



Puppet Firewall Module and Landb  
Integration

Supervisor: Steve Traylen  
Student: Andronidis Anastasios

Summer 2012

## **Abstract**

During my stay at CERN as an intern, I had to complete two tasks that are related to Puppet project.

The first task was to debug and add a new feature to a Puppet plugin called Puppetlabs Firewall. The new feature will make the plugin able to ignore some firewall chains so Nova OpenStack could work the right way.

The second task was to create a ruby library to communicate with landb through a SOAP protocol and then, utilize this library to make a Puppet function. After that, Puppet templates could use this function to retrieve information from landb.

## Introduction To Puppet

Puppet is a tool designed to manage the configuration of [Unix-like](#) and [Microsoft Windows](#) systems declaratively. The user describes system resources and their state, either using Puppet's declarative language or a Ruby DSL ([domain-specific language](#)). This information is stored in files called "Puppet manifests". Puppet discovers the system information via a utility called `Facter`, and compiles the Puppet manifests into a system-specific catalog containing resources and resource dependency, which are applied against the target systems. Any actions taken by Puppet are then reported.

Among the most powerful features of Puppet are its flexibility and extensibility. In addition to the existing facts, resource types, providers, and functions, you can quickly and easily add custom code specific to your environment or to meet a particular need.

## The Puppetlabs Firewall

Such a case is Puppetlabs Firewall module, which provides the resource 'firewall' which provides the capability to manage firewall rules within puppet.

At the moment this report is being written, Puppetlabs Firewall module supports:

- iptables
- ip6tables

as firewall mechanisms to manage rules. And

- iptables
- ip6tables
- ebtables

as firewall mechanisms to manage chains.

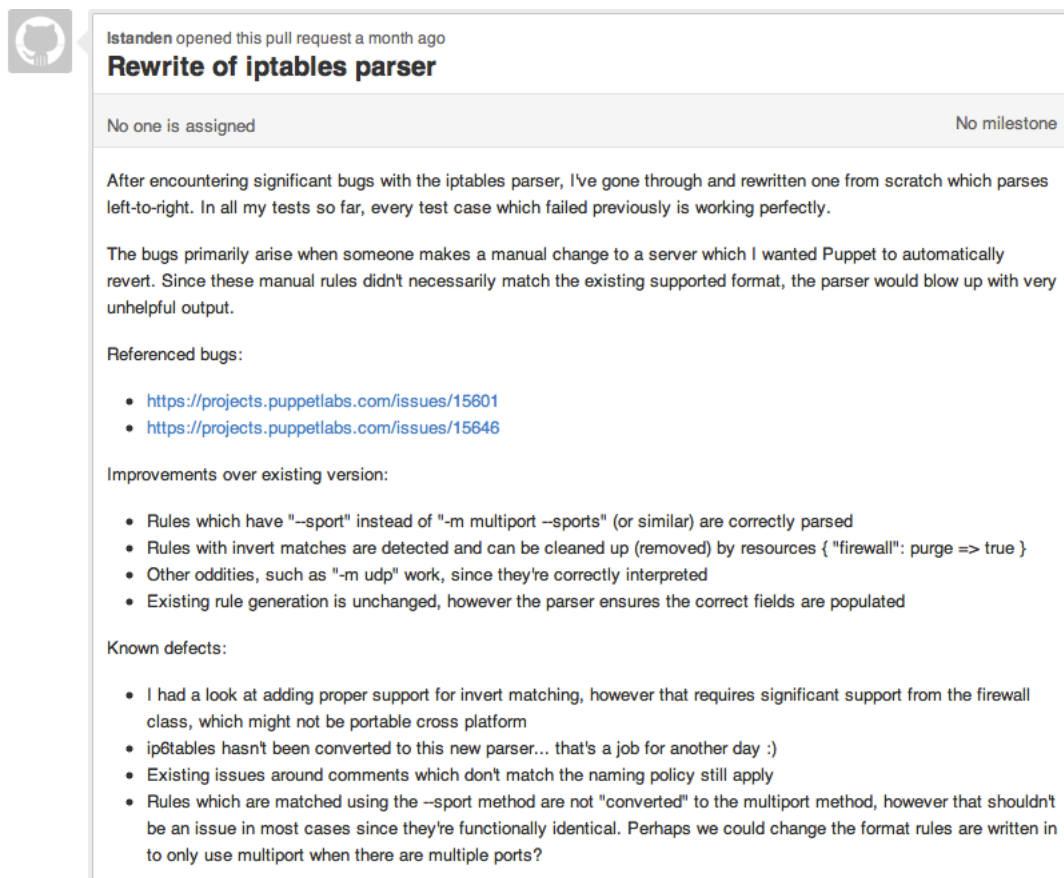
Some more information of how to install and to use the module can be found here: <http://github.com/puppetlabs/puppetlabs-firewall>

