
insights-core Documentation

Release 0.0.1

Author

Dec 06, 2022

Contents

1	Preparing Your Development Environment	3
1.1	Begin Setup	3
1.2	Manual Setup (Not required if you ran setup_env.sh)	4
1.3	Setup Complete	5
2	Tutorial - Custom Parser	7
2.1	Parser Development	7
2.1.1	Secure Shell Parser	7
2.1.1.1	Overview	7
2.1.1.2	Creating the Initial Parser Files	8
2.1.1.3	Parser Implementation	9
2.1.1.4	Parser Documentation	13
2.1.1.5	Parser Testing	15
3	Tutorial - Custom Combiner Development	19
3.1	Hostname Combiner	19
3.1.1	Overview	19
3.1.2	Creating the Initial Combiner Files	20
3.1.3	Combiner Implementation	21
3.1.3.1	Test Code	21
3.1.3.2	Combiner Code	22
3.1.4	Combiner Documentation and Testing	23
4	Tutorial - Rule Development	25
4.1	Rule Using Existing Parser and Combiner	25
4.1.1	Determine rule logic	25
4.1.2	Identify Parsers	25
4.1.3	Develop Plugin	26
4.1.4	Develop Tests	27
4.2	Rule Using Custom Parser	28
4.2.1	Overview	28
4.2.2	Secure Shell Server Rule	29
4.2.2.1	Rule Code	29
4.2.2.2	Rule Testing	34
5	Run All Tests	39

Contents:

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](#).

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:

```
1 from insights import Parser, parser
2 from insights.specs import Specs
3
4
5 @parser(Specs.sshd_config)
6 class SSHDConfig(Parser):
7
8     def parse_content(self, content):
9         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:

```
1 from mycomponents.parsers.secure_shell import SSHDConfig
2
3
4 def test_sshd_config():
5     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:

```

1  from mycomponents.parsers.secure_shell import SSHDConfig
2  from insights.tests import context_wrap
3
4  SSHD_CONFIG_INPUT = """
5  #   $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $
6
7  Port 22
8  #AddressFamily any
9  ListenAddress 10.110.0.1
10 Port 22
11 ListenAddress 10.110.1.1
12 #ListenAddress ::
13
14 # The default requires explicit activation of protocol 1
15 #Protocol 2
16 Protocol 1
17 """
18
19
20 def test_sshd_config():
21     sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
22     assert sshd_config is not None
23     assert 'Port' in sshd_config
24     assert 'PORT' in sshd_config
25     assert sshd_config['port'] == ['22', '22']
26     assert 'ListenAddress' in sshd_config
27     assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']
28     assert sshd_config['Protocol'] == ['1']
29     assert 'AddressFamily' not in sshd_config
30     ports = [l for l in sshd_config if l.keyword == 'Port']
31     assert len(ports) == 2
32     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:

```
1 from mycomponents.parsers.secure_shell import SSHDConfig
2 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:

```
4 SSHD_CONFIG_INPUT = """
5 # $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $
6
7 Port 22
8 #AddressFamily any
9 ListenAddress 10.110.0.1
10 Port 22
11 ListenAddress 10.110.1.1
12 #ListenAddress ::
13
14 # The default requires explicit activation of protocol 1
15 #Protocol 2
16 Protocol 1
17 """
```

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

```
31 def test_sshd_config():
32     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:

```
33     assert sshd_config is not None
34     assert 'Port' in sshd_config
35     assert 'PORT' in sshd_config
36     assert sshd_config['port'] == ['22', '22']
37     assert 'ListenAddress' in sshd_config
38     assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.0.1']
39     assert sshd_config['Protocol'] == ['1']
40     assert 'AddressFamily' not in sshd_config
41     ports = [l for l in sshd_config if l.keyword == 'Port']
42     assert len(ports) == 2
43     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.

```

1  from collections import namedtuple
2  from insights import Parser, parser, get_active_lines
3  from insights.core.spec_factory import SpecSet, simple_file
4  import os
5
6
7  class LocalSpecs (SpecSet):
8      """ Datasources for collection from local host """
9      conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
↵')
10
11     sshd_config = simple_file(conf_file)
12
13
14  @parser(LocalSpecs.sshd_config)
15  class SSHDConfig (Parser):
16
17     KeyValue = namedtuple('KeyValue', ['keyword', 'value', 'kw_lower'])
18
19     def parse_content(self, content):
20         self.lines = []
21         for line in get_active_lines(content):
22             kw, val = line.split(None, 1)
23             self.lines.append(self.KeyValue(kw.strip(), val.strip(), kw.lower().
↵strip()))
24         self.keywords = set([k.kw_lower for k in self.lines])
25
26     def __contains__(self, keyword):
27         return keyword.lower() in self.keywords
28
29     def __iter__(self):
30         for line in self.lines:
31             yield line
32
33     def __getitem__(self, keyword):
34         kw = keyword.lower()
35         if kw in self.keywords:
36             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.

