Patrole: Tempest Plugin for RBAC Testing

Release 0.14.1.dev2

OpenStack Foundation

Apr 25, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

1.1 Patrole - RBAC Integration Tempest Plugin

Patrole is a set of integration tests to be run against a live OpenStack cluster. It has a battery of tests dedicated to validating the correctness and integrity of the clouds RBAC implementation.

More importantly, Patrole is a security validation tool for verifying that Role-Based Access Control is correctly configured and enforced in an OpenStack cloud. It runs Tempest-based API tests using specified RBAC roles, thus allowing deployments to verify that only intended roles have access to those APIs.

Patrole is currently undergoing heavy development. As more projects move toward policy in code, Patrole will align its testing with the appropriate documentation.

- Free software: Apache license
- Documentation: https://docs.openstack.org/patrole/latest
- Source: https://opendev.org/openstack/patrole
- Bugs: https://storyboard.openstack.org/#!/project/openstack/patrole
- Release notes: https://docs.openstack.org/releasenotes/patrole/

1.1.1 Team and repository tags

1.1.2 Design Principles

As a Tempest plugin, Patrole borrows some design principles from Tempest design principles, but not all, as its testing scope is confined to policies.

- *Stability*. Patrole uses OpenStack public interfaces. Tests in Patrole should only touch public OpenStack APIs.
- *Atomicity*. Patrole tests should be atomic: they should test policies in isolation. Unlike Tempest, a Patrole test strives to only call a single endpoint at a time. This is because it is important to validate each policy is authorized correctly and the best way to do that is to validate each policy alone, to avoid test contamination.
- *Complete coverage*. Patrole should validate all policy in code defaults. For testing, Patrole uses the API-to-policy mapping contained in each projects policy in code documentation where applicable.

For example, Novas policy in code documentation is located in the Nova repository under `nova/policies`. Likewise, Keystones policy in code documentation is located in the Keystone repository under `keystone/common/policies`. The other OpenStack services follow the same directory layout pattern with respect to policy in code.

Note: Realistically this is not always possible because some services have not yet moved to policy in code.

- *Customizable*. Patrole should be able to validate custom policy overrides to ensure that those overrides enhance rather than undermine the clouds RBAC configuration. In addition, Patrole should be able to validate any role.
- *Self-cleaning*. Patrole should attempt to clean up after itself; whenever possible we should tear down resources when done.

Note: Patrole modifies roles dynamically in the background, which affects pre-provisioned credentials. Work is currently underway to clean up modifications made to pre-provisioned credentials.

- *Self-testing*. Patrole should be self-testing.

1.1.3 Features

- Validation of default policy definitions located in policy.json files.
- Validation of in-code policy definitions.
- Validation of custom policy file definitions that override default policy definitions.
- Built-in positive and negative testing. Positive and negative testing are performed using the same tests and role-switching.
- Valdation of custom roles as well as default OpenStack roles.

Note: Patrole does not yet support policy.yaml files, the new file format for policy files in OpenStack.

1.1.4 How It Works

Patrole leverages `oslo.policy` (OpenStacks policy enforcement engine) to determine whether a given role is allowed to perform a policy action, given a specific role and OpenStack service. The output from `oslo.policy` (the expected result) and the actual result from test execution are compared to each other: if both results match, then the test passes; else it fails.

Terminology

- Expected Result - The expected result of a given test.
- Actual Result - The actual result of a given test.
- Final Result - A match between both expected and actual results. A mismatch in the expected result and the actual result will result in a test failure.
  - Expected: Pass | Actual: Pass - Test Case Success
  - Expected: Pass | Actual: Fail - Test Case Under-Permission Failure
  - Expected: Fail | Actual: Pass - Test Case Over-Permission Failure
  - Expected: Fail | Actual: Fail (Expected exception) - Test Case Success
  - Expected: Fail | Actual: Fail (Unexpected exception) - Test Case Failure

1.1.5 Quickstart

To run Patrole, you must first have Tempest installed and configured properly. Please reference Tempest_quickstart guide to do so. Follow all the steps outlined therein. Afterward, proceed with the steps below.

1. You first need to install Patrole. This is done with pip after you check out the Patrole repo:

   ```
   $ git clone https://opendev.org/openstack/patrole
   $ pip install patrole/
   ```

   This can be done within a venv.

   Note: You may also install Patrole from source code by running:

   ```
   pip install -e patrole/
   ```

   ---

2. Next you must properly configure Patrole, which is relatively straightforward. For details on configuring Patrole refer to the Patrole Configuration.
3. Once the configuration is done youre now ready to run Patrole. This can be done using the tempest_run command. This can be done by running:

   ```
   $ tempest run --regex '^patrole_tempest_plugin\.tests\.api'
   ```

   There is also the option to use testr directly, or any testr based test runner, like ostestr. For example, from the workspace dir run:

   ```
   $ stestr --regex '(?!.*\[.*\bslow\b.*\])(^patrole_tempest_plugin\.tests\.
   ↪api))'
   ```

   will run the same set of tests as the default gate jobs.

   You can also run Patrole tests using tox, but as Patrole needs access to global packages use `--sitepackages` argument. To do so, `cd` into the **Tempest** directory and run:

```
$ tox -eall --sitepackages -- patrole_tempest_plugin.tests.api
```

Note: It is possible to run Patrole via `tox -eall` in order to run Patrole isolated from other plugins. This can be accomplished by including the installation of services that currently use policy in code for example, Nova and Keystone. For example:

```
$ tox -evenv-tempest -- pip install /opt/stack/patrole /opt/stack/
↪keystone /opt/stack/nova
$ tox -eall -- patrole_tempest_plugin.tests.api
```

4. Log information from tests is captured in `tempest.log` under the Tempest repository. Some Patrole debugging information is captured in that log related to expected test results and Role Overriding.

   More detailed RBAC testing log output is emitted to `patrole.log` under the Patrole repository. To configure Patroles logging, see the Patrole Configuration Guide.

1.1.6 RBAC Tests

To change the roles that the patrole tests are being run as, edit `rbac_test_roles` in the `patrole` section of tempest.conf:

```
[patrole]
rbac_test_roles = member,reader
...
```

Note: The `rbac_test_roles` is service-specific. member, for example, is an arbitrary role, but by convention is used to designate the default non-admin role in the system. Most Patrole tests should be run with **admin** and **member** roles. However, other services may use entirely different roles or role combinations.

For more information about RBAC, reference the rbac-overview documentation page.

For information regarding which projects Patrole offers RBAC testing for, reference the HACKING documentation page.

1.1.7 Unit Tests

Patrole also has a set of unit tests which test the Patrole code itself. These tests can be run by specifying the test discovery path:

```
$ stestr --test-path ./patrole_tempest_plugin/tests/unit run
```

By setting `--test-path` option to `./patrole_tempest_plugin/tests/unit` it specifies that test discovery should only be run on the unit test directory.

Alternatively, there are the py27 and py35 tox jobs which will run the unit tests with the corresponding version of Python.

One common activity is to just run a single test; you can do this with tox simply by specifying to just run py27 or py35 tests against a single test:

```
$ tox -e py27 -- -n patrole_tempest_plugin.tests.unit.test_rbac_utils.
↪RBACUtilsTest.test_override_role_with_missing_admin_role
```

Or all tests in the test_rbac_utils.py file:

```
$ tox -e py27 -- -n patrole_tempest_plugin.tests.unit.test_rbac_utils
```

You may also use regular expressions to run any matching tests:

```
$ tox -e py27 -- test_rbac_utils
```

For more information on these options and details about stestr, please see the stestr documentation.

1.1.8 Release Versioning

Patrole Release Notes shows which changes have been released for each version.

Patroles release versioning follows Tempests conventions. Like Tempest, Patrole is branchless and uses versioning instead.

1.1.9 Storyboard

Bugs and enhancements are tracked via Patroles Storyboard Page.

1.2 RBAC Overview

1.2.1 Role-Based Access Control Overview

Introduction

Role-Based Access Control (RBAC) is used by most OpenStack services to control user access to resources. Authorization is granted if a user has the necessary role to perform an action. Patrole is concerned with validating that each of these resources *can* be accessed by authorized users and *cannot* be accessed by unauthorized users.

OpenStack services use oslo.policy as the library for RBAC authorization. Patrole relies on the same library for deriving expected test results.

Policy in Code

Services publish their policy-to-API mapping via policy in code documentation. This mapping includes the list of APIs that authorize a policy, for each policy declared within a service.

For example, Novas policy in code documentation is located in the Nova repository under `nova/policies`. Likewise, Keystones policy in code documentation is located in the Keystone repository under `keystone/common/policies`. The other OpenStack services follow the same directory layout pattern with respect to policy in code.

The policy in code governance goal enumerates many advantages with following this RBAC design approach. A so-called library of in-code policies offers the following advantages, with respect to facilitating validation:

- includes every policy enforced by an OpenStack service, enabling the possibility of complete Patrole test coverage for that service (otherwise one has to read the source code to discover all the policies)
- provides the policy-to-API mapping for each policy which can be used to write correct Patrole tests (otherwise reading source code and experimentation are required to derive this mapping)
- by extension, the policy-to-API mapping facilitates writing multi-policy Patrole tests (otherwise even more experimentation and code reading is required to arrive at all the policies enforced by an API)
- policy in code documentation includes additional information, like descriptions and (in the case of some services, like Keystone) scope types, which help with understanding how to correctly write Patrole tests
- by extension, such information helps to determine whether a Patrole test should assume *hard authorization* or *soft authorization*

Policy in Code (Default) Validation

By default, Patrole validates default OpenStack policies. This is so that the out-of-the-box defaults are sanity-checked, to ensure that OpenStack services are secure, from an RBAC perspective, for each release.