## Objectives

### New Parser

Firstly I had to solve some minor bugs on the module parser. At the time this report is written, there is a pull request at github repository of this module, which introduces a new and better parser for the iptables rules.

(<http://github.com/puppetlabs/puppetlabs-firewall/pull/88>)



The screenshot shows a GitHub pull request interface. At the top left is the GitHub logo. The title of the pull request is "Rewrite of iptables parser" by user "Istanden", who opened it a month ago. Below the title, it indicates "No one is assigned" and "No milestone". The main body of the pull request contains the following text:

After encountering significant bugs with the iptables parser, I've gone through and rewritten one from scratch which parses left-to-right. In all my tests so far, every test case which failed previously is working perfectly.

The bugs primarily arise when someone makes a manual change to a server which I wanted Puppet to automatically revert. Since these manual rules didn't necessarily match the existing supported format, the parser would blow up with very unhelpful output.

Referenced bugs:

- <https://projects.puppetlabs.com/issues/15601>
- <https://projects.puppetlabs.com/issues/15646>

Improvements over existing version:

- Rules which have "--sport" instead of "-m multiport --sports" (or similar) are correctly parsed
- Rules with invert matches are detected and can be cleaned up (removed) by resources { "firewall": purge => true }
- Other oddities, such as "-m udp" work, since they're correctly interpreted
- Existing rule generation is unchanged, however the parser ensures the correct fields are populated

Known defects:

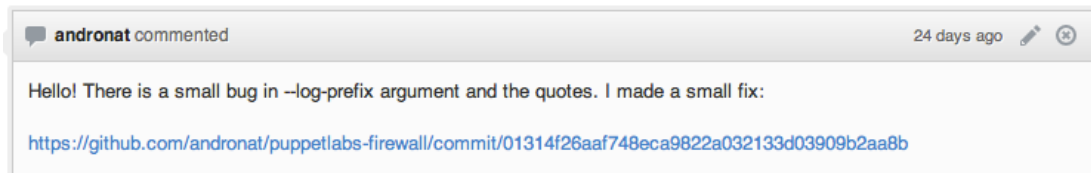
- I had a look at adding proper support for invert matching, however that requires significant support from the firewall class, which might not be portable cross platform
- iptables hasn't been converted to this new parser... that's a job for another day :)
- Existing issues around comments which don't match the naming policy still apply
- Rules which are matched using the --sport method are not "converted" to the multiport method, however that shouldn't be an issue in most cases since they're functionally identical. Perhaps we could change the format rules are written in to only use multiport when there are multiple ports?

### Double Rules

The new parser solved some of the problems. But there were more, like the double existence rule problem (<https://projects.puppetlabs.com/issues/15702>). In this bug, if a rule that you wanted deleted, exists more than once in your iptables rule table, it would be deleted only once.

## Rule Arguments

Although the new parses seems to work great, some minor bugs still existed. Some arguments like “--log-prefix” couldn’t be parsed correctly.