```

1  from collections import namedtuple
2  from insights import Parser, parser, get_active_lines
3  from insights.core.spec_factory import SpecSet, simple_file
4  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.

```

6  class LocalSpecs(SpecSet):
7      """ Datasources for collection from local host """
8      conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
→')
9
10     sshd_config = simple_file(conf_file)

```

To get the `sshd_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`.

```

15  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.

```

17  def parse_content(self, content):
18      self.lines = []
19      for line in get_active_lines(content):
20          kw, val = line.split(None, 1)
21          self.lines.append(self.KeyValue(kw.strip(), val.strip(), kw.lower()
→strip()))
22      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.

```

24  def __contains__(self, keyword):
25      return keyword.lower() in self.keywords
26
27  def __iter__(self):
28      for line in self.lines:
29          yield line
30
31  def __getitem__(self, keyword):

```

(continues on next page)

(continued from previous page)

```

32     kw = keyword.lower()
33     if kw in self.keywords:
34         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.

```

1  """
2  secure_shell - Files for configuration of `ssh`
3  =====
4
5  The ``secure_shell`` module provides parsing for the ``sshd_config``
6  file. The ``SSHDConfig`` class implements the parsing and
7  provides a ``list`` of all configuration lines present in
8  the file.
9
10 Sample content from the ``/etc/sshd/sshd_config`` file is::
11
12     #           $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $
13
14     Port 22
15     #AddressFamily any
16     ListenAddress 10.110.0.1
17     Port 22
18     ListenAddress 10.110.1.1
19     #ListenAddress ::
20
21     # The default requires explicit activation of protocol 1
22     #Protocol 2
23     Protocol 1
24
25 Examples:
26     >>> 'Port' in sshd_config
27     True
28     >>> 'PORT' in sshd_config # items are stored case-insensitive
29     True
30     >>> 'AddressFamily' in sshd_config # comments are ignored
31     False
32     >>> sshd_config['port'] # All value stored by keyword in lists
33     ['22', '22']
34     >>> sshd_config['Protocol'] # Single items have one list element
35     ['1']
36     >>> [line for line in sshd_config if line.keyword == 'Port'] # can be used as
↳an iterator
37     [KeyValue(keyword='Port', value='22', kw_lower='port'), KeyValue(keyword='Port',
↳value='22', kw_lower='port')]
38     >>> sshd_config.last('ListenAddress') # Easy way of finding the current
↳configuration for a single item

```

(continues on next page)