Patrole strives to validate RBAC by using the policy in code documentation, wherever possible. See *Validation Workflow Overview* for more details.

Custom Policies

Operators can override policy in code defaults using policy.yaml. While this allows operators to offer more fine-grained RBAC control to their tenants, it opens the door to misconfiguration and bugs. Patrole can be used to validate that custom policy overrides dont break anything and work as expected.

Custom Policy Validation

While testing default policy behavior is a valid use case, oftentimes default policies are modified with custom overrides in production. OpenStacks policy.yaml documentation claims that modifying policy can have unexpected side effects, which is why Patrole was created: to ensure that custom overrides allow the principle of least privilege to be tailor-made to exact specifications via policy overrides, without:

- causing unintended side effects (breaking API endpoints, breaking cross-service workflows, breaking the policy file itself); or
- resulting in poor RBAC configuration, promoting security vulnerabilities

This has implications on Patroles *Design Principles*: validating custom overrides requires the ability to handle arbitrary roles, which requires logic capable of dynamically determining expected test behavior.

Note that support for custom policies is limited. This is because custom policies can be arbitrarily complex, requiring that tests be very robust in order to handle all edge cases.

Multiple Policies

Behind the scenes, many APIs enforce multiple policies, for many reasons, including:

- to control complex cross-service workflows;
- to control whether a server is booted from an image or booted from a volume (for example);
- to control whether a response body should contain additional information conditioned upon successful policy authorization.

This makes *policy in code* especially important for policy validation: it is difficult to keep track of all the policies being enforced across all the individual APIs, without policy in code documentation.

Multi-Policy Validation

Patrole offers support for validating APIs that enforce multiple policies. Perhaps in an ideal world each API endpoint would enforce only one policy, but in reality some API endpoints enforce multiple policies. Thus, to offer accurate validation, Patrole handles multiple policies:

- for services *with* policy in code documentation: this documentation indicates that a single API endpoint enforces multiple policy actions.
- for services *without* policy in code documentation: the API code clearly shows multiple policy actions being validated. Note that in this case some degree of log tracing is required by developers to confirm that the expected policies are getting enforced, prior to the tests getting merged.

For more information, see *Multi-policy Validation*.

Error Codes

Most OpenStack services raise a `403 Forbidden` following failed *hard authorization*. Neutron, however, can raise a `404 NotFound` as well. See Neutrons authorization policy enforcement documentation for more details.

Admin Context Policy

The so-called admin context policy refers to the following policy definition (using the legacy policy file syntax):

```
{
  "context_is_admin": "role:admin"
  ...
}
```

Which is unfortunately used to bypass `oslo.policy` authorization checks, for example:

```
# This function is responsible for calling oslo.policy to check whether
# requests are authorized to perform an API action.
def enforce(context, action, target, [...]):
  # Here this condition, if True, skips over the enforce call below which
  # is what calls oslo.policy.
  if context.is_admin:
      return True
  _ENFORCER.enforce([...])  # This is what can be skipped over.
  [...]
```

This type of behavior is currently present in many services. Unless such logic is removed in the future for services that implement it, Patrole wont really be able to validate that admin role works from an `oslo.policy` perspective.

Glossary

The following nomenclature is used throughout Patrole documentation so it is important to understand what each term means in order to understand concepts related to RBAC in Patrole.

authorize The act of `oslo.policy` determining whether a user can perform a *policy* given his or her *role*.

enforce See *authorize*.

hard authorization The do_raise flag controls whether policy authorization should result in an exception getting raised or a boolean value getting returned. Hard authorization results in an exception getting raised. Usually, this results in a `403 Forbidden` getting returned for unauthorized requests. (See *Error Codes* for further details.)

Related term: *soft authorization*.

oslo.policy The OpenStack library providing support for RBAC policy enforcement across all OpenStack services. See the official documentation for more information.

policy Defines an RBAC rule. Each policy is defined by a one-line statement in the form <target> : <rule>. For more information, reference OpenStacks policy documentation.

policy action See *policy target*.

policy file Prior to governance goal used by all OpenStack services to define policy defaults. Still used by some services, which is why Patrole needs to read the policy files to derive policy information for testing.

policy in code Registers default OpenStack policies for a service in the services code base.

Beginning with the Queens release, policy in code became a governance goal.

policy rule The policy rule determines under which circumstances the API call is permitted.

policy target The name of a policy.

requirements file Requirements-driven approach to declaring the expected RBAC test results referenced by Patrole. Uses a high-level YAML syntax to crystallize policy requirements concisely and unambiguously. See *Requirements Authority Module* for more information.

role A designation for the set of actions that describe what a user can do in the system. Roles are managed through the Keystone Roles API.

Role-Based Access Control (RBAC) May be formally defined as an approach to restricting system access to authorized users.

rule See *policy rule*. Note that currently the Patrole code base conflates rule with *policy target* in some places.

soft authorization The do_raise flag controls whether policy authorization should result in an exception getting raised or a boolean value getting returned. Soft authorization results in a boolean value getting returned. When policy authorization evaluates to true, additional operations are performed as a part of the API request or additional information is included in the response body (see response filtering for an example).

Related term: *hard authorization*.

1.2.2 Multi-policy Validation

Introduction

Multi-policy validation exists in Patrole because if one policy were assumed, then tests could fail because they would not consider all the policies actually being enforced. The reasoning can be found in this spec. Basically, since Patrole derives the expected test result dynamically in order to test any role, each policy enforced by the API under test must be considered to derive an accurate expected test result, or else the expected and actual test results will not always match, resulting in overall test failure. For more information about Patroles RBAC validation work flow, reference *RBAC Rule Validation Module*.

Multi-policy support allows Patrole to more accurately offer RBAC tests for API endpoints that enforce multiple policy actions.

Scope

Multiple policies should be applied only to tests that require them. Not all API endpoints enforce multiple policies. Some services consistently enforce 1 policy per API, while on the other side of the spectrum, services like Neutron have much more involved policy enforcement work flows. See *Neutron Multi-policy Validation* for more information.

Neutron Multi-policy Validation

Neutron can raise different *Error Codes* following failed policy authorization. Many endpoints in Neutron enforce multiple policies, which complicates matters when trying to determine whether the endpoint raises a 403 or a 404 following unauthorized access.

Multi-policy Examples

General Examples

Below is an example of multi-policy validation for a carefully chosen Nova API:

```
@rbac_rule_validation.action(
service="nova",
rules=["os_compute_api:os-lock-server:unlock",
       "os_compute_api:os-lock-server:unlock:unlock_override"])
@decorators.idempotent_id('40dfeef9-73ee-48a9-be19-a219875de457')
def test_unlock_server_override(self):
    """Test force unlock server, part of os-lock-server.

    In order to trigger the unlock:unlock_override policy instead
    of the unlock policy, the server must be locked by a different
    user than the one who is attempting to unlock it.
    """
    self.os_admin.servers_client.lock_server(self.server['id'])
    self.addCleanup(self.servers_client.unlock_server, self.server['id'])

    with self.override_role():
        self.servers_client.unlock_server(self.server['id'])
```

While the `expected_error_codes` parameter is omitted in the example above, Patrole automatically populates it with a 403 for each policy in `rules`. Therefore, in the example above, the following expected error codes/rules relationship is observed:

- os_compute_api:os-lock-server:unlock => 403
- os_compute_api:os-lock-server:unlock:unlock_override => 403

Below is an example that uses `expected_error_codes` to account for the fact that Neutron is expected to raise a `404` on the first policy that is enforced server-side (get_port). Also, in this example, soft authorization is performed, meaning that it is necessary to check the response body for an attribute that is added only following successful policy authorization.

```
@utils.requires_ext(extension='binding', service='network')
@rbac_rule_validation.action(service="neutron",
                             rules=["get_port",
                                    "get_port:binding:vif_type"],
                             expected_error_codes=[404, 403])
@decorators.idempotent_id('125aff0b-8fed-4f8e-8410-338616594b06')
def test_show_port_binding_vif_type(self):

    # Verify specific fields of a port
    fields = ['binding:vif_type']

    with self.override_role():
        retrieved_port = self.ports_client.show_port(
            self.port['id'], fields=fields)['port']

    # Rather than throwing a 403, the field is not present, so raise exc.
    if fields[0] not in retrieved_port:
        raise rbac_exceptions.RbacMalformedResponse(
            attribute='binding:vif_type')
```

Note that in the example above, failure to authorize get_port:binding:vif_type results in the response body getting successfully returned by the server, but without additional dictionary keys. If Patrole fails to find those expected keys, it *acts as though* a 403 was thrown (by raising an exception itself, the `rbac_rule_validation` decorator handles the rest).

