insights-core Documentation

Release 0.0.1

Author

Apr 01, 2019

Contents

Contents:

CHAPTER 1

Preparing Your Development Environment

1.1 Begin Setup

First you need to ensure you have the `git` and `gcc` tools available. On Red Hat Enterprise Linux you can do this with the command, as root: `yum install git gcc`.

Now create your own fork of the insights-core-tutorials project. Do this by going to the *insights-core-tutorials* Repository on GitHub and clicking on the **Fork** button.

You will now have an *insights-core-tutorials* repository under your GitHub user that you can use to checkout the code to your development environment. To check out the code go to the repository page for your fork and copy the link to download the repo.

Tip: If you don't have a GitHub account or you are not interested in checking your modified code into GitHub right now then you can skip the forking step and clone the *insights-core-tutorials* repo directly using the HTTPS URL `https://github.com/RedHatInsights/insights-core-tutorials.git` with the `git clone` command below instead of the URL `git@github...`.

Once you have copied this link then go to a terminal in your working directory and use the `git` command to clone the repository. In this tutorial we will be using `work` as the directory that will contain the *insights-core-tutorials* project. If you choose to use a different root directory you will need to substitute `work` with your chosen root directory when referenced in the tutorial. So for the purposes of this document our working directory is `/home/userone/work`:

```
[userone@hostone ~]$ mkdir work
[userone@hostone ~]$ cd work
[userone@hostone work]$ git clone git@github.com:userone/insights-core-tutorials.git
Cloning into 'insights-core-tutorials'...
remote: Counting objects: 21251, done.
remote: Compressing objects: 100% (88/88), done.
remote: Total 21251 (delta 68), reused 81 (delta 43), pack-reused 21118
Receiving objects: 100% (21251/21251), 5.95 MiB | 2.44 MiB/s, done.
Resolving deltas: 100% (15938/15938), done.
```

Next you need to run the `setup_env.sh` script to set up your python environment:

```
[userone@hostone work]$ cd insights-core-tutorials
[userone@hostone insights-core-tutorials]$ setup_env.sh
```

If you ran the script to setup you environment you can skip the manual setup instructions and go straight to the *Setup Complete*.

1.2 Manual Setup (Not required if you ran setup_env.sh)

If you would rather create the development environment `mycomponents` manually you can follow these steps to create a virtual environment and set it up for development:

```
[userone@hostone work]$ cd insights-core-tutorials
[userone@hostone insights-core-tutorials]$ python3.6 -m venv .
Running virtualenv with interpreter /usr/bin/python3.6
Using base prefix '/usr'
New python executable in /home/userone/work/insights-core-tutorials/bin/python3.6
Also creating executable in /home/userone/work/insights-core-tutorials/bin/python
Installing setuptools, pip, wheel...done.

New python executable in ./bin/python
Installing Setuptools.....................................................done.
Installing Pip............................................................done.
```

Setup your environment to use the new virtualenv you just created, and upgrade `pip` to the latest version:

```
[userone@hostone insights-core-tutorials]$ source ./bin/activate
(env)[userone@hostone insights-core-tutorials]$ pip install --upgrade pip
```

Now install all of the required packages for *insights-core-tutorials* development:

```
(env)[userone@hostone insights-core-tutorials]$ pip install -e .[develop]
```

Next you will need to create `mycomponents` directory and directories to develop each of the components (parsers, combiners and rules) in.

The following are the commands to create the `mycomponents` development environment:

```
(env)[userone@hostone insights-core-tutorials]$ mkdir ./mycomponents
(env)[userone@hostone insights-core-tutorials]$ cd ./mycomponents
(env)[userone@hostone mycomponents]$ touch __init__.py
(env)[userone@hostone mycomponents]$ mkdir -p ./parsers/tests
(env)[userone@hostone mycomponents]$ touch ./parsers/__init__.py
(env)[userone@hostone mycomponents]$ touch ./parsers/tests/__init__.py
(env)[userone@hostone mycomponents]$ mkdir -p ./combiners/tests
(env)[userone@hostone mycomponents]$ touch ./combiners/__init__.py
(env)[userone@hostone mycomponents]$ touch ./combiners/tests/__init__.py
(env)[userone@hostone mycomponents]$ mkdir -p ./rules/tests
(env)[userone@hostone mycomponents]$ touch ./rules/__init__.py
(env)[userone@hostone mycomponents]$ touch ./rules/tests/__init__.py
(env)[userone@hostone mycomponents]$ export PYTHONPATH=$PWD:$PYTHONPATH
```

Once you have completed the setup of the environment by either running the provided script or running the setup steps manually, you will have a complete development environment for rules, parsers, combiners and for your mycomponents development directory.

You can now confirm that everything is setup correctly so far by running the tests, `pytest`.

If you ran the `setup_env.sh` script the pytests will have been run by the script, you should see the results in console when the script finishes.

This will test the components in the `insights_examples` directory. Your results should look something like this:

```
(env)[userone@hostone insights-core-tutorials]$ pytest
====================== test session starts ==========================
platform linux -- Python 3.6.6, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /home/userone/work/insights-core-tutorials, inifile: setup.cfg
plugins: cov-2.6.1
collected 10 items

insights_examples/combiners/tests/test_hostname_uh.py .
insights_examples/parsers/tests/test_secure_shell.py ...
insights_examples/rules/tests/integration.py ...
insights_examples/rules/tests/test_sshd_secure.py .

=================== 10 passed in 0.30 seconds =======================
```

1.3 Setup Complete

If during this step you see a test failure similar to the following make sure you have `unzip` installed on your system:

```
>           raise child_exception
E           CalledProcessError: <CalledProcessError(0, ['unzip', '-q', '-d',
'/tmp/tmplrXhIu', '/tmp/test.zip'], [Errno 2] No such file or directory)>

/usr/lib64/python2.7/subprocess.py:1327: CalledProcessError
```

Your development environment is now ready to begin development and you may move on to the next section. If you had problems with any of these steps then double check that you have completed all of the steps in order and if it still doesn't work, open a GitHub issue.

CHAPTER 2

Tutorial - Custom Parser

2.1 Parser Development

The purpose of a Parser is to process raw content collected by the Client and map it into a format that is usable by Combiners and Rules. Raw content is content obtained directly from a system file or command, and may be collected by Insights Client, or from some other source such as a SOS Report. The following examples will demonstrate development of different types of parsers.

You can find the complete implementation of the parser and test code in the directory `insights-core-tutorials/insights_examples/parsers`.

2.1.1 Secure Shell Parser

2.1.1.1 Overview

Secure Shell or `ssh` ("SSH") is a commonly used tool to access and interact with remote systems. SSH server is configured on a system using the `/etc/sshd_config` file. Red Hat Enterprise Linux utilizes OpenSSH and the documentation for the `/etc/sshd_config` file is located here.

Here is a portion of the configuration file showing the syntax:

```
#   $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

Port 22
#AddressFamily any
ListenAddress 10.110.0.1
#ListenAddress ::

# The default requires explicit activation of protocol 1
#Protocol 2
```

Many lines begin with a # indicating comments, and blank lines are used to aid readability. The important lines have a configuration keyword followed by space and then a configuration value. So in the parser we want to make sure we capture the important lines and ignore the comments and blank lines.

