pyFG Documentation

Release 14

David Barroso

Aug 11, 2017

Contents

This library does not pretend to hide or abstract FortiOS configuration. Its sole purpose is to give you a a programmatic way to read it and modify it. To be able to use you will still need to understand how the CLI works.

There are two main libraries:

- **`FortiOS` – This is the main library you will use. It allows you to connect to a device,** read its configuration, it provides an interface to the running and the candidate config and provides diff and commit operations amongst others.
- **`FortiConfig` – This library represents a block of configuration. You will usually** deal with this just to create new objects (like a new firewall policy) or just via the attribures running_config and candidate_config in a `FortiOS` object.

CHAPTER 1

Tutorials

First Steps

Connecting to the device

Let's start connecting to the vdom `test_vdom` for a particular device:

```
>>> from pyFG import FortiOS
>>> d = FortiOS('192.168.76.50', vdom='test_vdom')
>>> d.open()
```

Loading configuration

Now you can easily load a block of configuration and do some read operations:

```
>>> d.load_config('router bgp')
```

We can verify easily that we got the configuration doing the following:

```
>>> print d.running_config.to_text()
    config router bgp
        config redistribute isis
        end
        config redistribute6 connected
        end
        config redistribute6 isis
        end
        config redistribute static
        end
        config redistribute6 rip
        end
        config redistribute connected
```

```
        end
        config redistribute ospf
        end
        config redistribute6 static
        end
        config neighbor
            edit 172.20.213.23
              set remote-as 65555
              set route-map-in "test4"
            next
            edit 2.2.2.2
              set remote-as 123
              set shutdown enable
            next
        end
        config redistribute rip
        end
        config redistribute6 ospf
        end
    end
>>>
```

Or we can get the AS for each neighbor in a programmatic way:

```
>>> for neigh, param in d.running_config['router bgp']['neighbor'].iterblocks():
...     print neigh, param.get_param('remote-as')
...
172.20.213.23 65555
2.2.2.2 123
>>>
```

We can also iterate over all the parameters for a specific block. Let's get all the parameters for the neighbor 2.2.2.2:

```
>>> for param, value in d.running_config['router bgp']['neighbor']['2.2.2.2'].
↪iterparams():
...     print param, value
...
remote-as 123
shutdown enable
```

Adding a new sub block of configuration

Now, let's add a new bgp neighbor. First we have to create the configuration block and set the parameters:

```
>>> from pyFG import FortiConfig
>>> nn = FortiConfig(config_type='edit', name='3.3.3.3')
>>> nn.set_param('remote-as', 12346)
>>> nn.set_param('shutdown', 'enable')
```

Now we assign it to the candidate configuration and print it to see it looks like we want:

```
>>> d.candidate_config['router bgp']['neighbor']['3.3.3.3'] = nn
>>> print d.candidate_config.to_text()
    config router bgp
        config redistribute isis
        end
```

```
        config redistribute6 connected
        end
        config redistribute6 isis
        end
        config redistribute static
        end
        config redistribute6 rip
        end
        config redistribute connected
        end
        config redistribute ospf
        end
        config redistribute6 static
        end
        config neighbor
            edit 172.20.213.23
              set remote-as 65555
              set route-map-in "test4"
            next
            edit 2.2.2.2
              set remote-as 123
              set shutdown enable
            next
            edit 3.3.3.3
              set remote-as 12346
              set shutdown enable
            next
        end
        config redistribute rip
        end
        config redistribute6 ospf
        end
    end
>>>
```

Deleting a sub_block of configuration

Now, let's delete neighbor 2.2.2.2:

```
>>> d.candidate_config['router bgp']['neighbor'].del_block('2.2.2.2')
```

Checking configuration changes

After all this changes let's see what has changed:

```
>>> print d.compare_configuration()
conf vdom
  edit test_vdom
    config router bgp
        config neighbor
            delete 2.2.2.2
            edit 3.3.3.3
              set remote-as 12346
              set shutdown enable
            next
```

```
        end
    end
end
>>>
```

As you can see, that method returns all the necessary commands to reach the candidate configuration from the running configuration.

Setting a parameter

Let's set a route-map outbound to neighbor 172.20.213.23:

```
>>> d.candidate_config['router bgp']['neighbor']['172.20.213.23'].set_param('route-
↪map-out', 'non-existant-routemap')
```

Committing changes

Now that we have done several changes let's commit the changes:

```
>>> d.commit()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pyFG/fortios.py", line 242, in commit
    self._commit(config_text, force)
  File "pyFG/fortios.py", line 297, in _commit
    raise exceptions.FailedCommit(wrong_commands)
pyFG.exceptions.FailedCommit: [('-3', 'set route-map-out non-existant-routemap')]
>>>
```

The route map we tried to assign does not exist so the commit failed returning a FailedCommit exception. By default, if one single command fails during the commit operation the entire commit will be rolled back. At this point you have three options:

1. Create the route map
2. Delete that parameter or assign an existing route-map
3. Force the changes.

We are going to try forcing the changes:

```
>>> d.commit(force=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pyFG/fortios.py", line 242, in commit
    self._commit(config_text, force)
  File "pyFG/fortios.py", line 299, in _commit
    raise exceptions.ForcedCommit(wrong_commands)
pyFG.exceptions.ForcedCommit: [('-3', 'set route-map-out non-existant-routemap')]
>>> print d.compare_configuration()
conf vdom
  edit test_vdom
    config router bgp
        config neighbor
            edit 172.20.213.23
              set route-map-out non-existant-routemap
```

```
            next
        end
    end
end
>>>
```

We still got an exception although this time a different one; ForcedCommit. As you can see from the compare_configuration method the rest of the changes went through.

Rolling back changes

Now, let's assume we regret the changes we just did and we want to rollback:

```
>>> d.rollback()
>>> print d.running_config.to_text()
    config router bgp
        config redistribute isis
        end
        config redistribute6 connected
        end
        config redistribute6 isis
        end
        config redistribute static
        end
        config redistribute6 rip
        end
        config redistribute connected
        end
        config redistribute ospf
        end
        config redistribute6 static
        end
        config neighbor
            edit 172.20.213.23
              set remote-as 65555
              set route-map-in "test4"
            next
            edit 2.2.2.2
              set remote-as 123
              set shutdown enable
            next
        end
        config redistribute rip
        end
        config redistribute6 ospf
        end
    end
>>>
```

Voilà, we are back to our original configuration. BGP neighbor 3.3.3.3 is gone, 2.2.2.2 is back and that broken parameter map is not in there anymore.

Closing the session

Finally we close the ssh session:

```
>>> d.close()
```

Loading configuration from a file

The following example is very interesting if you plan to manage your devices using a configuration management system based in templates like ansible. Let's assume we have a configuration file that we have generated somehow with the following content:

```
config router bgp
        config neighbor
            edit "172.20.213.32"
                set remote-as 333
                set route-map-out "test4"
            next
        end
        config redistribute "connected"
        end
        config redistribute "rip"
        end
        config redistribute "ospf"
        end
        config redistribute "static"
        end
        config redistribute "isis"
        end
        config redistribute6 "connected"
        end
        config redistribute6 "rip"
        end
        config redistribute6 "ospf"
        end
        config redistribute6 "static"
        end
        config redistribute6 "isis"
        end
end
```

We want to load that configuration into a device, replacing its current configuration. First we have to connect to the device and load the running configuration we want to replace:

```
>>> from pyFG import FortiOS
>>> d = FortiOS('192.168.76.50', vdom='test_vdom')
>>> d.open()
>>> d.load_configuration('router bgp', empty_candidate=True)
```

The parameter `empty_candidate` will load only the running config. Now, we load the configuration file into the candidate config:

```
>>> with open ("bgp_config.txt", "r") as my_file:
...     data=my_file.read()
...
>>> d.load_configuration(config_text=data, in_candidate=True)
>>> print d.candidate_config.to_text()
config router bgp
    config redistribute isis
```

```
    end
    config redistribute6 connected
    end
    config redistribute6 isis
    end
    config redistribute static
    end
    config redistribute6 rip
    end
    config redistribute connected
    end
    config redistribute ospf
    end
    config redistribute6 static
    end
    config neighbor
        edit 172.20.213.32
          set remote-as 333
          set route-map-out "test4"
        next
    end
    config redistribute rip
    end
    config redistribute6 ospf
    end
end
```

Now you can check the differences like this:

```
>>> print d.compare_configuration()
conf vdom
  edit test_vdom
    config router bgp
        config neighbor
            delete 172.20.213.23
            delete 2.2.2.2
            edit 172.20.213.32
              set remote-as 333
              set route-map-out "test4"
            next
        end
    end
end
```

And commit the changes:

```
>>> d.commit()
>>> print d.compare_configuration()
>>>
>>> d.close()
```

A final compare_configuration returning an empty string will prove us that our changes were applied correctly.

CHAPTER 2

Options

SSH Config

You can now store some of the access details for your FortiGate appliacnes inside the ~/.ssh/config of the user. Currently supported:

- Username
- Hostname
- Proxy Command
- IdentityFile

Example 1

Simple "alias" whereas you would use the 'hostname' "fortigate" inside your scripts instead of the IP address. Handy if you do not have a DNS server:

```
Host fortigate
  Hostname 192.168.1.1
  User admin
```

Example 2

A bit more complex for when you need go via a SSH proxy server to reach the appliance:

```
Host fortigate
  Hostname 192.168.1.1
  User admin
  ProxyCommand ssh user@10.10.10.1 nc %h %p
```

CHAPTER 3

Classes

FortiOS

FortiConfig