```

39     '10.110.1.1'
40     """
41     from collections import namedtuple
42     from insights import Parser, parser, get_active_lines
43     from insights.specs import Specs
44     from insights.core.spec_factory import SpecSet, simple_file
45     import os
46
47
48     class LocalSpecs(SpecSet):
49         """ Datasources for collection from local host """
50         conf_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sshd_config
↪')
51
52         sshd_config = simple_file(conf_file)
53
54
55     @parser(LocalSpecs.sshd_config)
56     class SSHDConfig(Parser):
57         """Parsing for ``sshd_config`` file.
58
59         Attributes:
60             lines (list): List of `KeyValue` namedtuples for each line in
61                 the configuration file.
62             keywords (set): Set of keywords present in the configuration
63                 file, each keyword has been converted to lowercase.
64         """
65
66         KeyValue = namedtuple('KeyValue', ['keyword', 'value', 'kw_lower'])
67         """namedtuple: Represent name value pair as a namedtuple with case ."""
68
69         def parse_content(self, content):
70             self.lines = []
71             for line in get_active_lines(content):
72                 kw, val = (w.strip() for w in line.split(None, 1))
73                 self.lines.append(self.KeyValue(kw, val, kw.lower()))
74             self.keywords = set([k.kw_lower for k in self.lines])
75
76         def __contains__(self, keyword):
77             return keyword.lower() in self.keywords
78
79         def __iter__(self):
80             for line in self.lines:
81                 yield line
82
83         def __getitem__(self, keyword):
84             kw = keyword.lower()
85             if kw in self.keywords:
86                 return [kv.value for kv in self.lines if kv.kw_lower == kw]
87
88         def last(self, keyword):
89             """str: Returns the value of the last keyword found in config."""
90             entries = self.__getitem__(keyword)
91             if entries:
92                 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:

```

1 from mycomponents.parsers.secure_shell import SSHDConfig
2 from insights.parsers import secure_shell
3 from insights.tests import context_wrap
4 import doctest
5
6 SSHD_CONFIG_INPUT = """
7 # $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $
8
9 Port 22
10 #AddressFamily any
11 ListenAddress 10.110.0.1
12 Port 22
13 ListenAddress 10.110.1.1
14 #ListenAddress ::
15
16 # The default requires explicit activation of protocol 1
17 #Protocol 2
18 Protocol 1
19 """
20
21 def test_sshd_config():
22     sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
23     assert sshd_config is not None
24     assert 'Port' in sshd_config
25     assert 'PORT' in sshd_config
26     assert sshd_config['port'] == ['22', '22']
27     assert 'ListenAddress' in sshd_config
28     assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']
29     assert sshd_config['Protocol'] == ['1']
30     assert 'AddressFamily' not in sshd_config
31     ports = [l for l in sshd_config if l.keyword == 'Port']
32     assert len(ports) == 2
33     assert ports[0].value == '22'

```

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

```

37 def test_sshd_documentation():
38     """
39     Here we test the examples in the documentation automatically using
40     doctest. We set up an environment which is similar to what a
41     rule writer might see - a 'sshd_config' variable that has been
42     passed in as a parameter to the rule declaration. This saves doing
43     this setup in the example code.
44     """
45     env = {
46         'sshd_config': SSHDConfig(context_wrap(SSHD_CONFIG_INPUT)),
47     }
48     failed, total = doctest.testmod(secure_shell, globs=env)
49     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':

```

52 SSHD_DOCS_EXAMPLE = '''
53 Port 22
54 Port 22
55 '''
56
57 def test_sshd_corner_cases():
58     """
59     Here we test any corner cases for behavior we expect to deal with
60     in the parser but doesn't make a good example.
61     """
62     config = SSHDConfig(context_wrap(SSHD_DOCS_EXAMPLE))
63     assert config.last('AddressFamily') is None
64     assert config['AddressFamily'] is None
65     ports = [l for l in config if l.keyword == 'Port']
66     assert len(ports) == 2
67     assert ports[0].value == '22'

```

The final version of our test now looks like this:

```

1  from mycomponets.parsers.secure_shell import SSHDConfig
2  from insights.parsers import secure_shell
3  from insights.tests import context_wrap
4  import doctest
5
6  SSHD_CONFIG_INPUT = """
7  # $OpenBSD: sshd_config,v 1.93 2014/01/10 05:59:19 djm Exp $
8
9  Port 22
10 #AddressFamily any
11 ListenAddress 10.110.0.1
12 Port 22
13 ListenAddress 10.110.1.1
14 #ListenAddress ::
15
16 # The default requires explicit activation of protocol 1
17 #Protocol 2
18 Protocol 1
19 """
20
21 def test_sshd_config():
22     sshd_config = SSHDConfig(context_wrap(SSHD_CONFIG_INPUT))
23     assert sshd_config is not None
24     assert 'Port' in sshd_config
25     assert 'PORT' in sshd_config
26     assert sshd_config['port'] == ['22', '22']
27     assert 'ListenAddress' in sshd_config
28     assert sshd_config['ListenAddress'] == ['10.110.0.1', '10.110.1.1']

```

(continues on next page)

(continued from previous page)

