contents.

```
import myapp
from castellan import key_manager

manager = key_manager.API()
key = manager.get(myapp.context(), stored_key_id)
if key.get_encoded() == 'super_secret_password':
    myapp.do_secret_stuff()
```

This example demonstrates retrieving a stored key from the key manager service and checking its con-

tents. First we instantiate the key manager, then retrieve the key using a previously stored identifier, and finally we check the validity of key before performing our restricted actions.

Example. Deleting a key.

```
import myapp
from castellan import key_manager

manager = key_manager.API()
manager.delete(myapp.context(), stored_key_id)
```

Having finished our work with the key, we can now delete it from the key manager service. We once again instantiate a key manager, then we simply delete the key by using its identifier. Under normal conditions, this call will not return anything but may raise exceptions if there are communication, identification, or authorization issues.

Example. Secret consumers.

```
import myapp
from castellan import key_manager

manager = key_manager.API()

# Listing consumers:
stored_secret = self.key_mgr.get(myapp.context(), stored_id)
consumer_list = stored_secret.consumers  # consumers is a list of dicts

# Adding consumers:
consumer = {'service': 'glance',
            'resource_type': 'image',
            'resource_id': 'image_id'}
try:
    manager.add_consumer(myapp.context(), stored_id, consumer)
except NotImplementedError:
    pass  # backends like Vault don't support adding/removing consumers

# Remove the consumer before calling secret delete without the force flag:
try:
    manager.remove_consumer(myapp.context(), stored_id, consumer)
except NotImplementedError:
    pass
manager.delete(myapp.context(), stored_key_id)

# Alternatively, force delete a secret
manager.delete(myapp.context(), stored_key_id, force=True)
```

After creating a secret, we can add consumers to it. Secrets with consumers cannot be deleted without using the force flag.

> **Note**
>
> Secret consumers are currently only avaliable for the Barbican backend. https://docs.openstack.org/

barbican/latest/api/reference/secret_consumers.html

2.2.3 Configuring castellan

Castellan contains several options which control the key management service usage and the configuration of that service. It also contains functions to help configure the defaults and produce listings for use with the `oslo-config-generator` application.

In general, castellan configuration is handled by passing an `oslo_config.cfg.ConfigOpts` object into the `castellan.key_manager.API` call when creating your key manager. By default, when no `ConfigOpts` object is provided, the key manager will use the global `oslo_config.cfg.CONF` object.

Example. Using the global CONF object for configuration.

```
from castellan import key_manager

manager = key_manager.API()
```

Example. Using a predetermined configuration object.

```
from oslo_config import cfg
from castellan import key_manager

conf = cfg.ConfigOpts()
manager = key_manager.API(configuration=conf)
```

Controlling default options

To change the default behavior of castellan, and the key management service it uses, the `castellan.options` module provides the `set_defaults` function. This function can be used at run-time to change the behavior of the library or the key management service provider.

Example. Changing the barbican endpoint.

```
from oslo_config import cfg
from castellan import options
from castellan import key_manager

conf = cfg.ConfigOpts()
options.set_defaults(conf, barbican_endpoint='http://192.168.0.1:9311/')
manager = key_manager.API(conf)
```

Example. Changing the key manager provider while using the global configuration.

```
from oslo_config import cfg
from castellan import options
from castellan import key_manager

options.set_defaults(cfg.CONF, api_class='some.other.KeyManager')
manager = key_manager.API()
```

Logging from within Castellan

Castellan uses `oslo_log` for logging. Log information will be generated if your application has configured the `oslo_log` module. If your application does not use `oslo_log` then you can enable default logging using `enable_logging` in the `castellan.options` module.

Example. Enabling default logging.

```
from castellan import options
from castellan import key_manager

options.enable_logging()
manager = key_manager.API()
```

Generating sample configuration files

Castellan includes a tox configuration for creating a sample configuration file. This file will contain only the values that will be used by castellan. To produce this file, run the following command from the root of the castellan project directory:

```
$ tox -e genconfig
```

Parsing the configuration files

Castellan does not parse the configuration files by default. When you create the files and occupy them, you still need to manipulate the `oslo_config.cfg` object before passing it to the `castellan.key_manager.API` object. You can create a list of locations where the configuration files reside. If multiple configuration files are specified, the variables will be used from the most recently parsed file and overwrite any previous variables. In the example below, the configuration file in the `/etc/castellan` directory will overwrite the values found in the file in the users home directory. If a file is not found in one of the specified locations, then a config file not found error will occur.

Example. Parsing the config files.

```
from oslo_config import cfg
from castellan import key_manager

conf=cfg.CONF
config_files = ['~/castellan.conf', '/etc/castellan/castellan.conf']
conf(default_config_files=config_files)
manager = key_manager.API(configuration=conf)
```

There are two options for parsing the Castellan values from a configuration file:

- The values can be placed in a separate file.
- You can include the values in a configuration file you already use.