2.1.1.2 Creating the Initial Parser Files

First we need to create the parser file. Parser files are implemented in modules. The module should be limited to one type of application. In this case we are working with the `ssh` application so we will create an `secure_shell` module. Create the module file `~/work/insights-core-tutorials/mycomponents/parsers/secure_shell.py` in the parsers directory:

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials/mycomponents/parsers
(env)[userone@hostone parsers]$ touch secure_shell.py
```

Now edit the file and create the parser skeleton:

```
from insights import Parser, parser
from insights.specs import Specs


@parser(Specs.sshd_config)
class SSHDConfig(Parser):

    def parse_content(self, content):
        pass
```

We start by importing the `Parser` class and the `parser` decorator. Our parser will inherit from the `Parser` class and it will be associated with the `Specs.sshd_config` data source using the `parser` decorator. Finally we need to implement the `parse_content` subroutine which is required to parse and store the input data in our class. The base class `Parser` implements a constructor that will invoke our `parse_content` method when the class is created.

Next we'll create the parser test file `~/work/insights-core-tutorials/mycomponents/parsers/tests/test_secure_shell.py` as a skeleton that will aid in the parser development process:

```
from mycomponents.parsers.secure_shell import SSHDConfig


def test_sshd_config():
    pass
```

Once you have created and saved both of these files and we'll run the test to make sure everything is setup correctly:

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials
(env)[userone@hostone insights-core-tutorials]$ pytest -k secure_shell

======================== test session starts ============================
platform linux2 -- Python 2.7.15, pytest-3.0.6, py-1.7.0, pluggy-0.4.0
rootdir: /home/userone/work/mycomponents, inifile:
plugins: cov-2.4.0
collected 3 items

mycomponents/parsers/tests/test_secure_shell.py ...

===================== 3 passed in 1.26 seconds ==========================
```

Hint: You may sometimes see a message that `pytest` cannot be found, or see some other related message that doesn't make sense. The first think to check is that you have activated your virtual environment by executing the command `source bin/activate` from the root directory of your insights-core-tutorials project. You can deactivate the virtual environment by typing `deactivate`. You can find more information about virtual environments here: http://docs.python-guide.org/en/latest/dev/virtualenvs/

2.1.1.3 Parser Implementation

Typically parser and combiner development is driven by rules that need facts generated by the parsers and combiners. Regardless of the specific requirements, it is important (1) to implement basic functionality by getting the raw data into a usable format, and (2) to not overdo the implementation because we can't anticipate every use of the parser output. In our example we will eventually be implementing the rules that will warn us about systems that are not configured properly. Initially our parser implementation will be parsing the input data into key/value pairs. We may later discover that we can optimize rules by moving duplicate or complex processing into the parser.

Test Code

Referring back to our *sample SSHD input* we will start by creating a test for the output that we want from our parser:

```
from mycomponents.parsers.secure_shell import SSHDConfig
from insights.tests import context_wrap

SSHD_CONFIG_INPUT = """
#    $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

Port 22
#AddressFamily any
ListenAddress 10.110.0.1
Port 22
ListenAddress 10.110.1.1
#ListenAddress ::

# The default requires explicit activation of protocol 1
#Protocol 2
Protocol 1
"""


def test_sshd_config():
    sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
    assert sshd_config is not None
    assert 'Port' in sshd_config
    assert 'PORT' in sshd_config
    assert sshd_config['port'] == ['22', '22']
    assert 'ListenAddress' in sshd_config
    assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']
    assert sshd_config['Protocol'] == ['1']
    assert 'AddressFamily' not in sshd_config
    ports = [l for l in sshd_config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'
```

First we added an import for the helper function `context_wrap` which we'll use to put our input data into a `Context` object to pass to our class constructor:

```
from mycomponents.parsers.secure_shell import SSHDConfig
from insights.tests import context_wrap
```

Next we include the sample data that will be used for the test. Use of the `strip()` function ensures that all white space at the beginning and end of the data are removed:

```
SSHD_CONFIG_INPUT = """
#    $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

Port 22
#AddressFamily any
ListenAddress 10.110.0.1
Port 22
ListenAddress 10.110.1.1
#ListenAddress ::

# The default requires explicit activation of protocol 1
#Protocol 2
Protocol 1
"""
```

Next, to the body of the test, we add code to create an instance of our parser class:

```
def test_sshd_config():
    sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
```

Finally we add our tests using the attributes that we want to be able to access in our rules. First a assumptions about the data:

1. some keywords may be present more than once in the config file
2. we want to access keywords in a case insensitive way
3. order of the keywords matter
4. we are not trying to validate the configuration file so we won't parse the values or analyze sequence of keywords

Now here are the tests:

```
    assert sshd_config is not None
    assert 'Port' in sshd_config
    assert 'PORT' in sshd_config
    assert sshd_config['port'] == ['22', '22']
    assert 'ListenAddress' in sshd_config
    assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.0.1']
    assert sshd_config['Protocol'] == ['1']
    assert 'AddressFamily' not in sshd_config
    ports = [l for l in sshd_config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'
```

Our tests assume that we want to know whether a particular keyword is present, regardless of character case used in the keyword, and we want to know the values of the keyword if present. We don't want our rules to have to assume any particular case of characters in keywords so we can make it easy by performing case insensitive compares and assuming all lowercase for access. This may not always work, but in this example it is a safe assumption.

Parser Code

The subroutine `parse_content` is responsible for parsing the input data and storing the results in class attributes. You may choose the attributes that are necessary for your parser, there are no requirements to use specific names or types. Some general recommendations for parser class implementation are:

- Choose attributes that make sense for use by actual rules, or how you anticipate rules to use the information. If rules need to iterate over the information then a `list` might be best, or if rules could access via keywords then `dict` might be better.
- Choose attribute types that are not so complex they cannot be easily understood or serialized. Unless you know you need something complex keep it simple.
- Use the `@property` decorator to create read-only getters and simplify access to information.

Now we need to implement the parser that will satisfy our tests.

```
from collections import namedtuple
from insights import Parser, parser, get_active_lines
from insights.core.spec_factory import SpecSet, simple_file
import os


class LocalSpecs(SpecSet):
    """ Datasources for collection from local host """
    conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
↪')

    sshd_config = simple_file(conf_file)


@parser(LocalSpecs.sshd_config)
class SSHDConfig(Parser):

    KeyValue = namedtuple('KeyValue', ['keyword', 'value', 'kw_lower'])

    def parse_content(self, content):
        self.lines = []
        for line in get_active_lines(content):
            kw, val = line.split(None, 1)
            self.lines.append(self.KeyValue(kw.strip(), val.strip(), kw.lower().
↪strip()))
        self.keywords = set([k.kw_lower for k in self.lines])

    def __contains__(self, keyword):
        return keyword.lower() in self.keywords

    def __iter__(self):
        for line in self.lines:
            yield line

    def __getitem__(self, keyword):
        kw = keyword.lower()
        if kw in self.keywords:
            return [kv.value for kv in self.lines if kv.kw_lower == kw]
```

We added an imports to our skeleton to utilize `get_active_lines()` and `namedtuples`. `get_active_lines()` is one of the many helper methods that you can find in `insights/parsers/__init__.py`, `insights/core/__init__.py`, and `insights/util/__init__.py`.

`get_active_lines()` will remove all blank lines and comments from the input which simplifies your parsers parsing logic.

```
from collections import namedtuple
from insights import Parser, parser, get_active_lines
from insights.core.spec_factory import SpecSet, simple_file
import os
```

Since the `sshd_config` spec requires root access to access the `/etc/ssh/sshd_config` file we created a local `SpecSet` class called `LocalSpecs` that will contain a local `sshd_config` spec that uses a local `sshd_config` file that does not require root access to read.

```
class LocalSpecs(SpecSet):
    """ Datasources for collection from local host """
    conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
↪')

    sshd_config = simple_file(conf_file)
```

To get the `ssh_config` file needed for the local sshd_config spec you can copy it from `~/work/insights-core-tutorials/insights_examples/parsers/sshd_config` to the `~/work/insights-core-tutorials/mycomponents/parsers` directory as shown below.