```

29  assert sshd_config['Protocol'] == ['1']
30  assert 'AddressFamily' not in sshd_config
31  ports = [l for l in sshd_config if l.keyword == 'Port']
32  assert len(ports) == 2
33  assert ports[0].value == '22'
34
35
36  def test_sshd_documentation():
37      """
38      Here we test the examples in the documentation automatically using
39      doctest. We set up an environment which is similar to what a
40      rule writer might see - a 'sshd_config' variable that has been
41      passed in as a parameter to the rule declaration. This saves doing
42      this setup in the example code.
43      """
44      env = {
45          'sshd_config': SSHDConfig(context_wrap(SSHD_CONFIG_INPUT)),
46      }
47      failed, total = doctest.testmod(secure_shell, globs=env)
48      assert failed == 0
49
50
51  SSHD_DOCS_EXAMPLE = '''
52  Port 22
53  Port 22
54  '''
55
56
57  def test_sshd_corner_cases():
58      """
59      Here we test any corner cases for behavior we expect to deal with
60      in the parser but doesn't make a good example.
61      """
62      config = SSHDConfig(context_wrap(SSHD_DOCS_EXAMPLE))
63      assert config.last('AddressFamily') is None
64      assert config['AddressFamily'] is None
65      ports = [l for l in config if l.keyword == 'Port']
66      assert len(ports) == 2
67      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.

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:

```
1 from insights.core.plugins import combiner
2 from insights.parsers.hostname import Hostname
3 from insights.parsers.uname import Uname
4
5
6 @combiner([Hostname, Uname])
7 class HostnameUH(object):
8
9     def __init__(self, hostname, uname):
10         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 will 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:

```
1 from mycomponents.combiners.hostname_uh import HostnameUH
2
3
4 def test_hostname_uh():
5     pass
```

Once you have created and saved both of these files, you can 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.

```

1 from mycomponents.combiners.hostname_uh import HostnameUH
2 from insights.parsers.hostname import Hostname
3 from insights.parsers.uname import Uname
4 from insights.tests import context_wrap
5
6 HOSTNAME = "hostone_h.example.com"
7 UNAME = "Linux hostone_u.example.com 3.10.0-693.21.1.el7.x86_64 #1 SMP Fri Feb 23_
8 ↪18:54:16 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux"
9
10 def test_hostname_uh():
11     hostname = Hostname(context_wrap(HOSTNAME))
12     uname = Uname(context_wrap(UNAME))
13
14     hostname_uh = HostnameUH(hostname, None)
15     assert hostname_uh.hostname == HOSTNAME
16
17     hostname_uh = HostnameUH(None, uname)
18     assert hostname_uh.hostname == "hostone_u.example.com"
19
20     hostname_uh = HostnameUH(hostname, uname)
21     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:

```

1 from insights.combiners.hostname_uh import HostnameUH
2 from insights.parsers.hostname import Hostname
3 from insights.parsers.uname import Uname
4 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:

```
6 HOSTNAME = "hostone_h.example.com"
7 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:

```
10 def test_hostname_uh():
11     hostname = Hostname(context_wrap(HOSTNAME))
12     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:

```
14 hostname_uh = HostnameUH(hostname, None)
15 assert hostname_uh.hostname == HOSTNAME
16
17 hostname_uh = HostnameUH(None, uname)
18 assert hostname_uh.hostname == "hostone_u.example.com"
19
20 hostname_uh = HostnameUH(hostname, uname)
21 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.

```
1 from insights.core.plugins import combiner
2 from insights.parsers.hostname import Hostname
3 from insights.parsers.uname import Uname
4
5
6 @combiner([Hostname, Uname])
7 class HostnameUH(object):
8
9     def __init__(self, hostname, uname):
10         if hostname:
```

(continues on next page)

(continued from previous page)

```

11         self.hostname = hostname.fqdn
12     else:
13         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](#)

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.

```
1 from insights.parsers.redhat_release import RedhatRelease
2 from insights import rule, make_fail, make_pass
3
4 @rule(RedhatRelease)
5 def report(rhrel):
6     pass
```

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

```
1 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.

```
2 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.

```
6 ERROR_KEY_IS_FEDORA = "IS_FEDORA"
7
8 CONTENT = {
9     ERROR_KEY_IS_FEDORA: "This machine ({{hostname}}) runs ({{product}}).",
10 }
```

Here we define the Jinja2 template for message to be displayed for either response tag

```
12 @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

```
12 @rule(RedhatRelease, content=CONTENT)
13 def report(rhrel):
14     """Fires if the machine is running Fedora."""
15
16     if "Fedora" in rel.product:
17         return make_pass(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,
↵product=rel.product)
18     else:
19         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.