## New Feature

The main part of my work is focused on a new feature that will give the possibility to the module, to ignore certain firewall chains. For instance, you will be able to declare a chain at the Puppet template like this:

```
firewallchain { 'MY_CHAIN_2:filter:IPv4':  
  ensure => present,  
  managed => false  
}
```

and every rule that is inside this chain will be ignored for management.

This feature will help the collaboration between OpenStack and Puppet. Nova (which is part of OpenStack) is dynamically generating rules for the VMs. Until now, Puppet was overwriting every change OpenStack was making.

## Implementation

### New Parser

In my point of view, git is the best tool to manage source code from many different sources. In this part of my project I had to merge the main code from puppetlabs (<http://github.com/puppetlabs/puppetlabs-firewall>) with the new parser source code (<http://github.com/lstanden/puppetlabs-firewall>).

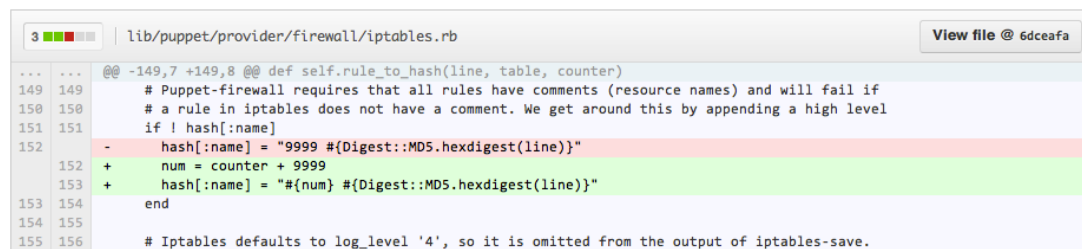
A good approach to make merges like of this kind, is to fork the main project from github, and then add as a remote source the project you want to merge and change. In our example we fork the puppetlabs master branch and we add as a remote source the new parser. We should fetch the new branch as a different branch to our repository.

### Double Rules

After some testing to the new parser I found the double rule bug. This bug was caused because the rules that had to be deleted, had the same name. The fix was easy. I just added a unique number at front of each rule.

(<https://projects.puppetlabs.com/issues/15702>)

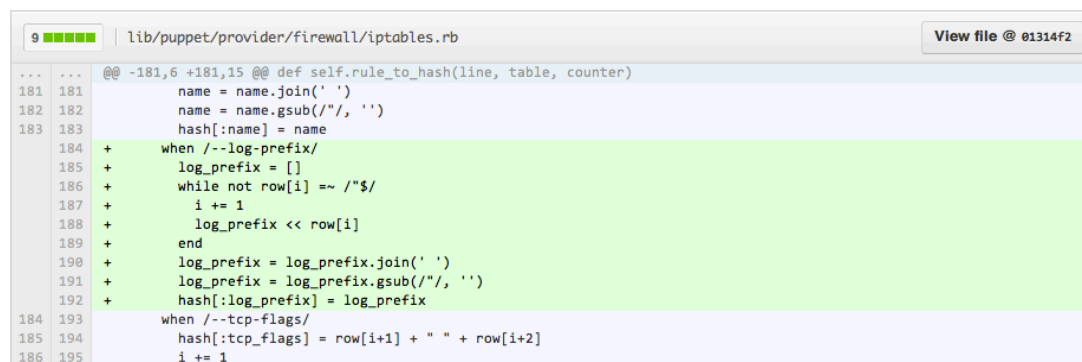
(<http://github.com/puppetlabs/puppetlabs-firewall/pull/90>)



```
3 | lib/puppet/provider/firewall/iptables.rb | View file @ 6dceaafa
... .. @@ -149,7 +149,8 @@ def self.rule_to_hash(line, table, counter)
149 149     # Puppet-firewall requires that all rules have comments (resource names) and will fail if
150 150     # a rule in iptables does not have a comment. We get around this by appending a high level
151 151     if ! hash[:name]
152 152     -   hash[:name] = "9999 #{Digest::MD5.hexdigest(line)}"
153 153     +   num = counter + 9999
154 154     +   hash[:name] = "#{num} #{Digest::MD5.hexdigest(line)}"
155 155     end
156 156     # Iptables defaults to log_level '4', so it is omitted from the output of iptables-save.
```

### Rule Arguments

As the testing continues, I found a problem parsing the "--log-prefix" argument. The exact problem was the quoting. The new parser was expecting this argument to have a strip string (without quotes and be only on word). By adding some code I managed to make it parse it correctly.