> (env)[userone@hostone insights-core-tutorials]$ cp ./insights_examples/parsers/sshd_config ./mycomponents/parsers/

We can use `namedtuples` to help simplify access to the information we are storing in our parser by creating a namedtuple with the named attributes `keyword`, `value`, and `kw_lower` where *kw_lower* is the lowercase version of the *keyword*.

```
    KeyValue = namedtuple('KeyValue', ['keyword', 'value', 'kw_lower'])
```

In this particular parser we have chosen to store all lines (`self.lines`) as `KeyValue` named tuples since we don't know what future rules might. We are also storing the `set` of lowercase keywords (`self.keywords`) to make it easier to determine if a keyword is present in the data. The values are left unparsed as we don't know how a rule might need to evaluate them.

```
    def parse_content(self, content):
        self.lines = []
        for line in get_active_lines(content):
            kw, val = line.split(None, 1)
            self.lines.append(self.KeyValue(kw.strip(), val.strip(), kw.lower().
↪strip()))
        self.keywords = set([k.kw_lower for k in self.lines])
```

Finally we implement some "dunder" methods to simplify use of the class. `__contains__` enables the `in` operator for keyword checking. `__iter__` enables iteration over the contents of `self.lines`. And `__getitem__` enables access to all values of a keyword.

```
    def __contains__(self, keyword):
        return keyword.lower() in self.keywords

    def __iter__(self):
        for line in self.lines:
            yield line

    def __getitem__(self, keyword):
```

(continues on next page)

(continued from previous page)

```
        kw = keyword.lower()
        if kw in self.keywords:
            return [kv.value for kv in self.lines if kv.kw_lower == kw]
```

We now have a complete implementation of our parser. It could certainly perform further analysis of the data and more methods for access, but it is better keep the parser simple in the beginning. Once it is in use by rules it will be easy to add functionality to the parser to allow simplification of the rules.

2.1.1.4 Parser Documentation

The last step to complete implementation of our parser is to create the documentation. The guidelines and examples for parser documentation is provided in the section Documentation Guidelines.

The following shows our completed parser including documentation.

```
"""
secure_shell - Files for configuration of `ssh`
===============================================

The ``secure_shell`` module provides parsing for the ``sshd_config``
file.  The ``SSHDConfig`` class implements the parsing and
provides a ``list`` of all configuration lines present in
the file.

Sample content from the ``/etc/sshd/sshd_config`` file is::

    #       $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

    Port 22
    #AddressFamily any
    ListenAddress 10.110.0.1
    Port 22
    ListenAddress 10.110.1.1
    #ListenAddress ::

    # The default requires explicit activation of protocol 1
    #Protocol 2
    Protocol 1

Examples:
    >>> 'Port' in sshd_config
    True
    >>> 'PORT' in sshd_config  # items are stored case-insensitive
    True
    >>> 'AddressFamily' in sshd_config  # comments are ignored
    False
    >>> sshd_config['port']  # All value stored by keyword in lists
    ['22', '22']
    >>> sshd_config['Protocol']  # Single items have one list element
    ['1']
    >>> [line for line in sshd_config if line.keyword == 'Port']  # can be used as␣
↪an iterator
    [KeyValue(keyword='Port', value='22', kw_lower='port'), KeyValue(keyword='Port',␣
↪value='22', kw_lower='port')]
    >>> sshd_config.last('ListenAddress')  # Easy way of finding the current␣
↪configuration for a single item
```

(continues on next page)

(continued from previous page)

```
    '10.110.1.1'
"""
from collections import namedtuple
from insights import Parser, parser, get_active_lines
from insights.specs import Specs
from insights.core.spec_factory import SpecSet, simple_file
import os


class LocalSpecs(SpecSet):
    """ Datasources for collection from local host """
    conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
↪')

    sshd_config = simple_file(conf_file)


@parser(LocalSpecs.sshd_config)
class SSHDConfig(Parser):
    """Parsing for ``sshd_config`` file.

    Attributes:
        lines (list): List of `KeyValue` namedtupules for each line in
            the configuration file.
        keywords (set): Set of keywords present in the configuration
            file, each keyword has been converted to lowercase.
    """

    KeyValue = namedtuple('KeyValue', ['keyword', 'value', 'kw_lower'])
    """namedtuple: Represent name value pair as a namedtuple with case ."""

    def parse_content(self, content):
        self.lines = []
        for line in get_active_lines(content):
            kw, val = (w.strip() for w in line.split(None, 1))
            self.lines.append(self.KeyValue(kw, val, kw.lower()))
        self.keywords = set([k.kw_lower for k in self.lines])

    def __contains__(self, keyword):
        return keyword.lower() in self.keywords

    def __iter__(self):
        for line in self.lines:
            yield line

    def __getitem__(self, keyword):
        kw = keyword.lower()
        if kw in self.keywords:
            return [kv.value for kv in self.lines if kv.kw_lower == kw]

    def last(self, keyword):
        """str: Returns the value of the last keyword found in config."""
        entries = self.__getitem__(keyword)
        if entries:
            return entries[-1]
```

2.1.1.5 Parser Testing

It is important that we ensure our tests will run successfully after any change to our parser. We are able to do that in two ways, first by using `doctest` to test our *Examples* section of the `secure_shell` module, and second by writing tests that can be tested automatically using `pytest`. Starting with adding `import doctest` our original code:

```
from mycomponents.parsers.secure_shell import SSHDConfig
from insights.parsers import secure_shell
from insights.tests import context_wrap
import doctest

SSHD_CONFIG_INPUT = """
#   $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

Port 22
#AddressFamily any
ListenAddress 10.110.0.1
Port 22
ListenAddress 10.110.1.1
#ListenAddress ::

# The default requires explicit activation of protocol 1
#Protocol 2
Protocol 1
"""

def test_sshd_config():
    sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
    assert sshd_config is not None
    assert 'Port' in sshd_config
    assert 'PORT' in sshd_config
    assert sshd_config['port'] == ['22', '22']
    assert 'ListenAddress' in sshd_config
    assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']
    assert sshd_config['Protocol'] == ['1']
    assert 'AddressFamily' not in sshd_config
    ports = [l for l in sshd_config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'
```

To test the documentation, we can then use `doctest`:

```
def test_sshd_documentation():
    """
    Here we test the examples in the documentation automatically using
    doctest.  We set up an environment which is similar to what a
    rule writer might see - a 'sshd_config' variable that has been
    passed in as a parameter to the rule declaration.  This saves doing
    this setup in the example code.
    """
    env = {
        'sshd_config': SSHDConfig(context_wrap(SSHD_CONFIG_INPUT)),
    }
    failed, total = doctest.testmod(secure_shell, globs=env)
    assert failed == 0
```

The environment setup allows us to 'hide' the set-up of the environment that normally provided to the rule, which is

the context in which the example code is written. There's no easy way to show the declaration of the rule, nor the parameter that is created with the parser object, but it's good practice to supply an obvious name that rule writers might then use in their code.

The `assert` line here makes sure that any failures in the examples are detected by pytest. This will also include the testing output from doctest, showing where the code failed to evaluate or where the output differed from what was given.