Neutron Examples

A basic Neutron example that only expects 403s to be raised:

```
@utils.requires_ext(extension='external-net', service='network')
@rbac_rule_validation.action(service="neutron",
                             rules=["create_network",
                                    "create_network:router:external"],
                             expected_error_codes=[403, 403])
@decorators.idempotent_id('51adf2a7-739c-41e0-8857-3b4c460cbd24')
def test_create_network_router_external(self):

    """Create External Router Network Test

    RBAC test for the neutron create_network:router:external policy
    """
    with self.override_role():
        self._create_network(router_external=True)
```

Note that above the following expected error codes/rules relationship is observed:

- create_network => 403
- create_network:router:external => 403

A more involved example that expects a 404 to be raised, should the first policy under `rules` fail authorization, and a 403 to be raised for any subsequent policy authorization failure:

```
@rbac_rule_validation.action(service="neutron",
                             rules=["get_network",
                                    "update_network",
                                    "update_network:shared"],
                             expected_error_codes=[404, 403, 403])
@decorators.idempotent_id('37ea3e33-47d9-49fc-9bba-1af98fbd46d6')
def test_update_network_shared(self):

    """Update Shared Network Test

    RBAC test for the neutron update_network:shared policy
    """
    with self.override_role():
        self._update_network(shared_network=True)
    self.addCleanup(self._update_network, shared_network=False)
```

Note that above the following expected error codes/rules relationship is observed:

- get_network => 404
- update_network => 403
- update_network:shared => 403

Limitations

Multi-policy validation in RBAC tests comes with limitations, due to technical and practical challenges.

Technically, there are challenges associated with multiple policies across cross-service API communication in OpenStack, such as between Nova and Cinder or Nova and Neutron. The current framework does not account for these cross-service policy enforcement workflows, and it is still up for debate whether it should.

Practically, it is not possible to enumerate every policy enforced by every API in Patrole, as the maintenance overhead would be huge.

CHAPTER

TWO

USERS GUIDE

2.1 Patrole Configuration Guide

2.1.1 Patrole Configuration Guide

Patrole can be customized by updating Tempests `tempest.conf` configuration file. All Patrole-specific configuration options should be included under the `patrole` group.

RBAC Test Roles

The RBAC test roles govern the list of roles to be used when running Patrole tests. For example, setting `rbac_test_roles` to admin will execute all RBAC tests using admin credentials. Changing the `rbac_test_roles` value will *override* Tempests primary credentials to use that role.

This implies that, if `rbac_test_roles` is admin, regardless of the Tempest credentials used by a client, the client will be calling APIs using the admin role. That is, `self.os_primary.servers_client` will run as though it were `self.os_admin.servers_client`.

Similarly, setting `rbac_test_roles` with various roles, results in Tempests primary credentials being overridden by the roles specified by `rbac_test_roles`.

Note: Only the roles of the primary Tempest credentials (os_primary) are modified. The `user_id` and `project_id` remain unchanged.

Custom Policy Files

Patrole supports testing custom policy file definitions, along with default policy definitions. Default policy definitions are used if custom file definitions are not specified. If both are specified, the custom policy definition takes precedence (that is, replaces the default definition, as this is the default behavior in OpenStack).

The `custom_policy_files` option allows a user to specify a comma-separated list of custom policy file locations that are on the same host as Patrole. Each policy file must include the name of the service that is being tested: for example, if compute tests are executed, then Patrole will use the first policy file contained in `custom_policy_files` that contains the nova keyword.

Note: Patrole currently does not support policy files located on a host different than the one on which it is running.

Policy Feature Flags

Patroles `[policy-feature-enabled]` configuration group includes one option per supported policy feature flag. These feature flags are introduced when an OpenStack service introduces a new policy or changes a policy in a backwards-incompatible way. Since Patrole is branchless, it copes with the unexpected policy change by making the relevant policy change as well, but also introduces a new policy feature flag so that the test wont break N-1/N-2 releases where N is the currently supported release.

The default value for the feature flag is enabled for N and disabled for any releases prior to N in which the feature is not available. This is done by overriding the default value of the feature flag in DevStacks `lib/patrole` installation script. The change is made in Tempests DevStack script because Patroles DevStack plugin is hosted in-repo, which is branch-less (whereas the former is branched).

After the backwards-incompatible change no longer affects any supported release, then the corresponding policy feature flag is removed.

For more information on feature flags, reference the relevant Tempest documentation.

Sample Configuration File

The following is a sample Patrole configuration for adaptation and use. It is auto-generated from Patrole when this documentation is built, so if you are having issues with an option, please compare your version of Patrole with the version of this documentation.

Note that the Patrole configuration options actually live inside the Tempest configuration file; at runtime, Tempest populates its own configuration file with Patrole groups and options, assuming that Patrole is correctly installed and recognized as a plugin.

The sample configuration can also be viewed in file form.

```
[DEFAULT]


[patrole]

#
# From patrole.config
#

# DEPRECATED: The current RBAC role against which to run
# Patrole tests. (string value)
# This option is deprecated for removal.
# Its value may be silently ignored in the future.
# Reason: This option is deprecated and being
# replaced with ``rbac_test_roles``.
#rbac_test_role = admin
```

(continues on next page)

(continued from previous page)

```
# List of the RBAC roles against which to run
# Patrole tests. (list value)
#rbac_test_roles = admin

# List of the paths to search for policy files. Each
# policy path assumes that the service name is included in the path
# once. Also
# assumes Patrole is on the same host as the policy files. The paths
# should be
# ordered by precedence, with high-priority paths before low-priority
# paths. All
# the paths that are found to contain the service's policy file will
# be used and
# all policy files will be merged. Allowed ``json`` or ``yaml``
# formats.
#  (list value)
#custom_policy_files = /etc/%s/policy.json

#
# This option determines whether Patrole should run against a
# ``custom_requirements_file`` which defines RBAC requirements. The
# purpose of setting this flag to ``True`` is to verify that RBAC
# policy
# is in accordance to requirements. The idea is that the
# ``custom_requirements_file`` precisely defines what the RBAC
# requirements are.
#
# Here are the possible outcomes when running the Patrole tests
# against
# a ``custom_requirements_file``:
#
# YAML definition: allowed
# test run: allowed
# test result: pass
#
# YAML definition: allowed
# test run: not allowed
# test result: fail (under-permission)
#
# YAML definition: not allowed
# test run: allowed
# test result: fail (over-permission)
#  (boolean value)
#test_custom_requirements = false

#
# File path of the YAML file that defines your RBAC requirements. This
# file must be located on the same host that Patrole runs on. The YAML
# file should be written as follows:
```

(continues on next page)

(continued from previous page)

```
#
# .. code-block:: yaml
#
#     <service_foo>:
#       <api_action_a>:
#         - <allowed_role_1>
#         - <allowed_role_2>
#         - <allowed_role_3>
#       <api_action_b>:
#         - <allowed_role_2>
#         - <allowed_role_4>
#     <service_bar>:
#       <api_action_c>:
#         - <allowed_role_3>
#
# Where:
#
# service = the service that is being tested (Cinder, Nova, etc.).
#
# api_action = the policy action that is being tested. Examples:
#
# * volume:create
# * os_compute_api:servers:start
# * add_image
#
# allowed_role = the ``oslo.policy`` role that is allowed to perform
# the API.
#  (string value)
#custom_requirements_file = <None>

# Some of the policy rules have deprecated version,
# Patrole should be able to run check against default and deprecated
# rules,
# otherwise the result of the tests may not be correct.
#  (boolean value)
#validate_deprecated_rules = true


[patrole_log]

#
# From patrole.config
#

# Enables reporting on RBAC expected and actual test results for each
# Patrole test (boolean value)
#enable_reporting = false

# Name of file where output from 'enable_reporting' is logged. Note
```

(continues on next page)

(continued from previous page)

```
# that this file is recreated on each invocation of patrole (string
# value)
#report_log_name = patrole.log

# Path (relative or absolute) where the output from 'enable_reporting'
# is logged. This is combined with report_log_name to generate the
# full path. (string value)
#report_log_path = .


[policy-feature-enabled]

#
# From patrole.config
#

# Is the Neutron policy
# "create_port:fixed_ips:ip_address" available in the cloud? This
# policy was
# changed in a backwards-incompatible way. (boolean value)
#create_port_fixed_ips_ip_address_policy = true

# Is the Neutron policy
# "update_port:fixed_ips:ip_address" available in the cloud? This
# policy was
# changed in a backwards-incompatible way. (boolean value)
#update_port_fixed_ips_ip_address_policy = true

# Is the Cinder policy
# "limits_extension:used_limits" available in the cloud? This policy
# was
# changed in a backwards-incompatible way. (boolean value)
#limits_extension_used_limits_policy = true

# Is the Cinder policy
# "volume_extension:volume_actions:attach" available in the cloud?
# This policy
# was changed in a backwards-incompatible way. (boolean value)
#volume_extension_volume_actions_attach_policy = true

# Is the Cinder policy
# "volume_extension:volume_actions:reserve" available in the cloud?
# This policy
# was changed in a backwards-incompatible way. (boolean value)
#volume_extension_volume_actions_reserve_policy = true

# Is the Cinder policy
# "volume_extension:volume_actions:unreserve" available in the cloud?
# This policy
```

(continues on next page)