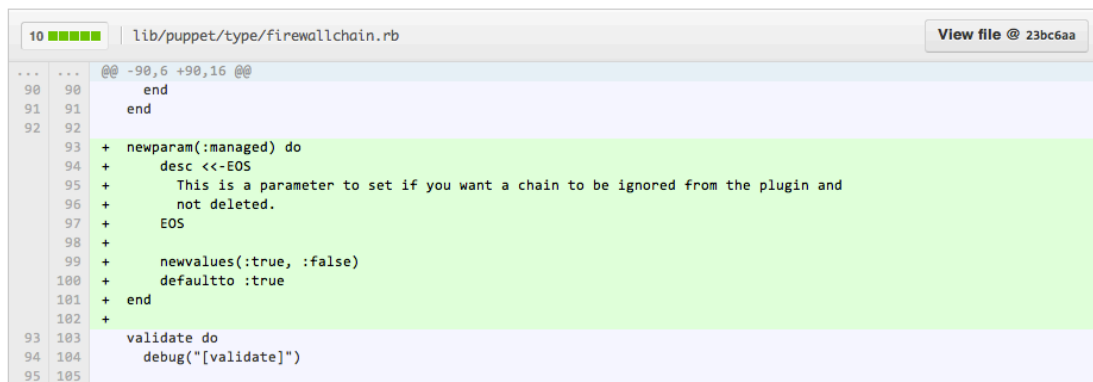
```
9 | lib/puppet/provider/firewall/iptables.rb | View file @ 01314f2
... .. @@ -181,6 +181,15 @@ def self.rule_to_hash(line, table, counter)
181 181     name = name.join(' ')
182 182     name = name.gsub('/', '')
183 183     hash[:name] = name
184 184     +   when /--log-prefix/
185 185     +     log_prefix = []
186 186     +     while not row[i] =~ /\$/
187 187     +       i += 1
188 188     +       log_prefix << row[i]
189 189     +     end
190 190     +     log_prefix = log_prefix.join(' ')
191 191     +     log_prefix = log_prefix.gsub('/', '')
192 192     +     hash[:log_prefix] = log_prefix
193 193     when /--tcp-flags/
194 194     hash[:tcp_flags] = row[i+1] + " " + row[i+2]
195 195     i += 1
```

## New Feature

As we explained before, the objective at this point is to make the module ignore rules that are declared at some specific firewall chains.

The Puppet module writing interface, it is separated in two major components, lib/puppet/type and lib/puppet/provider.

The lib/puppet/type is where you declare the interface of your module. At this point you declare the variables and actions that your module can provide. In our example in the type 'firewallchain' we will declare our new action 'managed'.



```
lib/puppet/type/firewallchain.rb
@@ -90,6 +90,16 @@
...
90 end
91 end
92
93 + newparam(:managed) do
94 +   desc <<-EOS
95 +   This is a parameter to set if you want a chain to be ignored from the plugin and
96 +   not deleted.
97 +   EOS
98 +
99 +   newvalues(:true, :false)
100 +   defaultto :true
101 + end
102 +
93 103 validate do
94 104   debug(["validate"])
95 105
```