Because this code essentially duplicates many of the things previously tested explicitly in the `test_sshd_config` function, we can remove some of those tests and only test the 'corner cases':

```
SSHD_DOCS_EXAMPLE = '''
Port 22
Port 22
'''

def test_sshd_corner_cases():
    """
    Here we test any corner cases for behavior we expect to deal with
    in the parser but doesn't make a good example.
    """
    config = SSHDConfig(context_wrap(SSHD_DOCS_EXAMPLE))
    assert config.last('AddressFamily') is None
    assert config['AddressFamily'] is None
    ports = [l for l in config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'
```

The final version of our test now looks like this:

```
from mycomponets.parsers.secure_shell import SSHDConfig
from insights.parsers import secure_shell
from insights.tests import context_wrap
import doctest

SSHD_CONFIG_INPUT = """
#   $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $

Port 22
#AddressFamily any
ListenAddress 10.110.0.1
Port 22
ListenAddress 10.110.1.1
#ListenAddress ::

# The default requires explicit activation of protocol 1
#Protocol 2
Protocol 1
"""

def test_sshd_config():
    sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
    assert sshd_config is not None
    assert 'Port' in sshd_config
    assert 'PORT' in sshd_config
    assert sshd_config['port'] == ['22', '22']
    assert 'ListenAddress' in sshd_config
    assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']
```

(continues on next page)

(continued from previous page)

```
    assert sshd_config['Protocol'] == ['1']
    assert 'AddressFamily' not in sshd_config
    ports = [l for l in sshd_config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'


def test_sshd_documentation():
    """
    Here we test the examples in the documentation automatically using
    doctest.  We set up an environment which is similar to what a
    rule writer might see - a 'sshd_config' variable that has been
    passed in as a parameter to the rule declaration.  This saves doing
    this setup in the example code.
    """
    env = {
        'sshd_config': SSHDConfig(context_wrap(SSHD_CONFIG_INPUT)),
    }
    failed, total = doctest.testmod(secure_shell, globs=env)
    assert failed == 0


SSHD_DOCS_EXAMPLE = '''
Port 22
Port 22
'''


def test_sshd_corner_cases():
    """
    Here we test any corner cases for behavior we expect to deal with
    in the parser but doesn't make a good example.
    """
    config = SSHDConfig(context_wrap(SSHD_DOCS_EXAMPLE))
    assert config.last('AddressFamily') is None
    assert config['AddressFamily'] is None
    ports = [l for l in config if l.keyword == 'Port']
    assert len(ports) == 2
    assert ports[0].value == '22'
```

To run `pytest` on just the completed `secure_shell` parser execute the following command:

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials
(env)[userone@hostone insights-core-tutorials]$ pytest -k secure_shell
```

Once your tests all run successfully your parser is complete.

CHAPTER 3

Tutorial - Custom Combiner Development

In the Map-Reduce model, Parsers are responsible for mapping the data from a datasource and Combiners are responsible for reducing the data from multiple Parsers into a reduced dataset. Combiners help to consolidate information from different data sources, hide differences between the same data source across operating system versions, and make Parser output more rule friendly.

For example, the *hostname* of a system may be obtained from the `Hostname`, `Facter`, and `SystemID` parsers. A rule could rely on all three parsers and might need to process each one differently depending upon which was present. However, a rule could instead simply rely on the `hostname` combiner which does the job of evaluating all of the parser data and determining the best source of *hostname*, greatly simplifying logic in the rule.

Another example is the `GrubConf` combiner which evaluates multiple versions (1, 2, non-EFI, EFI) Grub configuration data to provide one consolidated source of information for Grub configuration on a system.

You can find the complete implementation of the combiner and test code in the directory `insights-core-tutorials/insights_examples/combiners`.

3.1 Hostname Combiner

3.1.1 Overview

The same development environment will be used that was setup at the beginning of the tutorial using the *Preparing Your Development Environment* section.

For this tutorial we will create a new *hostname* combiner that will consolidate information from the `insights.parsers.uname.Uname` and `insights.parsers.hostname.Hostname` parsers. There is an existing `insights.combiners.hostname.Hostname` combiner so we will call ours `HostnameUH` to avoid confusion.

3.1.2 Creating the Initial Combiner Files

First we need to create the combiner file. Combiner files are implemented in modules. The module should be limited to one purpose. In this case we are working with `hostname` data so we will create an `hostname_uh` module. Also there is already a `hostname` combiner module so we want to avoid confusion. Create the module file `mycomponents/combiners/hostname_uh.py` in the `mycomponents/combiners directory`:

```
(env)[userone@hostone mycomponents]$ touch combiners/hostname_uh.py
```

Now edit the file and create the combiner skeleton:

```
from insights.core.plugins import combiner
from insights.parsers.hostname import Hostname
from insights.parsers.uname import Uname


@combiner([Hostname, Uname])
class HostnameUH(object):

    def __init__(self, hostname, uname):
        pass
```

We start by importing the `combiner` decorator. As discussed above our combiner will depend upon the `Hostname` and `Uname` parsers and these are imported and included as arguments to the `combiner` decorator. Notice that the decorator arguments are in a `list` meaning that our combiner needs at least one of the parsers in the list. See the discussion of *Rule Decorators* for more information on *required*, *at least one*, and *optional* arguments to the combiner decorator.

We also need to pass the parser instance objects as arguments to the `__init__` method of our combiner. If either of these objects is not present then its value with be `None`.

Next we'll create the combiner test file `mycomponents/combiners/tests/test_hostname_uh.py` as a skeleton that will aid in the combiner development process:

```
from mycomponents.combiners.hostname_uh import HostnameUH


def test_hostname_uh():
    pass
```

Once you have created and saved both of these files, you can the test to make sure everything is setup correctly:

```
(env)[userone@hostone insights-core-tutorials]$ pytest -k hostname_uh
======================= test session starts ============================
   platform linux -- Python 3.6.6, pytest-3.0.6, py-1.7.0, pluggy-0.4.0
   rootdir: /home/userone/work/insights-core-tutorials, inifile:

   collected 16 items / 14 deselected

   insights_examples/combiners/tests/test_hostname_uh.py .
↪                                                                    [ 50%]
↪
   mycomponents/combiners/tests/test_hostname_uh.py .

============ 2 passed, 14 deselected in .27 seconds ====================
```

When you invoke `pytest` with the `-k` option it will only run tests which match the filter, in this case tests that match *hostname_uh*. So our test passed as expected.

Hint: You may sometimes see a message that `pytest` cannot be found, or see some other related message that doesn't make sense. The first think to check is that you have activated your virtual environment by executing the command `source bin/activate` from the root directory of your insights-core-tutorials project. You can deactivate the virtual environment by typing `deactivate`. You can find more information about virtual environments here: http://docs.python-guide.org/en/latest/dev/virtualenvs/

3.1.3 Combiner Implementation

Typically parser and combiner development is driven by rules that need facts generated by the parsers and combiners. Regardless of the specific requirements, it is important (1) to implement basic functionality by getting the raw data into a usable format, and (2) to not overdo the implementation because we can't anticipate every use of the combiner output. In our example the output is simple, but some combiners can be complicated so keep these two criteria in mind when developing new parsers or combiners. You can always add more capability later on if needed by your rules.

3.1.3.1 Test Code

We will start by creating a test for the output that we want from our combiner using the two input sources. You can look at the documentation for `insights.parsers.hostname` and `insights.parsers.uname` to see what methods will be available. In our tests we want to ensure that we can test with the parser object so we'll use input data to feed the parsers and then use the parsers as input to our combiner tests.