(continued from previous page)

```
# was changed in a backwards-incompatible way. (boolean value)
#volume_extension_volume_actions_unreserve_policy = true

# Are the Nova API extension policies available in the
# cloud (e.g. os_compute_api:os-extended-availability-zone)? These
# policies were
# removed in Stein because Nova API extension concept was removed in
# Pike. (boolean value)
#removed_nova_policies_stein = true

# Are the Nova API policies being removed in wallaby
# cycle (e.g. os_compute_api:os-agents)? (boolean value)
#removed_nova_policies_wallaby = true

# Are the obsolete Keystone policies available in the
# cloud (e.g. identity:[create|update|get|delete]_credential)? These
# policies
# were removed in Stein. (boolean value)
#removed_keystone_policies_stein = true

# Are the Cinder Stein policies available in the cloud
# (e.g. [create|update|get|delete]_encryption_policy)? These policies
# are added
# in Stein. (boolean value)
#added_cinder_policies_stein = true

# Is the cloud running the Train release or newer? If
# so, the Keystone Trust API is enforced differently depending on
# passed
# arguments (boolean value)
#keystone_policy_enforcement_train = true

# Are the Nova API policies available in the
# cloud (e.g. os_compute_api:os-services)? These policies were
# changed in Ussuri. (boolean value)
#changed_nova_policies_ussuri = true

# Are the Nova deprecated API policies available in the
# cloud (e.g. os_compute_api:os-networks)? These policies were
# changed in Victoria. (boolean value)
#changed_nova_policies_victoria = true

# Are the Cinder API policies changed in the
# cloud (e.g. 'group:group_types_specs')? These policies were
# changed in Xena. (boolean value)
#changed_cinder_policies_xena = true
```

2.2 Patrole Installation Guide

2.2.1 Patrole Installation Guide

Manual Installation Information

At the command line:

```
$ git clone http://git.openstack.org/openstack/patrole
$ sudo pip install ./patrole
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv patrole_env
$ workon patrole_env
$ pip install ./patrole
```

Or to install from the source:

```
$ navigate to patrole directory
$ sudo pip install -e .
```

DevStack Installation

2.2.2 Enabling in Devstack

Warning: The `stack.sh` script must be run in a disposable VM that is not being created automatically. See the README file in the DevStack repository for more information.

1. Download DevStack:

   ```
   git clone https://git.openstack.org/openstack-dev/devstack.git
   cd devstack
   ```

2. Patrole can be installed like any other DevStack plugin by including the `enable_plugin` directive inside local.conf:

   ```
   > cat local.conf
   [[local|localrc]]
   enable_plugin patrole https://git.openstack.org/openstack/patrole
   ```

3. Run `stack.sh` found in the DevStack repo.

CHAPTER

THREE

FIELD GUIDES

3.1 Patrole Field Guide Overview

3.1.1 Testing Scope

Patrole testing scope is strictly confined to Role-Based Access Control (RBAC). In OpenStack, `oslo.policy` is the RBAC library used by all major services. Thus, Patrole is concerned with validating that public API endpoints are correctly using `oslo.policy` for authorization.

In other words, all tests in Patrole are RBAC tests.

3.1.2 Patrole Field Guide to RBAC Tests

RBAC tests are Tempest-like API tests plus Patroles *RBAC Rule Validation Module*. All Patrole tests are RBAC validation tests for the OpenStack API.

3.1.3 Stable Tests

In the discussion below, correct means that a test is consistent with a services API-to-policy mapping and stable means that a test should require minimal maintenance for the supported releases.

Present

During the Queens release, a governance spec was pushed to support policy in code, which documents the mapping between APIs and each of their policies.

This documentation is an important prerequisite for ensuring that Patrole tests for a given service are correct. This mapping can be referenced to confirm that Patroles assumed mapping for a test is correct. For example, Nova has implemented policy in code which can be used to verify that Patroles Nova RBAC tests use the same mapping.

If a given service does not have policy in code, this implies that it is *more likely* that the RBAC tests for that service are inconsistent with the *intended* policy mapping. Until that service implements policy in code, it is difficult for Patrole maintainers to verify that tests for that service are correct.

Future

Once all services that Patrole tests have implemented policy in code and once Patrole has updated all its tests in accordance with the policy in code documentation then Patrole tests can guaranteed to be stable.

This stability will be denoted with a 1.0 version release.

3.2 Patrole Field Guide to RBAC Tests

3.2.1 What are these tests?

Patroles primary responsibility is to ensure that your OpenStack cloud has properly configured Role-Based Access Control (RBAC). All Patrole tests cases are devoted to this responsibility. Tempest API clients and utility functions are leveraged to accomplish this goal, but such functionality is secondary to RBAC validation.

Like Tempest, Patrole not only tests expected positive paths for RBAC validation, but also and more importantly negative paths. While Patrole could be thought of as validating RBAC, it more importantly verifies that your OpenStack cloud is secure from the perspective of RBAC (there are many gotchas when it comes to security, not just RBAC).

Negative paths are arguably more important than positive paths when it comes to RBAC and by extension security, because it is essential that your cloud be secure from unauthorized access. For example, while it is important to verify that the admin role has access to admin-level functionality, it is of critical importance to verify that non-admin roles *do not* have access to such functionality.

Unlike Tempest, Patrole accomplishes negative testing implicitly by abstracting it away in the background. Patrole dynamically determines whether a role should have access to an API depending on your clouds policy configuration and then confirms whether that is true or false.

3.2.2 Why are these tests in Patrole?

These tests constitute the core mission in Patrole: to verify RBAC. These tests are mainly intended to validate RBAC, but can also *unofficially* be used to discover the policy-to-API mapping for an OpenStack component.

It could be argued that some of these tests could be implemented in the projects themselves, but that approach has the following shortcomings:

- The projects do not validate RBAC from an integration testing perspective.
- By extension, RBAC across cross-service communication is not usually validated.
- The projects tests do not pass all the metadata to `oslo.policy` that is in reality passed by the deployed server to that library to determine whether a given user is authorized to perform an API action.
- The projects do not exhaustively do RBAC testing for all positive and negative paths.
- Patrole is designed to work with any role via configuration settings, but on the other hand the projects handpick which roles to test.

Why not use Patrole framework on Tempest tests?

The Patrole framework cant be applied to existing Tempest tests via *RBAC Rule Validation Module*, because:

- Tempest tests arent factored the right way: Theyre not granular enough. They call too many APIs and too many policies are enforced by each test.
- Tempest tests assume default policy rules: Tempest uses `os_admin` credentials for admin APIs and `os_primary` for non-admin APIs. This breaks for custom policy overrides.
- Tempest doesnt have tests that enforce all the policy actions, regardless. Some RBAC tests require that tests be written a very precise way for the server to authorize the expected policy actions.

Why are these tests not in Tempest?

Patrole should be a separate project that specializes in RBAC tests. This was agreed upon during discussion that led to the approval of the RBAC testing framework spec, which was the genesis for Patrole.

Philosophically speaking:

- Tempest supports API and scenario testing. RBAC testing is out of scope.
- The OpenStack project structure reform evolved OpenStack to a more decentralized model where [projects like QA] provide processes and tools to empower projects to do the work themselves. This model resulted in the creation of the Tempest external plugin interface.
- Tempest supports plugins. Why not use one for RBAC testing?

Practically speaking:

- The Tempest team should not be burdened with having to support Patrole, too. Tempest is a big project and having to absorb RBAC testing is difficult.
- Tempest already has many in-tree Zuul checks/gates. If Patrole tests lived in Tempest, then adding more Zuul checks/gates for Patrole would only make it harder to get changes merged in Tempest.

3.2.3 Scope of these tests

RBAC tests should always use the Tempest implementation of the OpenStack API, to take advantage of Tempests stable library.

Each test should test a specific API endpoint and the related policy.

Each policy should be tested in isolation of one another or at least as close to this rule as possible to ensure proper validation of RBAC.

Each test should be able to work for positive and negative paths.

All tests should be able to be run on their own, not depending on the state created by a previous test.

CHAPTER

FOUR

FOR CONTRIBUTORS

- If you are a new contributor to Patrole please refer: *So You Want to Contribute*

4.1 So You Want to Contribute

For general information on contributing to OpenStack, please check out the contributor guide to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

Below will cover the more project specific information you need to get started with Tempest.

4.1.1 Communication

- IRC channel `#openstack-qa` at OFTC
- Mailing list (prefix subjects with `[qa]` for faster responses) http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss

4.1.2 Contacting the Core Team

Please refer to the Patrole Core Team contacts.

4.1.3 New Feature Planning

If you want to propose a new feature please read Feature Proposal Process Patrole features are tracked on Storyboard.

4.1.4 Task Tracking

We track our tasks in Storyboard.

If youre looking for some smaller, easier work item to pick up and get started on, search for the low-hanging-fruit tag.

4.1.5 Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so on StoryBoard.

4.1.6 Getting Your Patch Merged

All changes proposed to the Patrole requires one `Code-Review +2` votes from Patrole core reviewers before one of the core reviewers can approve patch by giving `Workflow +1` vote. More detailed guidelines for reviewers are available at *Reviewing Patrole Code*.

4.1.7 Project Team Lead Duties

All common PTL duties are enumerated in the PTL guide.

The Release Process for QA is documented in QA Release Process.

CHAPTER

FIVE

DEVELOPERS GUIDE

5.1 Patrole Coding Guide

5.1.1 Patrole Coding Guide

- Step 1: Read the OpenStack Style Commandments: https://docs.openstack.org/hacking/latest/
- Step 2: Review Tempests Style Commandments: https://docs.openstack.org/tempest/latest/HACKING.html
- Step 3: Read on

Patrole Specific Commandments

Patrole borrows the following commandments from Tempest; refer to Tempests Commandments for more information:

Note: The original Tempest Commandments do not include Patrole-specific paths. Patrole-specific paths replace the Tempest-specific paths within Patroles hacking checks.