```

16     if "Fedora" in rhrel.product:
17         return make_pass(ERROR_KEY_IS_FEDORA, hostname=hostname.hostname,
↳product=rel.product)
18     else:
19         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.

```

1  from .. import is_fedora
2  from insights.specs import Specs
3  from insights.tests import InputData, archive_provider
4  from insights.core.plugins import make_fail, make_pass
5
6
7  @archive_provider(is_fedora.report)
8  def integration_test():
9      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.

```

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

```

(continues on next page)

(continued from previous page)

```

18 input_data.add(Specs.hostname, TEST_HOSTNAME)
19 expected = make_pass("IS_FEDORA", hostname=TEST_HOSTNAME, product="Fedora")
20
21 yield input_data, expected
22
23 input_data = InputData("test_rhel")
24 input_data.add(Specs.redhat_release, RHEL)
25 input_data.add(Specs.hostname, TEST_HOSTNAME)
26 expected = make_fail("IS_FEDORA", hostname=TEST_HOSTNAME, product="Red Hat_
↳Enterprise Linux Server")
27
28 yield input_data, expected

```

Now for the entire test:

```

1 from .. import is_fedora
2 from insights.specs import Specs
3 from insights.tests import InputData, archive_provider
4 from insights.core.plugins import make_fail, make_pass
5
6 FEDORA = "Fedora release 28 (Twenty Eight)"
7 RHEL = "Red Hat Enterprise Linux Server release 7.4 (Maipo)"
8 TEST_HOSTNAME = "testhost.someplace.com"
9
10
11 @archive_provider(is_fedora.report)
12 def integration_test():
13
14     input_data = InputData("test_fedora")
15     input_data.add(Specs.redhat_release, FEDORA)
16     input_data.add(Specs.hostname, TEST_HOSTNAME)
17     expected = make_pass("IS_FEDORA", hostname=TEST_HOSTNAME, product="Fedora")
18
19
20     yield input_data, expected
21
22     input_data = InputData("test_rhel")
23     input_data.add(Specs.redhat_release, RHEL)
24     input_data.add(Specs.hostname, TEST_HOSTNAME)
25     expected = make_fail("IS_FEDORA", hostname=TEST_HOSTNAME, product="Red Hat_
↳Enterprise Linux Server")
26
27     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:

```
1 from insights.core.plugins import make_fail, rule
2 from mycomponents.parsers.secure_shell import SSHDConfig
3
4 ERROR_KEY = "SSHD_SECURE"
5
6
7 @rule(SSHDConfig)
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)
```

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.

```
1 from insights.core.plugins import make_fail, rule
2 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	<code>SSHDConfig, InstalledRpms</code>
At Least One	<code>[ChkConfig, UnitFiles]</code>
Optional	<code>optional=[IPTables, IpAddr]</code>

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:

```

1 from insights.core.plugins import make_fail, rule
2 from mycomponents.parsers.secure_shell import SSHDConfig
3 from insights.parsers.installed_rpms import InstalledRpms
4
5 ERROR_KEY = "SSHD_SECURE"
6
7
8 @rule(InstalledRpms, SSHDConfig)
9 def report(installed_rpms, sshd_config):
10     errors = {}
11
12     auth_method = sshd_config.last('AuthenticationMethods')
13     if auth_method:
14         if auth_method.lower() != 'publickey':
15             errors['AuthenticationMethods'] = auth_method
16     else:
17         errors['AuthenticationMethods'] = 'default'
18
19     log_level = sshd_config.last('LogLevel')
20     if log_level:
21         if log_level.lower() != 'verbose':
22             errors['LogLevel'] = log_level
23     else:
24         errors['LogLevel'] = 'default'
25
26     permit_root = sshd_config.last('PermitRootLogin')
27     if permit_root:
28         if permit_root.lower() != 'no':
29             errors['PermitRootLogin'] = permit_root
30     else:
31         errors['PermitRootLogin'] = 'default'
32
33     # Default Protocol is 2
34     protocol = sshd_config.last('Protocol')
35     if protocol:
36         if protocol.lower() != '2':
37             errors['Protocol'] = protocol
38
39     if errors:
40         openssh_version = installed_rpms.get_max('openssh')
41         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:

```

1 from insights.core.plugins import make_fail, rule
2 from insights.parsers.secure_shell import SSHDConfig
3 from insights.parsers.installed_rpms import InstalledRpms
4
5 ERROR_KEY = "SSHD_SECURE"
6