```
from mycomponents.combiners.hostname_uh import HostnameUH
from insights.parsers.hostname import Hostname
from insights.parsers.uname import Uname
from insights.tests import context_wrap

HOSTNAME = "hostone_h.example.com"
UNAME = "Linux hostone_u.example.com 3.10.0-693.21.1.el7.x86_64 #1 SMP Fri Feb 23␣
↪18:54:16 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux"


def test_hostname_uh():
    hostname = Hostname(context_wrap(HOSTNAME))
    uname = Uname(context_wrap(UNAME))

    hostname_uh = HostnameUH(hostname, None)
    assert hostname_uh.hostname == HOSTNAME

    hostname_uh = HostnameUH(None, uname)
    assert hostname_uh.hostname == "hostone_u.example.com"

    hostname_uh = HostnameUH(hostname, uname)
    assert hostname_uh.hostname == HOSTNAME
```

First we added an import for the combiner object and the parser objects. Next we import a helper function `context_wrap` which we'll use to create our parser instance objects:

```
from insights.combiners.hostname_uh import HostnameUH
from insights.parsers.hostname import Hostname
from insights.parsers.uname import Uname
from insights.tests import context_wrap
```

Next we include the sample data that will be used for the test. We will use data for input to the parsers so we need both sample outputs of the `hostname` command and the `uname -a` command:

```
HOSTNAME = "hostone_h.example.com"
UNAME = "Linux hostone_u.example.com 3.10.0-693.21.1.el7.x86_64 #1 SMP Fri Feb 23␣
↪18:54:16 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux"
```

Next, to the body of the test, we add code to create instances of the necessary parser classes:

```
def test_hostname_uh():
    hostname = Hostname(context_wrap(HOSTNAME))
    uname = Uname(context_wrap(UNAME))
```

Finally we add our tests using the attributes that we want to be able to access in our rules. For our combiner we trust `hostname` more than `uname` so we give `hostname` priority by checking it first and then fall back to `uname` if hostname is not available. If neither of these is available the combiner will not be called. It is always guaranteed that our combiner will get at least one of the parsers when called.

Now here are the tests:

```
hostname_uh = HostnameUH(hostname, None)
assert hostname_uh.hostname == HOSTNAME

hostname_uh = HostnameUH(None, uname)
assert hostname_uh.hostname == "hostone_u.example.com"

hostname_uh = HostnameUH(hostname, uname)
assert hostname_uh.hostname == HOSTNAME
```

We use a different hostname in each parser so that we can confirm that the correct parser data is chosen.

3.1.3.2 Combiner Code

The class `__init__` method performs all of the work in our combiner. If your combiner is more complex you may need to add additional methods and utility functions. Some general recommendations for the combiner class implementation are:

- Choose attributes that make sense for use by actual rules, or how you anticipate rules to use the information. If rules need to iterate over the information then a `list` might be best, or if rules could access via keywords then `dict` might be better.
- Choose attribute types that are not so complex they cannot be easily understood or serialized. Unless you know you need something complex keep it simple.
- Use the `@property` decorator to create read-only getters and simplify access to information.

Now we need to implement the combiner that will satisfy our tests.

```
from insights.core.plugins import combiner
from insights.parsers.hostname import Hostname
from insights.parsers.uname import Uname


@combiner([Hostname, Uname])
class HostnameUH(object):

    def __init__(self, hostname, uname):
        if hostname:
```

(continues on next page)

(continued from previous page)

```
            self.hostname = hostname.fqdn
        else:
            self.hostname = uname.nodename
```

We've replaced our original `__init__` to include the logic for our combiner. The `Hostname` parser is passed in as the `hostname` attribute, and if it is present then we use it to acquire the hostname data. If `hostname` is `None`, meaning that there was no data or there was some error in the data for the `Hostname` parser, we fall back to use the `Uname` parser data passed in the `uname` attribute.

Now save this file and run the tests again to confirm that we have successfully written our combiner to pass all tests:

```
(env)[userone@hostone insights-core-tutorials]$ pytest -k hostname_uh
======================== test session starts ==============================
   platform linux -- Python 3.6.6, pytest-3.0.6, py-1.7.0, pluggy-0.4.0
   rootdir: /home/userone/insights-core-tutorials, inifile: setup.cfg
   plugins: cov-2.6.1
   collected 6 items / 5 deselected

   insights_examples/combiners/tests/test_hostname_uh.py .
↪                                                                        [ 50%]
↪
   mycomponents/combiners/tests/test_hostname_uh.py .

============ 2 passed, 14 deselected in 0.35 seconds ====================
```

3.1.4 Combiner Documentation and Testing

The last step to complete implementation of our combiner is to create the documentation. The guidelines and examples for combiner documentation is provided in the section Documentation Guidelines and parallels the information provided in the instructions for *Parser Documentation*. Combiner testing parallels the information provided in the instructions for the *Parser Testing*

CHAPTER 4

Tutorial - Rule Development

In this tutorial we'll walk through authoring a new insights-core rule.

There are three primary phases to developing a rule:

1. Determine rule logic and identify parsers and combiners
2. Develop the rule
3. Develop the tests

We'll cover each step in detail in the sections ahead, so let's get started!

4.1 Rule Using Existing Parser and Combiner

4.1.1 Determine rule logic

The most effective way to get started in developing a rule is first to identify the problem you want to address.

For the purposes of this tutorial we'll look at a very simple scenario. Sometimes when researching an issue one of the things that we might need to know is which Red Hat OS the host is running. For simplicity sake, in this example we will concentrate only on determining if the red hat release is `Fedora`.

For this case there is one thing we need to check:

1. Is the Red Hat release `Fedora`:

4.1.2 Identify Parsers

- We can check Red Hat Release using the `RedhatRelease` parser.

4.1.3 Develop Plugin

Now that we have identified the required parsers, let's get started on developing our plugin.

Create a file called `is_fedora.py` in a Python package called `tutorial`.

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials/mycomponents/rules
(env)[userone@hostone rules]$ touch is_fedora.py
```

Open `is_fedora.py` in your text editor of choice and start by stubbing out the rule function and imports.

```
from insights.parsers.redhat_release import RedhatRelease
from insights import rule, make_fail, make_pass

@rule(RedhatRelease)
def report(rhrel):
    pass
```

Let's go over each line and describe the details:

```
from insights.parsers.redhat_release import RedhatRelease
```

Parsers you want to use must be imported. You must pass the parser class objects directly to the `@rule` decorator to declare them as dependencies for your rule.

```
from insights import rule, make_fail, make_pass
```

`rule` is a function decorator used to specify your main plugin function. Combiners have a set of optional dependencies that are specified via the `requires` kwarg.

`make_fail, make_pass` are formatting functions used to format the return value of a rule function.

```
ERROR_KEY_IS_FEDORA = "IS_FEDORA"

CONTENT = {
    ERROR_KEY_IS_FEDORA: "This machine ({{hostname}}) runs {{product}}.",
}
```

Here we defined the `Jinga` template for message to be displayed for the response tag for either pass or fail

```
@rule(RedhatRelease)
```

Here we are specifying that this rule requires the output of the `insights.parsers.redhat_release.RedhatRelease`,

Now let's add the rule logic

```
@rule(RedhatRelease, content=CONTENT)
def report(rhrel):
    """Fires if the machine is running Fedora."""

    if "Fedora" in rel.product:
        return make_pass(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,␣
↪product=rel.product)
    else:
        return make_fail(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,␣
↪product=rel.product)
```

Now lets look at what the rule is doing.

The `RedhatRelease` parser parses content from the `/etc/redhat-release` file on the host it is running on and returns an object containing the Red Hat OS information for the host.

```
    if "Fedora" in rhrel.product:
        return make_pass(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,␣
↪product=rel.product)
    else:
        return make_fail(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,␣
↪product=rel.product)
```

Here we check to see if the value `Fedora` is in the "product" property of the "rhrel" object. If true then the rule returns a response telling us that the host is indeed running `Fedora`, along with the product information returned by the parser. If false then the rule returns a response telling us that the host is not running `Fedora`, along with the product information returned by the parser.