- [T102] Cannot import OpenStack python clients in `patrole_tempest_plugin/tests/api`
- [T105] Tests cannot use setUpClass/tearDownClass
- [T107] Check that a service tag isnt in the module path
- [T108] Check no hyphen at the end of rand_name() argument
- [T109] Cannot use testtools.skip decorator; instead use `decorators.skip_because` from `tempest.lib`
- [T113] Check that tests use `data_utils.rand_uuid()` instead of `uuid.uuid4()`
- [N322] Methods default argument shouldnt be mutable

The following are Patroles specific Commandments:

- [P100] The `rbac_rule_validation.action` decorator must be applied to all RBAC tests
- [P101] RBAC test filenames must end with _rbac.py; for example, test_servers_rbac.py, not test_servers.py
- [P102] RBAC test class names must end in RbacTest

- [P103] `self.client` must not be used as a client alias; this allows for code that is more maintainable and easier to read
- [P104] RBAC extension test class names must end in ExtRbacTest

Supported OpenStack Components

Patrole only offers **in-tree** integration testing coverage for the following components:

- Cinder
- Glance
- Keystone
- Neutron
- Nova

Patrole currently has no stable library, so reliance upon Patroles framework for external RBAC testing should be done with caution. Nonetheless, even when Patrole has a stable library, it will only offer in-tree RBAC testing for the components listed above.

Role Overriding

Correct role overriding is vital to correct RBAC testing within Patrole. If a test does not call `self.override_role()` within the RBAC test, followed by the API endpoint that enforces the expected policy action, then the test is **not** a valid Patrole test: The API endpoint under test will be performed with admin role, which is always wrong unless `CONF.patrole.rbac_test_role` is also admin.

Branchless Patrole Considerations

Like Tempest, Patrole is branchless. This is to better ensure API and RBAC consistency between releases because API and RBAC behavior should not change between releases. This means that the stable branches are also gated by the Patrole master branch, which also means that proposed commits to Patrole must work against both the master and all the currently supported stable branches of the projects. As such there are a few special considerations that have to be accounted for when pushing new changes to Patrole.

1. New Tests for new features

Patrole, like Tempest, *implicitly* tests new features because new policies oftentimes accompany new features. The same Tempest philosophy regarding feature flags and new features also applies to Patrole.

2. New Tests for new policies

When adding tests for new policies that were not in previous releases of the projects, the new test must be properly skipped with a feature flag. This involves using the `testtools.skip(Unless|If)` decorator above the test to check if the required policy is enabled. Similarly, a feature flag must be used whenever an OpenStack service covered by Patrole changes one of its policies in a backwards-incompatible way. If there isnt a method of selecting the new policy from the config file then there wont be a mechanism to disable the test with older stable releases and the new test wont be able to merge.

Introduction of a new feature flag requires specifying a default value for the corresponding config option that is appropriate in the latest OpenStack release. Because Patrole is branchless, the feature flags default value will need to be overridden to a value that is appropriate in earlier releases in which the feature isnt available. In DevStack, this can be accomplished by modifying Patroles lib installation script for previous branches (because DevStack is branched).

3. Bug fix on core project needing Patrole changes

When trying to land a bug fix which changes a tested API youll have to use the following procedure:

1. Propose change to the project, get a +2 on the change even with the test failing Patrole side.
2. Propose skip to the relevant Patrole test which will only be approved after the corresponding change in the project has a +2.
3. Land project change in master and all open stable branches (if required).
4. Land changed test in Patrole.

Otherwise the bug fix wont be able to land in the project.

4. New Tests for existing features or policies

The same Tempest logic regarding new tests for existing features or policies also applies to Patrole.

Black Box vs. White Box Testing

Tempest is a black box testing framework, meaning that it is concerned with testing public API endpoints and doesnt concern itself with testing internal implementation details. Patrole, as a Tempest plugin, also falls underneath the category of black box testing. However, even with policy in code documentation, some degree of white box testing is required in order to correctly write RBAC tests.

This is because *Policy in Code* documentation, while useful in many respects, is usually quite brief and its main purpose is to help operators understand how to customize policy configuration rather than to help developers understand complex policy authorization work flows. For example, policy in code documentation doesnt make deriving *multiple policies* easy. Such documentation also doesnt usually mention that a specific parameter needs to be set, or that a particular microversion must be enabled, or that a particular set of prerequisite API or policy actions must be executed, in order for the policy under test to be enforced by the server. This means that test writers must account for the internal RBAC implementation in API code in order to correctly understand the complete RBAC work flow within an API.

Besides, as mentioned *elsewhere* in this documentation, not all services currently implement policy in code, making some degree of white box testing a necessary evil for writing robust RBAC tests.

5.2 Reviewing Patrole Code

5.2.1 Reviewing Patrole Code

To start read the OpenStack Common Review Checklist

Ensuring code is executed

Any new test or change to an existing test has to be verified in the gate. This means that the first thing to check with any change is that a gate job actually runs it. Tests which arent executed either because of configuration or skips should not be accepted.

Execution time

Along with checking that the jobs log that a new test is actually executed, also pay attention to the execution time of that test. Patrole already runs hundreds of tests per job in its check and gate pipelines and it is important that the overall runtime of the jobs be constrained as much as possible. Consider applying the `@decorators.attr(type='slow')` test attribute decorator to a test if its runtime is longer than 30 seconds.

Unit Tests

For any change that adds new functionality to common functionality unit tests are required. This is to ensure we dont introduce future regressions and to test conditions which we may not hit in the gate runs.

API Stability

Tests should only be added for published stable APIs. If a patch contains tests for an API which hasnt been marked as stable or for an API which doesnt conform to the API stability guidelines then it should not be approved.

Similarly, tests should only be added for policies that are covered by policy in code documentation. Any existing tests that test policies not covered by such documentation are either:

- part of a service that has not yet migrated to policy in code; or
- legacy in the sense that they were created prior to policy in code

For the first bullet, the tests should not be considered stable, but should be kept around to maintain coverage. These tests are a best-effort attempt at offering RBAC test coverage for the service that has not yet migrated to policy in code.

For the second bullet, the tests should be updated to conform to policy in code documentation, if applicable.

Reject Copy and Paste Test Code

When creating new tests that are similar to existing tests it is tempting to simply copy the code and make a few modifications. This increases code size and the maintenance burden. Such changes should not be approved if it is easy to abstract the duplicated code into a function or method.

Tests overlap

When a new test is being proposed, question whether this feature is not already tested with Patrole. Patrole has more than 600 tests, spread amongst many directories, so its easy to introduce test duplication.

Test Duplication

Test duplication means:

- testing an API endpoint in more than one test
- testing the same policy in more than one test

For the first bullet, try to avoid calling the same API inside the `self.override_role()` call.

Note: If the same API is tested against different policies, consider combining the different tests into only 1 test, that tests the API against all the policies it enforces.

For the second bullet, try to avoid testing the same policies across multiple tests.

Note: This is not always possible since policy granularity doesnt exist for all APIs. In cases where policy granularity doesnt exist, make sure that the policy overlap only exists for the non-granular APIs that enforce the same policy.

Being explicit

When tests are being added that depend on a configurable feature or extension, polling the API to discover that it is enabled should not be done. This will just result in bugs being masked because the test can be skipped automatically. Instead the config file should be used to determine whether a test should be skipped or not. Do not approve changes that depend on an API call to determine whether to skip or not.

Multi-Policy Guidelines

Care should be taken when using multiple policies in an RBAC test. The following guidelines should be followed before deciding to add multiple policies to a Patrole test.

General Multi-policy API Code Guidelines

The list below enumerates guidelines beginning with those with the highest priority and ending with those with the lowest priority. A higher priority item takes precedence over lower priority items.

1. Order the policies in the `rules` parameter chronologically with respect to the order theyre called by the API endpoint under test.
2. Only use policies that map to the API by referencing the appropriate policy in code documentation.
3. Only include the minimum number of policies needed to test the API correctly: dont add extraneous policies.
4. If possible, only use policies that directly relate to the API. If the policies are used across multiple APIs, try to omit it. If a generic policy needs to be added to get the test to pass, then this is fair game.
5. Limit the number of policies to a reasonable number, such as 3.

Neutron Multi-policy API Code Guidelines

Because Neutron can raise a 403 or 404 following failed authorization, Patrole uses the `expected_error_codes` parameter to accommodate this behavior. Each policy action enumerated in `rules` must have a corresponding entry in `expected_error_codes`. Each expected error code must be either a 403 or a 404, which indicates that, when policy enforcement fails for the corresponding policy action, that error code is expected by Patrole. For more information about these parameters, see *RBAC Rule Validation Module*.

The list below enumerates additional multi-policy guidelines that apply in particular to Neutron. A higher priority item takes precedence over lower priority items.

1. Order the expected error codes in the `expected_error_codes` parameter chronologically with respect to the order each corresponding policy in `rules` is authorized by the API under test.
2. Ensure the *Neutron Multi-policy Validation* is followed when determining the expected error code for each corresponding policy.

The same guidelines under *General Multi-policy API Code Guidelines* should be applied afterward.

Release Notes

Release notes are how we indicate to users and other consumers of Patrole what has changed in a given release. There are certain types of changes that require release notes and we should not approve them without including a release note. These include but arent limited to, any addition, deprecation or removal from the framework code, any change to configuration options (including deprecation), major feature additions, and anything backwards incompatible or would require a user to take note or do something extra.

Deprecated Code

Sometimes we have some bugs in deprecated code. Basically, we leave it. Because we dont need to maintain it. However, if the bug is critical, we might need to fix it. When it will happen, we will deal with it on a case-by-case basis.