```

(continues on next page)

(continued from previous page)

```

7
8 def check_auth_method(sshd_config, errors):
9     auth_method = sshd_config.last('AuthenticationMethods')
10    if auth_method:
11        if auth_method.lower() != 'publickey':
12            errors['AuthenticationMethods'] = auth_method
13    else:
14        errors['AuthenticationMethods'] = 'default'
15    return errors
16
17
18 def check_log_level(sshd_config, errors):
19     log_level = sshd_config.last('LogLevel')
20    if log_level:
21        if log_level.lower() != 'verbose':
22            errors['LogLevel'] = log_level
23    else:
24        errors['LogLevel'] = 'default'
25    return errors
26
27
28 def check_permit_root(sshd_config, errors):
29     permit_root = sshd_config.last('PermitRootLogin')
30    if permit_root:
31        if permit_root.lower() != 'no':
32            errors['PermitRootLogin'] = permit_root
33    else:
34        errors['PermitRootLogin'] = 'default'
35    return errors
36
37
38 def check_protocol(sshd_config, errors):
39     # Default Protocol is 2 if not specified
40     protocol = sshd_config.last('Protocol')
41    if protocol:
42        if protocol.lower() != '2':
43            errors['Protocol'] = protocol
44    return errors
45
46
47 @rule(InstalledRpms, SSHDConfig)
48 def report(installed_rpms, sshd_config):
49     errors = {}
50     errors = check_auth_method(sshd_config, errors)
51     errors = check_log_level(sshd_config, errors)
52     errors = check_permit_root(sshd_config, errors)
53     errors = check_protocol(sshd_config, errors)
54
55     if errors:
56         openssh_version = installed_rpms.get_max('openssh')
57         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 Jinja2 formatting to the `sshd_secure` rule. Here is the refactored code with the additional Jinja2 formatting:

```

1 from insights.core.plugins import make_fail, rule

```

(continues on next page)

(continued from previous page)

```

2 from insights.parsers.secure_shell import SSHDConfig
3 from insights.parsers.installed_rpms import InstalledRpms
4
5 ERROR_KEY = "SSHD_SECURE"
6
7 # Jinja2 template displayed for make_response results
8 CONTENT = ERROR_KEY + """
9 :{
10     {% for key, value in errors.items() -%}
11         {{key}}: {{value}}
12     {% endfor -%} }
13 OPEN_SSH_PACKAGE: {{openssh}}""".strip()
14
15
16 def check_auth_method(sshd_config, errors):
17     auth_method = sshd_config.last('AuthenticationMethods')
18     if auth_method:
19         if auth_method.lower() != 'publickey':
20             errors['AuthenticationMethods'] = auth_method
21     else:
22         errors['AuthenticationMethods'] = 'default'
23     return errors
24
25
26 def check_log_level(sshd_config, errors):
27     log_level = sshd_config.last('LogLevel')
28     if log_level:
29         if log_level.lower() != 'verbose':
30             errors['LogLevel'] = log_level
31     else:
32         errors['LogLevel'] = 'default'
33     return errors
34
35
36 def check_permit_root(sshd_config, errors):
37     permit_root = sshd_config.last('PermitRootLogin')
38     if permit_root:
39         if permit_root.lower() != 'no':
40             errors['PermitRootLogin'] = permit_root
41     else:
42         errors['PermitRootLogin'] = 'default'
43     return errors
44
45
46 def check_protocol(sshd_config, errors):
47     # Default Protocol is 2 if not specified
48     protocol = sshd_config.last('Protocol')
49     if protocol:
50         if protocol.lower() != '2':
51             errors['Protocol'] = protocol
52     return errors
53
54
55 @rule(InstalledRpms, SSHDConfig)
56 def report(installed_rpms, sshd_config):
57     errors = {}
58     errors = check_auth_method(sshd_config, errors)

```

(continues on next page)

(continued from previous page)