4.1.4 Develop Tests

Start out by creating a `test_is_fedora.py` module in a `tests` package.

```
(env) [userone@hostone ~]$ cd ~/work/insights-core-tutorials/rules/tests
(env) [userone@hostone tests]$ touch __init__.py
(env) [userone@hostone tests]$ touch test_is_fedora.py
```

Open `test_is_fedora.py` in your text editor of choice and start by stubbing out a test and the required imports.

```
from .. import is_fedora
from insights.specs import Specs
from insights.tests import InputData, archive_provider
from insights.core.plugins import make_fail, make_pass


@archive_provider(is_fedora.report)
def integration_test():
    pass
```

The framework provides an integration test framework that allows you to define an `InputData` object filled with raw examples of files required by your rule and an expected response. The object is evaluated by the pipeline as it would be in a production context, after which the response is compared to your expected output.

The `@archive_provider` decorator registers your test function with the framework. This function must be a generator that yields `InputData` and an expected response in a two tuple. The `@archive_provider` decorator takes one parameter, the rule function to test.

The bulk of the work in building a test for a rule is in defining the `InputData` object. If you remember our rule we accept `RedhatRelease`. We will define a data snippet for each test.

```
FEDORA = "Fedora release 28 (Twenty Eight)".strip()
RHEL = "Red Hat Enterprise Linux Server release 7.4 (Maipo)".strip()
TEST_HOSTNAME = "testhost.someplace.com"
```

Next for each test we need to build `InputData` objects and populate it with the content and build the expected return. Then finally we need to yield the pair.

```
input_data = InputData("test_fedora")
input_data.add(Specs.redhat_release, FEDORA)
```

(continues on next page)

(continued from previous page)

```
    input_data.add(Specs.hostname, TEST_HOSTNAME)
    expected = make_pass("IS_FEDORA", hostname=TEST_HOSTNAME, product="Fedora")

    yield input_data, expected

    input_data = InputData("test_rhel")
    input_data.add(Specs.redhat_release, RHEL)
    input_data.add(Specs.hostname, TEST_HOSTNAME)
    expected = make_fail("IS_FEDORA", hostname=TEST_HOSTNAME, product="Red Hat␣
↪Enterprise Linux Server")

    yield input_data, expected
```

Now for the entire test:

```
from .. import is_fedora
from insights.specs import Specs
from insights.tests import InputData, archive_provider
from insights.core.plugins import make_fail, make_pass

FEDORA = "Fedora release 28 (Twenty Eight)"
RHEL = "Red Hat Enterprise Linux Server release 7.4 (Maipo)"
TEST_HOSTNAME = "testhost.someplace.com"


@archive_provider(is_fedora.report)
def integration_test():

    input_data = InputData("test_fedora")
    input_data.add(Specs.redhat_release, FEDORA)
    input_data.add(Specs.hostname, TEST_HOSTNAME)
    expected = make_pass("IS_FEDORA", hostname=TEST_HOSTNAME, product="Fedora")


    yield input_data, expected

    input_data = InputData("test_rhel")
    input_data.add(Specs.redhat_release, RHEL)
    input_data.add(Specs.hostname, TEST_HOSTNAME)
    expected = make_fail("IS_FEDORA", hostname=TEST_HOSTNAME, product="Red Hat␣
↪Enterprise Linux Server")

    yield input_data, expected
```

4.2 Rule Using Custom Parser

4.2.1 Overview

The purpose of a rule is to evaluate various facts and determine one or more results about a system. For our example rule we are interested in knowing whether a system with `sshd` is configured according to the following guidelines:

```
# Password based logins are disabled - only public key based logins are allowed.
AuthenticationMethods publickey
```

(continues on next page)

(continued from previous page)

```
# LogLevel VERBOSE logs user's key fingerprint on login. Needed to have
# a clear audit track of which key was using to log in.
LogLevel VERBOSE

# Root login is not allowed for auditing reasons. This is because it's
# difficult to track which process belongs to which root user:
PermitRootLogin No

# Use only protocol 2 which is the default.  1 should not be listed
# Protocol 2
```

We also want to know what version of OpenSSH we are running if we find any problems.

You can find the complete implementation of the rule and test code in the directory `insights-core-tutorials/insights_examples/rules`.

The same development environment will be used that was setup at the beginning of the tutorial using the *Preparing Your Development Environment* section.

4.2.2 Secure Shell Server Rule

4.2.2.1 Rule Code

First we need to create a template rule file. It is recommended that you name the file based on the results it produces. Since we are looking at sshd security we will name the file `mycomponents/rules/sshd_secure.py`. Notice that the file is located in the `rules` subdirectory of your project:

```
(env)[userone@hostone mycomponents]$ touch rules/sshd_secure.py
```

Here's the basic contents of the rule file:

```
from insights.core.plugins import make_fail, rule
from mycomponents.parsers.secure_shell import SSHDConfig

ERROR_KEY = "SSHD_SECURE"


@rule(SSHDConfig)
def report(sshd_config):
    """
    1. Evaluate config file facts
    2. Evaluate version facts
    """
    if results_found:
        return make_fail(ERROR_KEY, results=the_results)
```

First we import the insights-core methods `make_fail()` for creating a response and `rule()` to decorate our rule method so that it will be invoked by insights-core with the appropriate parser information. Then we import the parsers that provide the facts we need.

```
from insights.core.plugins import make_fail, rule
from mycomponents.parsers.secure_shell import SSHDConfig
```

Next we define a unique error key string, `ERROR_KEY` that will be collected by insights-core when our rule is executed, and provided in the results for all rules. This string must be unique among all of your rules, or the last rule to execute will overwrite any results from other rules with the same key.

```
4 ERROR_KEY = "SSHD_SECURE"
```

The `@rule()` decorator is used to mark the rule method that will be invoked by insights-core. Arguments to `@rule()` consist of the parser and combiner objects that are necessary for rule processing. Each object may be either *required*, *at least one* from a list, or *optional*. All *required* objects must be available or the rule will not be called. One or more objects from the *at least one* list must be available or the rule will not be called. Zero or more objects can be available from the *optional* list.

In the `rule` decorator required objects are listed first, next are the "at least one" as a `list` argument, and finally the optional object as a `list` using the keyword `optional`. For example if the a rule has the following input requirements:

Criteria	`@rule` Decorator Arguments
Requires	`SSHDConfig, InstalledRpms`
At Least One	`[ChkConfig, UnitFiles]`
Optional	`optional=[IPTables, IpAddr]`

The decorator for the rule and the rule signature will look like this:

```
@rule(SSHDConfig, InstalledRpms, [ChkConfig, UnitFiles], optional=[IPTables, IpAddr])
def report(sshd_config, installed_rpms, chk_config, unit_files, ip_tables, ip_addr):
    # sshd_config and installed_rpms will always be present
    # at least one of chk_config and unit_files will be present
    # ip_tables and ip_addr will be present if data is available
    # arguments will be None if data is not available
```

Currently our rule requires one parser `SSHDConfig`. We will add a requirement to obtain facts about installed RPMs in the final code.

```
7 @rule(SSHDConfig)
```

The name of our rule method is `report`, but the name may be any valid method name. The purpose of the method is to evaluate the parser facts stored in the parser object `sshd_config`. If any results are found in the evaluation then a response is created with the `ERROR_KEY` and any data that you want to be associated with the results are included in the response. This data can be viewed in the results made available to a customer in the Red Hat Insights web interface. You may use zero or more named arguments to provide the data to `make_fail`. You should use meaningful argument names as it helps in understanding of the results.

```
8  def report(sshd_config):
9      """
10     1. Evaluate config file facts
11     2. Evaluate version facts
12     """
13     if results_found:
14         return make_fail(ERROR_KEY, results=the_results)
```

In order to perform the evaluation we need the facts for `sshd_config` and for the OpenSSH version. The `SSHDConfig` parser we developed will provide the facts for `sshd_config` and we can use another parser, `InstalledRpms` to help us determine facts about installed software.