When to approve

- Every patch can be approved with single +2 which means single reviewer can approve.
- Its OK to hold off on an approval until a subject matter expert reviews it.
- If a patch has already been approved but requires a trivial rebase to merge, you do not have to wait for a +2, since the patch has already had +2s. With single +2 rule, this means that author can also approve this case if he/she has approve rights.

5.3 Patrole Test Writing Overview

5.3.1 Introduction

Patrole tests are broken up into 3 stages:

1. *Test Setup*
2. *Test Execution*
3. *Test Cleanup*

See the *framework overview documentation* for a high-level explanation of the entire testing work flow and framework implementation. The guide that follows is concerned with helping developers know how to write Patrole tests.

5.3.2 Role Overriding

Role overriding is the way Patrole is able to create resources and delete resources including those that require admin credentials while still being able to exercise the same set of Tempest credentials to perform the API action that authorizes the policy under test, by manipulating roles of the Tempest credentials.

Patrole implicitly splits up each test into 3 stages: set up, test execution, and teardown.

The role workflow is as follows:

1. Setup: Admin role is used automatically. The primary credentials are overridden with the admin role.
2. Test execution: `[patrole] rbac_test_roles` is used manually via the call to `with self.override_role()`. Everything that is executed within this contextmanager uses the primary credentials overridden with the `[patrole] rbac_test_roles`.
3. Teardown: Admin role is used automatically. The primary credentials have been overridden with the admin role.

5.3.3 Test Setup

Automatic role override in background.

Resources can be set up inside the `resource_setup` class method that Tempest provides. These resources are typically reserved for expensive resources in terms of memory or storage requirements, like volumes and VMs. These resources are **always** created via the admin role; Patrole automatically handles this.

Like Tempest, however, Patrole must also create resources inside tests themselves. At the beginning of each test, the primary credentials have already been overridden with the admin role. One can create whatever test-level resources one needs, without having to worry about permissions.

5.3.4 Test Execution

Manual role override required.

Test execution here means calling the API endpoint that enforces the policy action expected by the `rbac_rule_validation` decorator. Test execution should be performed *only after* calling `with self.override_role()`.

Immediately after that call, the API endpoint that enforces the policy should be called.

Examples

Always use the contextmanager before calling the API that enforces the expected policy action.

Example:

```
@rbac_rule_validation.action(
    service="nova",
    rules=["os_compute_api:os-aggregates:show"])
def test_show_aggregate_rbac(self):
    # Do test setup before the ``override_role`` call.
    aggregate_id = self._create_aggregate()
    # Call the ``override_role`` method so that the primary credentials
    # have the test role needed for test execution.
    with self.override_role():
        self.aggregates_client.show_aggregate(aggregate_id)
```

When using a waiter, do the wait outside the contextmanager. Waiting always entails executing a `GET` request to the server, until the state of the returned resource matches a desired state. These `GET` requests enforce a different policy than the one expected. This is undesirable because Patrole should only test policies in isolation from one another.

Otherwise, the test result will be tainted, because instead of only the expected policy getting enforced with the `os_primary` role, at least two policies get enforced.

Example using waiter:

```
@rbac_rule_validation.action(
    service="nova",
    rules=["os_compute_api:os-admin-password"])
```

(continues on next page)

(continued from previous page)

```
def test_change_server_password(self):
    original_password = self.servers_client.show_password(
        self.server['id'])
    self.addCleanup(self.servers_client.change_password, self.server['id'],
                    adminPass=original_password)

    with self.override_role():
        self.servers_client.change_password(
            self.server['id'], adminPass=data_utils.rand_password())
    # Call the waiter outside the ``override_role`` contextmanager, so that
    # it is executed with admin role.
    waiters.wait_for_server_status(
        self.servers_client, self.server['id'], 'ACTIVE')
```

Below is an example of a method that enforces multiple policies getting called inside the contextmanager. The `_complex_setup_method` below performs the correct API that enforces the expected policy in this case `self.resources_client.create_resource` but then proceeds to use a waiter.

Incorrect:

```
def _complex_setup_method(self):
    resource = self.resources_client.create_resource(
        **kwargs)['resource']
    self.addCleanup(test_utils.call_and_ignore_notfound_exc,
                    self._delete_resource, resource)
    waiters.wait_for_resource_status(
        self.resources_client, resource['id'], 'available')
    return resource

@rbac_rule_validation.action(
    service="example-service",
    rules=["example-rule"])
def test_change_server_password(self):
    # Never call a helper function inside the contextmanager that calls a
    # bunch of APIs. Only call the API that enforces the policy action
    # contained in the decorator above.
    with self.override_role():
        self._complex_setup_method()
```

To fix this test, see the Example using waiter section above. It is recommended to re-implement the logic in a helper method inside a test such that only the relevant API is called inside the contextmanager, with everything extraneous outside.

5.3.5 Test Cleanup

Automatic role override in background.

After the test no matter whether it ended successfully or in failure the credentials are overridden with the admin role by the Patrole framework, *before* `tearDown` or `tearDownClass` are called. This means that resources are always cleaned up using the admin role.

5.4 Framework

5.4.1 RBAC Testing Validation

Validation Workflow Overview

RBAC testing validation is broken up into 3 stages:

1. Expected stage. Determine whether the test should be able to succeed or fail based on the test roles defined by `[patrole] rbac_test_roles`) and the policy action that the test enforces.
2. Actual stage. Run the test by calling the API endpoint that enforces the expected policy action using the test roles.
3. Comparing the outputs from both stages for consistency. A consistent result is treated as a pass and an inconsistent result is treated as a failure. Consistent (or successful) cases include:

   - Expected result is `True` and the test passes.
   - Expected result is `False` and the test fails.

   For example, a 200 from the API call and a `True` result from `oslo.policy` or a 403 from the API call and a `False` result from `oslo.policy` are successful results.

   Inconsistent (or failing) cases include:

   - Expected result is `False` and the test passes. This results in an `RbacOverPermissionException` exception getting thrown.
   - Expected result is `True` and the test fails. This results in a `RbacOverPermissionException` exception getting thrown.

   For example, a 200 from the API call and a `False` result from `oslo.policy` or a 403 from the API call and a `True` result from `oslo.policy` are failing results.

Warning: Note that Patrole cannot currently derive the expected policy result for service-specific `oslo.policy` checks, like Neutrons FieldCheck, because such checks are contained within the services code base itself, which Patrole cannot import.

The RBAC Rule Validation Module

High-level module that provides the decorator that wraps around Tempest tests and serves as the entry point for RBAC testing validation. The workflow described above is ultimately carried out by the decorator.

For more information about this module, please see *RBAC Rule Validation Module*.

The Policy Authority Module

Module called by *RBAC Rule Validation Module* to verify whether the test roles are allowed to execute a policy action by querying `oslo.policy` with required test data. The result is used by *RBAC Rule Validation Module* as the Expected result.

For more information about this module, please see *Policy Authority Module*.

The RBAC Utils Module

This module is responsible for handling role switching, the mechanism by which Patrole is able to set up, tear down and execute APIs using the same set of credentials. Every RBAC test must perform a role switch even if the role that is being switched to is admin.

For more information about this module, please see *RBAC Utils Module*.

5.4.2 RBAC Rule Validation Module

Overview

Module that implements the decorator which serves as the entry point for RBAC validation testing. The decorator should be applied to every RBAC test with the appropriate `service` (OpenStack service) and `rule` (OpenStack policy name defined by the `service`).

Implementation

`RBAC Rule Validation Module`

5.4.3 RBAC Authority Module

Overview

This module implements an abstract class that is implemented by the classes below. Each implementation is used by the *RBAC Rule Validation Module* framework to determine each expected test result.

Policy Authority Module

The *default* *RbacAuthority* implementation class which is used for policy validation. Uses `oslo.policy` to determine the expected test result.

All Patrole Zuul gates use this *RbacAuthority* class by default.

Requirements Authority Module

Optional *RbacAuthority* implementation class which is used for policy validation. It uses a high-level requirements-driven approach to validating RBAC in Patrole.

Implementation

RBAC Authority Module

5.4.4 Policy Authority Module

Overview

This module is only called for calculating the Expected result if `[patrole] test_custom_requirements` is `False`.

Using the *PolicyAuthority* class, policy verification is performed by:

1. Pooling together the default *in-code* policy rules.
2. Overriding the defaults with custom policy rules located in a policy.json, if the policy file exists and the custom policy definition is explicitly defined therein.
3. Confirming that the policy action for example, list_users exists. (`oslo.policy` otherwise claims that role foo is allowed to perform policy action bar, for example, because it defers to the default policy rule and oftentimes the default can be anyone allowed).
4. Performing a call with all necessary data to `oslo.policy` and returning the expected result back to `rbac_rule_validation` decorator.

When to use

This *RbacAuthority* class can be used to validate the default OpenStack policy configuration. It is recommended that this approach be used for RBAC validation for clouds that use little to no policy customizations or overrides.

This validation approach should be used when:

- Validating the out-of-the-box policy-in-code OpenStack policy configuration.

  It is important that the default OpenStack policy configuration be validated before deploying OpenStack into production. Bugs exist in software and the earlier they can be caught and prevented (via CI/CD, for example), the better. Patrole continues to be used to identify default policy bugs across OpenStack services.

- Validating policy reliably and accurately.