In order to see all of the default values used by Castellan, generate a sample configuration by referring to the section directly above.

Adding castellan to configuration files

One common task for OpenStack projects is to create project configuration files. Castellan provides a `list_opts` function in the `castellan.options` module to aid in generating these files when using the `oslo-config-generator`. This function can be specified in the `setup.cfg` file of your project to inform oslo of the configuration options. *Note, this will use the default values supplied by the castellan package.*

Example. Adding castellan to the oslo.config entry point.

```
[entry_points]
oslo.config.opts =
    castellan.config = castellan.options:list_opts
```

The new namespace also needs to be added to your projects oslo-config-generator conf, e.g. *etc/oslo-config-generator/myproject.conf*:

```
[DEFAULT]
output_file = etc/myproject/myproject.conf
namespace = castellan.config
```

For more information on the oslo configuration generator, please see https://docs.openstack.org/oslo.config/latest/cli/generator.html

2.3 Contributor Document

2.3.1 Contributing

The best way to join the community and get involved is to talk with others online or at a meetup and offer contributions. Here are some of the many ways you can contribute to the Castellan project:

- Development and Code Reviews
- Bug reporting/Bug fixes
- Wiki and Documentation
- Blueprints/Specifications
- Testing
- Deployment scripts

Before you start contributing take a look at the Openstack Developers Guide.

OFTC IRC (Chat)

You can find Castellaneers in our publicly accessible channel on OFTC `#openstack-barbican`. All conversations are logged and stored for your convenience at eavesdrop.openstack.org. For more information regarding OpenStack IRC channels please visit the OpenStack IRC Wiki.

Launchpad

Like other OpenStack related projects, we utilize Launchpad for our bug and release tracking.

- Castellan Launchpad Project

Note

Bugs should be filed on Launchpad, not Github.

Source Repository

Like other OpenStack related projects, the official Git repository is available on Castellan on GitHub.

Gerrit

Like other OpenStack related projects, we utilize the OpenStack Gerrit review system for all code reviews. If youre unfamiliar with using the OpenStack Gerrit review system, please review the Gerrit Workflow wiki documentation.

Note

Pull requests submitted through GitHub will be ignored.

2.3.2 Testing

Every Castellan code submission is automatically tested against a number of gating jobs to prevent regressions. Castellan developers should have a habit of running tests locally to ensure the code works as intended before submission.

For your convenience we provide the ability to run all tests through the `tox` utility. If you are unfamiliar with tox please see refer to the tox documentation for assistance.

Unit Tests

Currently, we provide tox environments for a variety of different Python versions. By default all available test environments within the tox configuration will execute when calling `tox`. If you want to run an independent version, you can do so with the following command:

```
# Executes tests on Python 2.7
$ tox -e py27
```

Note

Other available environments are py35 and pep8.

If you do not have the appropriate Python versions available, consider setting up PyEnv to install multiple versions of Python. See the documentation regarding Setting up a Barbican development environment for more information.

Functional Tests

Unlike running unit tests, the functional tests require Barbican and Keystone services to be running in order to execute. For more information on this please see Setting up a Barbican development environment and Using Keystone Middleware with Barbican

Castellan uses either `/etc/castellan/castellan-functional.conf` or `./etc/castellan/castellan-functional.conf` in order to run functional tests. A sample file can be generated by running:

```
# Generate a sample configuration file
$ tox -e genconfig
```

`castellan/etc/castellan/castellan-functional.conf.sample` is generated. It must be renamed to `castellan-functional.conf` and placed in `/etc/castellan` or `./etc/castellan`.

The file should look something like the following:

```
[DEFAULT]

[identity]
username = 'admin'
password = 'openstack'
project_name = 'admin'
auth_url = 'http://localhost/identity/v3'
```

Once you have the appropriate services running and configured you can execute the functional tests through tox.

```
# Execute Barbican Functional Tests
$ tox -e functional
```

By default, the functional tox job will use `testr` to execute the functional tests.

Debugging

In order to be able to debug code in Castellan, you must use the Python Debugger. This can be done by adding `import pdb; pdb.set_trace()` to set the breakpoint. Then run the following command to hit the breakpoint:

```
# hit the pdb breakpoint
$ tox -e debug
```

Once in the Python Debugger, you can use the commands as stated in the *Debugger Commands* section here: https://docs.python.org/2/library/pdb.html

Pep8 Check

Pep8 is a style guide for Python code. Castellan code should be have proper style before submission. In order to ensure that pep8 tests can be run through tox as follows:

```
# Checks python code style
$ tox -e pep8
```

Any comments on bad coding style will output to the terminal.