```

59     errors = check_log_level(sshd_config, errors)
60     errors = check_permit_root(sshd_config, errors)
61     errors = check_protocol(sshd_config, errors)
62
63     if errors:
64         openssh_version = installed_rpms.get_max('openssh')
65         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:

```

1  from mycomponents.rules import sshd_secure
2  from insights.tests import InputData, archive_provider, context_wrap
3  from insights.core.plugins import make_fail
4  from insights.specs import Specs
5  # The following imports are not necessary for integration tests
6  from mycomponents.parsers.secure_shell import SSHDConfig
7
8  OPENSSSH_RPM = """
9  openssh-6.6.1p1-31.el7.x86_64
10 openssh-6.5.1p1-31.el7.x86_64
11 """.strip()
12
13 EXPECTED_OPENSSSH = "openssh-6.6.1p1-31.el7"
14
15 GOOD_CONFIG = """
16 AuthenticationMethods publickey
17 LogLevel VERBOSE
18 PermitRootLogin No
19 # Protocol 2
20 """.strip()
21
22 BAD_CONFIG = """
23 AuthenticationMethods badkey
24 LogLevel normal
25 PermitRootLogin Yes
26 Protocol 1
27 """.strip()
28
29 DEFAULT_CONFIG = """
30 # All default config values
31 """.strip()
32
33
34
35 @archive_provider(sshd_secure.report)
36 def integration_tests():
37     """
38     InputData acts as the data source for the parsers
39     so that they may execute and then be used as input

```

(continues on next page)

(continued from previous page)

```

40  to the rule. So this is essentially an end-to-end
41  test of the component chain.
42  """
43  input_data = InputData("GOOD_CONFIG")
44  input_data.add(Specs.sshd_config, GOOD_CONFIG)
45  input_data.add(Specs.installed_rpms, OPENSSSH_RPM)
46  yield input_data, None
47
48  input_data = InputData("BAD_CONFIG")
49  input_data.add(Specs.sshd_config, BAD_CONFIG)
50  input_data.add(Specs.installed_rpms, OPENSSSH_RPM)
51  errors = {
52      'AuthenticationMethods': 'badkey',
53      'LogLevel': 'normal',
54      'PermitRootLogin': 'Yes',
55      'Protocol': '1'
56  }
57  expected = make_fail(sshd_secure.ERROR_KEY,
58                      errors=errors,
59                      openssh=EXPECTED_OPENSSSH)
60  yield input_data, expected
61
62  input_data = InputData("DEFAULT_CONFIG")
63  input_data.add(Specs.sshd_config, DEFAULT_CONFIG)
64  input_data.add(Specs.installed_rpms, OPENSSSH_RPM)
65  errors = {
66      'AuthenticationMethods': 'default',
67      'LogLevel': 'default',
68      'PermitRootLogin': 'default'
69  }
70  expected = make_fail(sshd_secure.ERROR_KEY,
71                      errors=errors,
72                      openssh=EXPECTED_OPENSSSH)
73  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, OPENSSSH_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.

```

8  OPENSSSH_RPM = """
9  openssh-6.6.1p1-31.el7.x86_64
10 openssh-6.5.1p1-31.el7.x86_64
11 """.strip()
12
13 EXPECTED_OPENSSSH = "openssh-6.6.1p1-31.el7"
14
15 GOOD_CONFIG = """
16 AuthenticationMethods publickey
17 LogLevel VERBOSE
18 PermitRootLogin No

```

(continues on next page)

(continued from previous page)

```

19 # Protocol 2
20 """.strip()
21
22 BAD_CONFIG = """
23 AuthenticationMethods badkey
24 LogLevel normal
25 PermitRootLogin Yes
26 Protocol 1
27 """.strip()
28
29 DEFAULT_CONFIG = """
30 # All default config values
31 """.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`.

```

57 @archive_provider(sshd_secure.report)
58 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`.

```

59     input_data = InputData("GOOD_CONFIG")
60     input_data.add(Specs.sshd_config, GOOD_CONFIG)
61     input_data.add(Specs.installed_rpms, OPENSSSH_RPM)
62     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`.

```

64     input_data = InputData(name="BAD_CONFIG")
65     input_data.add(Specs.sshd_config, BAD_CONFIG)
66     input_data.add(Specs.installed-rpms, OPENSSSH_RPM)
67     errors = {
68         'AuthenticationMethods': 'badkey',
69         'LogLevel': 'normal',
70         'PermitRootLogin': 'Yes',
71         'Protocol': '1'
72     }
73     expected = make_fail(sshd_secure.ERROR_KEY,
74                         errors=errors,
75                         openssh=EXPECTED_OPENSSSH)
76     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 `insights-run -p rules/ssh_secure.py`:

```

(insights-core)[userone@hostone mycomponents]$ insights-run -p rules/ssh_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`