  Relying on `oslo.policy` to compute the expected test results provides accurate tests, without the hassle of having to reinvent the wheel. Since OpenStack APIs use `oslo.policy` for policy enforcement, it makes sense to compute expected results by using the same library, ensuring test reliability.

- Continuously validating policy changes to OpenStack projects under development by gating them against Patrole CI/CD jobs run by Zuul.

Implementation

Policy Authority Module

5.4.5 Requirements Authority Module

Overview

Requirements-driven approach to declaring the expected RBAC test results referenced by Patrole. These requirements express the *intention* behind the policy. A high-level YAML syntax is used to concisely and clearly map each policy action to the list of associated roles.

Note: The *Custom Requirements File* is required to use this validation approach and, currently, must be manually generated.

This validation approach can be toggled on by setting the `[patrole].test_custom_requirements` configuration option to `True`; see *Patrole Configuration Guide* for more information.

When to use

This *RbacAuthority* class can be used to achieve a requirements-driven approach to validating an OpenStack clouds RBAC implementation. Using this approach, Patrole computes expected test results by performing lookups against a *Custom Requirements File* which precisely defines the clouds RBAC requirements.

This validation approach should be used when:

- The cloud has heavily customized policy files that require careful validation against ones requirements.

  Heavily customized policy files can contain relatively nuanced/technical syntax that impinges upon the goal of using a clear and concise syntax present in the *Custom Requirements File* to drive RBAC validation.

- The cloud has non-OpenStack services that require RBAC validation but which dont leverage the `oslo.policy` framework.

  Services like Contrail that are present in an OpenStack-based cloud that interface with OpenStack services like Neutron also require RBAC validation. The requirements-driven approach to RBAC validation is framework-agnostic and so can work with any policy engine.

- Expected results are captured as clear-cut, unambiguous requirements.

  Validating a clouds RBAC against high-level, clear-cut requirements is a valid use case. Relying on `oslo.policy` validating customized policy files is not sufficient to satisfy this use case.

As mentioned above, the trade-off with this approach is having to manually generate the *Custom Requirements File*. There is currently no tooling to automatically do this.

Custom Requirements File

File path of the YAML file that defines your RBAC requirements. This file must be located on the same host that Patrole runs on. The YAML file should be written as follows:

```
<service_foo>:
  <logical_or_example>:
    - <allowed_role_1>
    - <allowed_role_2>
  <logical_and_example>:
    - <allowed_role_3>, <allowed_role_4>
<service_bar>:
  <logical_not_example>:
    - <!disallowed_role_5>
```

Where:

- `service` - the service that is being tested (Cinder, Nova, etc.).
- `api_action` - the policy action that is being tested. Examples:
  - volume:create
  - os_compute_api:servers:start
  - add_image
- `allowed_role` - the `oslo.policy` role that is allowed to perform the API.

Each item under `logical_or_example` is logical OR-ed together. Each role in the comma-separated string under `logical_and_example` is logical AND-ed together. And each item prefixed with `!` under `logical_not_example` is logical negated.

Note: The custom requirements file only allows policy actions to be mapped to the associated roles that define it. Complex `oslo.policy` constructs like `literals` or `GenericChecks` are not supported. For more information, reference the oslo.policy documentation.

Examples

Items within `api_action` are considered as logical or, so you may read:

```
<service_foo>:
  # "api_action_a: allowed_role_1 or allowed_role_2 or allowed_role_3"
  <api_action_a>:
    - <allowed_role_1>
    - <allowed_role_2>
    - <allowed_role_3>
```

as `<allowed_role_1> or <allowed_role_2> or <allowed_role_3>`.

Roles within comma-separated items are considered as logic and, so you may read:

```
<service_foo>:
  # "api_action_a: (allowed_role_1 and allowed_role_2) or allowed_role_3"
  <api_action_a>:
    - <allowed_role_1>, <allowed_role_2>
    - <allowed_role_3>
```

as `<allowed_role_1> and <allowed_role_2> or <allowed_role_3>`.

Also negative roles may be defined with an exclamation mark ahead of role:

```
<service_foo>:
  # "api_action_a: (allowed_role_1 and allowed_role_2 and not
  # disallowed_role_4) or allowed_role_3"
  <api_action_a>:
    - <allowed_role_1>, <allowed_role_2>, !<disallowed_role_4>
    - <allowed_role_3>
```

This example must be read as `<allowed_role_1> and <allowed_role_2> and not <disallowed_role_4> or <allowed_role_3>`.

Implementation

Requirements Authority Module

5.4.6 RBAC Utils Module

Overview

Patrole manipulates the `os_primary` Tempest credentials, which are the primary set of Tempest credentials. It is necessary to use the same credentials across the entire test setup/test execution/test teardown workflow because otherwise 400-level errors will be thrown by OpenStack services.

This is because many services check the request contexts project scope and in very rare cases, user scope. However, each set of Tempest credentials (via dynamic credentials) is allocated its own distinct project. For example, the `os_admin` and `os_primary` credentials each have a distinct project, meaning that it is not always possible for the `os_primary` credentials to access resources created by the `os_admin` credentials.

The only foolproof solution is to manipulate the role for the same set of credentials, rather than using distinct credentials for setup/teardown and test execution, respectively. This is especially true when considering custom policy rule definitions, which can be arbitrarily complex.

Implementation

RBAC Utils Module

5.4.7 patrole_tempest_plugin

patrole_tempest_plugin package

Submodules

patrole_tempest_plugin.policy_authority module

class patrole_tempest_plugin.policy_authority.**PolicyAuthority**(*project_id*, *user_id*, *service*, *extra_target_data=None*)

Bases: *patrole_tempest_plugin.rbac_authority.RbacAuthority*

A class that uses `oslo.policy` for validating RBAC.

allowed(*rule_name*, *roles*)

Checks if a given rule in a policy is allowed with given role.

Parameters

- **rule_name** (*string*) Policy name to pass to“oslo.policy“.
- **roles** (*List[string]*) List of roles to validate for authorization.

Raises *RbacParsingException* If `rule_name` does not exist in the cloud (in policy file or among registered in-code policy defaults).

classmethod discover_policy_files()

Dynamically discover the policy file for each service in `cls.available_services`. Pick all candidate paths found out of the potential paths in `[patrole] custom_policy_files`.

get_rules()

os_admin = None

classmethod validate_service(*service*)

Validate whether the service passed to `__init__` exists.

patrole_tempest_plugin.rbac_authority module

class patrole_tempest_plugin.rbac_authority.**RbacAuthority**

Bases: object

Class for validating whether a given role can perform a policy action.

Any class that extends `RbacAuthority` provides the logic for determining whether a role has permissions to execute a policy action.

abstract allowed(*rule*, *role*)

Determine whether the role should be able to perform the API.

Parameters

- **rule** The name of the policy enforced by the API.
- **role** The role used to determine whether `rule` can be executed.

Returns True if the `role` has permissions to execute `rule`, else False.

patrole_tempest_plugin.rbac_exceptions module

exception patrole_tempest_plugin.rbac_exceptions.**BasePatroleException**(**args*, ***kwargs*)

Bases: tempest.lib.exceptions.TempestException

message = 'An unknown RBAC exception occurred'

exception patrole_tempest_plugin.rbac_exceptions.**BasePatroleResponseBodyException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

message = 'Response body incomplete due to RBAC authorization failure'

exception patrole_tempest_plugin.rbac_exceptions.**RbacEmptyResponseBody**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleResponseBodyException*

Raised when a list or show action is empty following RBAC authorization failure.

message = 'The response body is empty due to policy enforcement failure.'

exception patrole_tempest_plugin.rbac_exceptions.**RbacExpectedWrongException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when the expected exception does not match the actual exception raised, when both are instances of Forbidden or NotFound, indicating the test provides a wrong argument to *expected_error_codes*.

message = 'Expected %(expected)s to be raised but %(actual)s was raised instead. Actual exception: %(exception)s'

exception patrole_tempest_plugin.rbac_exceptions.**RbacInvalidErrorCode**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

message = 'Unsupported error code passed in test'

exception patrole_tempest_plugin.rbac_exceptions.**RbacInvalidServiceException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when an invalid service is passed to `rbac_rule_validation` decorator.

message = 'Attempted to test an invalid service'

exception patrole_tempest_plugin.rbac_exceptions.**RbacMissingAttributeResponseBody**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleResponseBodyException*

Raised when a list or show action is missing an attribute following RBAC authorization failure.

message = 'The response body is missing the expected %(attribute)s due to policy enforcement failure'

exception patrole_tempest_plugin.rbac_exceptions.**RbacOverPermissionException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when the expected result is failure but the actual result is pass.

message = 'Unauthorized action was allowed to be performed'

exception patrole_tempest_plugin.rbac_exceptions.**RbacOverrideRoleException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when override_role is used incorrectly or fails somehow.

Used for safeguarding against false positives that might occur when the expected exception isnt raised inside the `override_role` context. Specifically, when:

- `override_role` isnt called
- an exception is raised before `override_role` context
- an exception is raised after `override_role` context

message = 'Override role failure or incorrect usage'

exception patrole_tempest_plugin.rbac_exceptions.**RbacParsingException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

message = 'Attempted to test an invalid policy file or action'

exception patrole_tempest_plugin.rbac_exceptions.**RbacPartialResponseBody**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleResponseBodyException*

Raised when a list action only returns a subset of the available resources.