Here is our updated rule with check for the configuration options and the software version:

```
from insights.core.plugins import make_fail, rule
from mycomponents.parsers.secure_shell import SSHDConfig
from insights.parsers.installed_rpms import InstalledRpms

ERROR_KEY = "SSHD_SECURE"


@rule(InstalledRpms, SSHDConfig)
def report(installed_rpms, sshd_config):
    errors = {}

    auth_method = sshd_config.last('AuthenticationMethods')
    if auth_method:
        if auth_method.lower() != 'publickey':
            errors['AuthenticationMethods'] = auth_method
    else:
        errors['AuthenticationMethods'] = 'default'

    log_level = sshd_config.last('LogLevel')
    if log_level:
        if log_level.lower() != 'verbose':
            errors['LogLevel'] = log_level
    else:
        errors['LogLevel'] = 'default'

    permit_root = sshd_config.last('PermitRootLogin')
    if permit_root:
        if permit_root.lower() != 'no':
            errors['PermitRootLogin'] = permit_root
    else:
        errors['PermitRootLogin'] = 'default'

    # Default Protocol is 2
    protocol = sshd_config.last('Protocol')
    if protocol:
        if protocol.lower() != '2':
            errors['Protocol'] = protocol

    if errors:
        openssh_version = installed_rpms.get_max('openssh')
        return make_fail(ERROR_KEY, errors=errors, openssh=openssh_version.package)
```

This rules code implements the checking of the four configuration values `AuthenticationMethods`, `LogLevel`, `PermitRootLogin`, and `Protocol`, and returns any errors found using `make_fail` in the return. Also, if errors are found, the `InstalledRpms` parser facts are queried to determine the version of *OpenSSH* installed and that value is also returned. If no values are found then an implicit `None` is returned.

Now that we have the logic to check all of the rule conditions it is possible to refactor the rule to make the condition checks more obvious. This is sometimes helpful in testing your rule as will be discussed below. Here is the refactored rule:

```
from insights.core.plugins import make_fail, rule
from insights.parsers.secure_shell import SSHDConfig
from insights.parsers.installed_rpms import InstalledRpms

ERROR_KEY = "SSHD_SECURE"

```

(continues on next page)

(continued from previous page)

```

def check_auth_method(sshd_config, errors):
    auth_method = sshd_config.last('AuthenticationMethods')
    if auth_method:
        if auth_method.lower() != 'publickey':
            errors['AuthenticationMethods'] = auth_method
    else:
        errors['AuthenticationMethods'] = 'default'
    return errors


def check_log_level(sshd_config, errors):
    log_level = sshd_config.last('LogLevel')
    if log_level:
        if log_level.lower() != 'verbose':
            errors['LogLevel'] = log_level
    else:
        errors['LogLevel'] = 'default'
    return errors


def check_permit_root(sshd_config, errors):
    permit_root = sshd_config.last('PermitRootLogin')
    if permit_root:
        if permit_root.lower() != 'no':
            errors['PermitRootLogin'] = permit_root
    else:
        errors['PermitRootLogin'] = 'default'
    return errors


def check_protocol(sshd_config, errors):
    # Default Protocol is 2 if not specified
    protocol = sshd_config.last('Protocol')
    if protocol:
        if protocol.lower() != '2':
            errors['Protocol'] = protocol
    return errors


@rule(InstalledRpms, SSHDConfig)
def report(installed_rpms, sshd_config):
    errors = {}
    errors = check_auth_method(sshd_config, errors)
    errors = check_log_level(sshd_config, errors)
    errors = check_permit_root(sshd_config, errors)
    errors = check_protocol(sshd_config, errors)

    if errors:
        openssh_version = installed_rpms.get_max('openssh')
        return make_fail(ERROR_KEY, errors=errors, openssh=openssh_version.package)
```

To increase the readability of the rule output and possibly make the transition to insights content format smoother, add Jinga formatting to the sshd_secure rule. Here is the refactored code with the additional Jinga formatting:

```
from insights.core.plugins import make_fail, rule
```

(continues on next page)

(continued from previous page)

```
from insights.parsers.secure_shell import SSHDConfig
from insights.parsers.installed_rpms import InstalledRpms

ERROR_KEY = "SSHD_SECURE"

 # Jinga template displayed for make_fail results
 CONTENT =  ERROR_KEY + """
 :{
                 {% for key, value in errors.items() -%}
                     {{key}}: {{value}}
                 {% endfor -%} }
 OPEN_SSH_PACKAGE: {{openssh}}""".strip()


def check_auth_method(sshd_config, errors):
    auth_method = sshd_config.last('AuthenticationMethods')
    if auth_method:
        if auth_method.lower() != 'publickey':
            errors['AuthenticationMethods'] = auth_method
    else:
        errors['AuthenticationMethods'] = 'default'
    return errors


def check_log_level(sshd_config, errors):
    log_level = sshd_config.last('LogLevel')
    if log_level:
        if log_level.lower() != 'verbose':
            errors['LogLevel'] = log_level
    else:
        errors['LogLevel'] = 'default'
    return errors


def check_permit_root(sshd_config, errors):
    permit_root = sshd_config.last('PermitRootLogin')
    if permit_root:
        if permit_root.lower() != 'no':
            errors['PermitRootLogin'] = permit_root
    else:
        errors['PermitRootLogin'] = 'default'
    return errors


def check_protocol(sshd_config, errors):
    # Default Protocol is 2 if not specified
    protocol = sshd_config.last('Protocol')
    if protocol:
        if protocol.lower() != '2':
            errors['Protocol'] = protocol
    return errors


@rule(InstalledRpms, SSHDConfig)
def report(installed_rpms, sshd_config):
    errors = {}
    errors = check_auth_method(sshd_config, errors)
```

(continues on next page)

(continued from previous page)

```
    errors = check_log_level(sshd_config, errors)
    errors = check_permit_root(sshd_config, errors)
    errors = check_protocol(sshd_config, errors)

    if errors:
        openssh_version = installed_rpms.get_max('openssh')
        return make_fail(ERROR_KEY, errors=errors, openssh=openssh_version.package)
```

4.2.2.2 Rule Testing

Testing is an important aspect of rule development and it helps ensure accurate rule logic. There are generally two types of testing to be performed on rules, unit and integration testing. If rule logic is divided among multiple methods then unit tests should be written to test the methods. If there is only one method then unit tests may not be necessary. Integration tests are necessary to test the rule in a simulated insights-core environment. This will be easier to understand by viewing the test code:

```
from mycomponents.rules import sshd_secure
from insights.tests import InputData, archive_provider, context_wrap
from insights.core.plugins import make_fail
from insights.specs import Specs
# The following imports are not necessary for integration tests
from mycomponents.parsers.secure_shell import SSHDConfig

OPENSSH_RPM = """
openssh-6.6.1p1-31.el7.x86_64
openssh-6.5.1p1-31.el7.x86_64
""".strip()

EXPECTED_OPENSSH = "openssh-6.6.1p1-31.el7"

GOOD_CONFIG = """
AuthenticationMethods publickey
LogLevel VERBOSE
PermitRootLogin No
# Protocol 2
""".strip()

BAD_CONFIG = """
AuthenticationMethods badkey
LogLevel normal
PermitRootLogin Yes
Protocol 1
""".strip()

DEFAULT_CONFIG = """
# All default config values
""".strip()



@archive_provider(sshd_secure.report)
def integration_tests():
    """
    InputData acts as the data source for the parsers
    so that they may execute and then be used as input
```