For example, admin can return more resources than member for a list action.

message = 'The response body only lists a subset of the available resources due to partial policy enforcement failure. Response body: %(body)s'

exception patrole_tempest_plugin.rbac_exceptions.**RbacResourceSetupFailed**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

message = 'RBAC resource setup failed'

exception patrole_tempest_plugin.rbac_exceptions.**RbacUnderPermissionException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when the expected result is pass but the actual result is failure.

message = 'Authorized action was not allowed to be performed'

exception patrole_tempest_plugin.rbac_exceptions.**RbacValidateListException**(**args*, ***kwargs*)

Bases: *patrole_tempest_plugin.rbac_exceptions.BasePatroleException*

Raised when override_role_and_validate_list is used incorrectly.

Specifically, when:

- Neither `resource_id` nor `resources` is initialized
- Both `resource_id` and `resources` are initialized
- **The `ctx.resources` variable wasnt set in** override_role_and_validate_list context.

message = 'Incorrect usage of override_role_and_validate_list: %(reason)s'

patrole_tempest_plugin.rbac_rule_validation module

patrole_tempest_plugin.rbac_rule_validation.**action**(*service*, *rules=None*, *expected_error_codes=None*, *extra_target_data=None*)

A decorator for verifying OpenStack policy enforcement.

A decorator which allows for positive and negative RBAC testing. Given:

- an OpenStack service,
- a policy action (`rule`) enforced by that service, and
- the test roles defined by `[patrole] rbac_test_roles`

determines whether the test role has sufficient permissions to perform an API call that enforces the `rule`.

This decorator should only be applied to an instance or subclass of `tempest.test.BaseTestCase`.

The result from `_is_authorized` is used to determine the *expected* test result. The *actual* test result is determined by running the Tempest test this decorator applies to.

Below are the following possibilities from comparing the *expected* and *actual* results:

1) If *expected* is True and the test passes (*actual*), this is a success.
2) If *expected* is True and the test fails (*actual*), this results in a `RbacUnderPermissionException` exception failure.
3) If *expected* is False and the test passes (*actual*), this results in an `RbacOverPermissionException` exception failure.
4) If *expected* is False and the test fails (*actual*), this is a success.

As such, negative and positive testing can be applied using this decorator.

Parameters

- **service** (*str*) An OpenStack service. Examples: nova or neutron.
- **rules** (*list[str] or list[callable]*) A list of policy actions defined in a policy file or in code. The rules are logical-ANDed together to derive the expected result. Also accepts list of callables that return a policy action.

  Note: Patrole currently only supports custom JSON policy files.

- **expected_error_codes** (*list[int]*) When the `rules` list parameter is used, then this list indicates the expected error code to use if one of the rules does not allow the role being tested. This list must coincide with and its elements remain in the same order as the rules in the rules list.

  Example:

  ```
  rules=["api_action1", "api_action2"]
  expected_error_codes=[404, 403]
  ```

  a) If api_action1 fails and api_action2 passes, then the expected error code is 404.

  b) if api_action2 fails and api_action1 passes, then the expected error code is 403.

  c) if both api_action1 and api_action2 fail, then the expected error code is the first error seen (404).

  If it is not passed, then it is defaulted to 403.

  Warning: A 404 should not be provided *unless* the endpoint masks a `Forbidden` exception as a `NotFound` exception.

- **extra_target_data** (*dict*) Dictionary, keyed with `oslo.policy` generic check names, whose values are string literals that reference nested `tempest.test.BaseTestCase` attributes. Used by `oslo.policy` for performing matching against attributes that are sent along with the API calls. Example:

```
extra_target_data={
    "target.token.user_id":
    "os_alt.auth_provider.credentials.user_id"
})
```

Raises

- ***RbacInvalidServiceException*** If `service` is invalid.
- ***RbacUnderPermissionException*** For item (2) above.
- ***RbacOverPermissionException*** For item (3) above.
- ***RbacExpectedWrongException*** When a 403 is expected but a 404 is raised instead or vice versa.

Examples:

```
@rbac_rule_validation.action(
    service="nova",
    rules=["os_compute_api:os-agents"])
def test_list_agents_rbac(self):
    # The call to `override_role` is mandatory.
    with self.override_role():
        self.agents_client.list_agents()
```

patrole_tempest_plugin.rbac_utils module

class patrole_tempest_plugin.rbac_utils.**RbacUtilsMixin**(**args*, ***kwargs*)

Bases: `object`

Utility mixin responsible for switching `os_primary` role.

Should be used as a mixin class alongside an instance of `tempest.test.BaseTestCase` to perform Patrole class setup for a base RBAC class. Child classes should not use this mixin.

Example:

```
class BaseRbacTest(rbac_utils.RbacUtilsMixin, base.BaseV2ComputeTest):

    @classmethod
    def setup_clients(cls):
        super(BaseRbacTest, cls).setup_clients()

        cls.hosts_client = cls.os_primary.hosts_client
        ...
```

This class is responsible for overriding the value of the primary Tempest credentials role (i.e. `os_primary` role). By doing so, it is possible to seamlessly swap between admin credentials, needed for setup and clean up, and primary credentials, needed to perform the API call which does policy enforcement. The primary credentials always cycle between roles defined by `CONF.identity.admin_role` and `CONF.patrole.rbac_test_roles`.

admin_roles_client = None

credentials = ['primary', 'admin']

get_all_needed_roles(*roles*)

Extending given roles with roles from mapping

Examples:: [admin] » [admin, member, reader] [member] » [member, reader] [reader] » [reader] [custom_role] » [custom_role]

Parameters **roles** list of roles

Returns extended list of roles

classmethod get_auth_providers()

Returns list of auth_providers used within test.

Tests may redefine this method to include their own or third party client auth_providers.

override_role()

Override the role used by `os_primary` Tempest credentials.

Temporarily change the role used by `os_primary` credentials to:

- `[patrole] rbac_test_roles` before test execution
- `[identity] admin_role` after test execution

Automatically switches to admin role after test execution.

Returns None

> **Warning:** This function can alter user roles for pre-provisioned credentials. Work is underway to safely clean up after this function.

Example:

```
@rbac_rule_validation.action(service='test',
                             rules=['a:test:rule'])
def test_foo(self):
    # Allocate test-level resources here.
    with self.override_role():
        # The role for `os_primary` has now been overridden. Within
        # this block, call the API endpoint that enforces the
        # expected policy specified by "rule" in the decorator.
        self.foo_service.bar_api_call()
    # The role is switched back to admin automatically. Note that
    # if the API call above threw an exception, any code below this
    # point in the test is not executed.
```

override_role_and_validate_list(*admin_resources=None*, *admin_resource_id=None*)

Call `override_role` and validate RBAC for a list API action.

List actions usually do soft authorization: partial or empty response bodies are returned instead of exceptions. This helper validates that unauthorized roles only return a subset of the available resources. Should only be used for validating list API actions.

Parameters

- **test_obj** Instance of `tempest.test.BaseTestCase`.
- **admin_resources** (*list*) The list of resources received before calling the `override_role_and_validate_list` function.
- **admin_resource_id** (*UUID*) An ID of a resource created before calling the `override_role_and_validate_list` function.

Returns py:class:*_ValidateListContext* object.

Example:

```
# the resource created by admin
admin_resource_id = (
    self.ntp_client.create_dscp_marking_rule()
    ["dscp_marking_rule"]["id'])
with self.override_role_and_validate_list(
        admin_resource_id=admin_resource_id) as ctx:
    # the list of resources available for member role
    ctx.resources = self.ntp_client.list_dscp_marking_rules(
        policy_id=self.policy_id)["dscp_marking_rules"]
```

classmethod restore_roles()

classmethod setup_clients()

patrole_tempest_plugin.rbac_utils.**is_admin**()

Verifies whether the current test role equals the admin role.

Returns True if `rbac_test_roles` contain the admin role.

patrole_tempest_plugin.requirements_authority module

class patrole_tempest_plugin.requirements_authority.**RequirementsAuthority**(*filepath=None*, *component=None*)

Bases: *patrole_tempest_plugin.rbac_authority.RbacAuthority*

A class that uses a custom requirements file to validate RBAC.

allowed(*rule_name*, *roles*)

Checks if a given rule in a policy is allowed with given role.

Parameters

- **rule_name** (*string*) Rule to be checked using provided requirements file specified by `[patrole].custom_requirements_file`. Must be a key present in this file, under the appropriate component.
- **roles** (*List[string]*) Roles to validate against custom requirements file.

Returns True if `role` is allowed to perform `rule_name`, else False.

Return type bool

Raises *RbacParsingException* If `rule_name` does not exist among the keyed policy names in the custom requirements file.

class patrole_tempest_plugin.requirements_authority.**RequirementsParser**(*filepath*)

Bases: `object`

A class that parses a custom requirements file.

class Inner(*filepath*)

Bases: `object`

static parse(*component*)

Parses a requirements file with the following format:

```
<service_foo>:
  <api_action_a>:
    - <allowed_role_1>
    - <allowed_role_2>,<allowed_role_3>
    - <allowed_role_3>
  <api_action_b>:
    - <allowed_role_2>
    - <allowed_role_4>
<service_bar>:
  <api_action_c>:
    - <allowed_role_3>
```

Parameters **component** (*str*) Name of the OpenStack service to be validated.

Returns The dictionary that maps each policy action to the list of allowed roles, for the given `component`.

Return type dict

Module contents

CHAPTER

SIX

SEARCH

- OpenStack wide search: Search the wider set of OpenStack documentation, including forums.

PYTHON MODULE INDEX

p

INDEX

O

P

R

S

V