(continues on next page)

(continued from previous page)

```
    to the rule.  So this is essentially an end-to-end
    test of the component chain.
    """
    input_data = InputData("GOOD_CONFIG")
    input_data.add(Specs.sshd_config, GOOD_CONFIG)
    input_data.add(Specs.installed_rpms, OPENSSH_RPM)
    yield input_data, None

    input_data = InputData("BAD_CONFIG")
    input_data.add(Specs.sshd_config, BAD_CONFIG)
    input_data.add(Specs.installed_rpms, OPENSSH_RPM)
    errors = {
        'AuthenticationMethods': 'badkey',
        'LogLevel': 'normal',
        'PermitRootLogin': 'Yes',
        'Protocol': '1'
    }
    expected = make_fail(sshd_secure.ERROR_KEY,
                         errors=errors,
                         openssh=EXPECTED_OPENSSH)
    yield input_data, expected

    input_data = InputData("DEFAULT_CONFIG")
    input_data.add(Specs.sshd_config, DEFAULT_CONFIG)
    input_data.add(Specs.installed_rpms, OPENSSH_RPM)
    errors = {
        'AuthenticationMethods': 'default',
        'LogLevel': 'default',
        'PermitRootLogin': 'default'
    }
    expected = make_fail(sshd_secure.ERROR_KEY,
                         errors=errors,
                         openssh=EXPECTED_OPENSSH)
    yield input_data, expected
```

Test Data

Data utilized for all tests is defined in the test module. In this case we will use an OpenSSH RPM version that is present in RHEL 7.2, `OPENSSH_RPM` and three configuration files for `sshd_config`. `GOOD_CONFIG` has all of the values that we are looking for and should not return any error results. `BAD_CONFIG` has all bad values so it should return all error results. And `DEFAULT_CONFIG` has no values present so it should return errors for all values except `Protocol` which defaults to the correct value.

```
OPENSSH_RPM = """
openssh-6.6.1p1-31.el7.x86_64
openssh-6.5.1p1-31.el7.x86_64
""".strip()

EXPECTED_OPENSSH = "openssh-6.6.1p1-31.el7"

GOOD_CONFIG = """
AuthenticationMethods publickey
LogLevel VERBOSE
PermitRootLogin No
```

(continues on next page)

(continued from previous page)

```
# Protocol 2
""".strip()

BAD_CONFIG = """
AuthenticationMethods badkey
LogLevel normal
PermitRootLogin Yes
Protocol 1
""".strip()

DEFAULT_CONFIG = """
# All default config values
""".strip()
```

Integration Tests

Integration tests are performed within the insights-core framework. The `InputData` class is used to define the raw data that we want to be present, and the framework creates an archive file to be input to the insights-core framework so that the parsers will be invoked, and then the rules will be invoked. You need to create `InputData` objects with all information that is necessary for parsers required by your rules. If input data is not present then parsers will not be executed, and if your rule requires a missing parser it will not be executed.

To create your integration tests you must first create a method that does not begin with `test_` and decorate that method with `@archive_provider(rule_name)` having an argument that is your rule function name. Typically we name the method `integration_tests`.

```
@archive_provider(sshd_secure.report)
def integration_tests():
```

Next we create an `InputData` object and it is useful to provide a name argument to the constructor. When you execute integration tests, that name will show up in the results and make it easier to debug if you have any problems. Next you add your test inputs to the `InputData` object that will be used to create the test archive. You add the data with the `add` method and identify the source of the data using the data source spec that is associated with the parser such as `Specs.sshd_config`. Once all of the data has been added, a `yield` statement provides the input data and expected results to the `archive_provider` to run the test. In this particular test case we provided all *good* data so we did not expect any results `None`.

```
    input_data = InputData("GOOD_CONFIG")
    input_data.add(Specs.sshd_config, GOOD_CONFIG)
    input_data.add(Specs.installed-rpms, OPENSSH_RPM)
    yield input_data, None
```

Note: If your input data has a path that is significant to the interpretation of the data, such as `/etc/sysconfig/network-scripts/ifcfg-eth0` where there may be multiple `ifcfg` scripts, you'll need to add the path as well. For example:

```
input_data.add(Specs.ifcfg,
               IFCFG_ETH0,
               path="etc/sysconfig/network-scripts/ifcfg-eth0")
input_data.add(Specs.ifcfg,
               IFCFG_ETH1,
               path="etc/sysconfig/network-scripts/ifcfg-eth1")
```

In the second test case we are using *bad* input data so we have to also provide the errors that we expect our rule to return to the framework. The expected results are in the same format that we create the return value in `ssh_secure.report`.

```
    input_data = InputData(name="BAD_CONFIG")
    input_data.add(Specs.sshd_config, BAD_CONFIG)
    input_data.add(Specs.installed-rpms, OPENSSH_RPM)
    errors = {
        'AuthenticationMethods': 'badkey',
        'LogLevel': 'normal',
        'PermitRootLogin': 'Yes',
        'Protocol': '1'
    }
    expected = make_fail(sshd_secure.ERROR_KEY,
                         errors=errors,
                         openssh=EXPECTED_OPENSSH)
    yield input_data, expected
```

Running the Tests

We execute these tests by moving to the root directory of our rules project, ensuring that our virtual environment is active, and running `pytest`:

```
(env)[userone@hostone mycomponents]$ pytest -k mycomponents/rules
====================== test session starts␣
↪============================================
platform linux -- Python 3.6.6, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /home/userone/work/insights-core-tutorials, inifile: setup.cfg
plugins: cov-2.4.0
collected 15 items / 9 deselected

mycomponents/rules/tests/integration.py .....                                ␣
↪                                                                            ␣
↪                        [ 83%]
=================== 6 passed, 9 deselected in 0.30 seconds␣
↪============================
```

You may also want to run the rule using `insights-run`. This will give you a better idea of what the output would be from the rule. We execute this test by moving to the root directory (`insights-core-tutorials`), ensuring that our virtual environment is active, and running `insight-run -p rules/sshd_secure.py`:

```
(insights-core)[userone@hostone mycomponents]$ insights-run -p rules/sshd_secure.py
---------
Progress:
---------
F

--------------
Rules Executed
--------------
[FAIL] rules.sshd_secure.report
------------------------
SSHD_SECURE:
    errors : {'AuthenticationMethods': 'default',
              'LogLevel': 'default',
              'PermitRootLogin': 'default',
```

(continues on next page)

(continued from previous page)

```
             'Protocol': '1'}
    openssh: 'openssh-7.7p1-6.fc28'



----------------------
Rule Execution Summary
----------------------
Missing Deps: 0
Passed      : 0
Fingerprint : 0
Failed      : 1
Metadata    : 0
Metadata Key: 0
Exceptions  : 0
```

Note: If you have already built your parser in the `mycomponents/parsers` directory then you will see the following, otherwise you would only see tests for rules...

If any tests fail you can use the following `pytest -s -v --appdebug` options to help get additional information. If you want to limit which test run you can also use the `-k test_filter_string` option.

CHAPTER 5

Run All Tests

Now you are ready to run the tests for components you just built as well as the components in the `insights_examples` directory...

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials
(env)[userone@hostone insights-core-tutorials]$ pytest
```

This will run the tests for the components you created in `mycomponents` as well as the components provided in the `insights_examples` directory.

If you would like to run only the tests in your newly created `mycomponents` directory run the following...

```
(env)[userone@hostone ~]$ cd ~/work/insights-core-tutorials
(env)[userone@hostone insights-core-tutorials]$ pytest -k mycomponents
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search