Next step is to code how the module is working. This will happen inside lib/puppet/provider. Puppet has some 'key' functions to make a workflow of how you should write your provider. In our case the 'key' function that we should do our changes is self.instances. This function is reading the every rule existing in the firewall of the current client. This means that we should create a method there that will give us the control of which rule will be loaded and which one will not.

```
def self.instances
  debug ["instances"]
  table = nil
  rules = []
  counter = 1

  # Get unmanaged chains and cache the result.
  @@unmanaged_chains ||= unmanaged_chains_from_catalog

  # String#lines would be nice, but we need to support Ruby 1.8.5
  iptables_save.split("\n").each do |line|
    unless line =~ /^#\s+|^:\s+|^COMMIT|^FATAL/
      if line =~ /\s*/
        table = line.sub(/\s*/, "")
      else
        if hash = rule_to_hash(line, table, counter)
          if !@@unmanaged_chains.include? hash[:chain]
            rules << new(hash)
            counter += 1
          end
        end
      end
    end
  end
end

rules
end
```

As you can see we created a class variable named '@@unmanaged\_chains' where we collect the names of the chains that should we ignored from Puppet. (How we collect the chain names will be explained in a minute.) The function 'rule\_to\_hash' is iterating all the rules that exist in the current machine and returns each rule as a hash. So we have to check of each rule if the chain of the rule is inside the collection of unmanaged\_chains variable. If it is we skip the rule.

One more thing I want to mention at this point is that I used class variable, instead of an instance variable or a local variable, for caching reasons. Puppet seems to do a lot of calls of self.instances for a single run. So using '@@unmanaged\_chains ||= unmanaged\_chains\_from\_catalog' we cache our results.

The way we collect the unmanaged chain names is through the method 'unmanaged\_chains\_from\_catalog'.

```
# Get all chains that have 'managed' attribute equal to false.
def self.unmanaged_chains_from_catalog
  unmanaged_chains = []

  # Get all chains from the current catalog.
  Puppet::Face[:catalog, :current].find(Puppet[:certname]).resources.each do |resource|
    if resource.type =~ /Firewallchain/
      # From all declared chains that have attribute 'managed => false', find the names.
      if !resource[:managed]
        unmanaged_chains << resource.to_hash[:name].split(':').first
      end
    end
  end
end

unmanaged_chains
end
```

This method is calling the Puppet catalog from memory and asks for all the resources of the current machine. It collects all Firewallchain types and then keeps only those that have the managed attribute equals to false. Then returns an array of all the chain names.

One last thing that I must mention is that at the end of each run, Puppet is calling a function called 'flush' to clear unwanted variables and conditions. We use this function to clear the '@@unmanaged\_chains' class variable so we can be sure that in each run, Puppet is always reading the latest catalog from memory.

```
# Flush the property hash once done.
def flush
  debug("[flush]")
  if @property_hash.delete(:needs_change)
    notice("Properties changed - updating rule")
    update
  end
  @property_hash.clear

  # Clear unmanaged chains in each run.
  @@unmanaged_chains = nil
end
```



## **Conclusion and Future Work**

The project is still beta. A lot of work must to be done for a production level product. Numerous bugs still exist and if someone wants to use the module must test it thoroughly.

As a future work proposal, the module can be extended to also handle firewall tables. A good hierarchy approach much be used though. My opinion is that the module must be splited in three different types (firewalltable, firewallchain, firewallrule) and each type must 'autorequire' each other from the table to chain to rule. In this way the instances of tables will be created first and from biggest (table) to small (rule), will be a hierarchy approach and management.

## Puppet and Landb integration

Landb is an online database where CERN stores information about networks and network hardware.

Puppet is a management tool that could use this stored information to automate configuration to lots of devices that need information from this database.

But this to happen we need a ruby wrapper of the communication protocol the database and a Puppet function to call this wrapper.

## Objectives

### Ruby gem

As a second part of my work at CERN Openlab, was to create a ruby library (actually a ruby gem) that will communicate with Landb through SOAP protocol, and retrieve information. For instance, for a given device name we must fetch the room of the device location. (<https://network.cern.ch/sc/soap/4/>)

### Example

```
cl = LandbClient.instance
r = cl.get_device_info(['PCITCS57'])
r.device_info.location.room
⇒ "0005"
```

### Puppet function

Later on, we need a Puppet function to integrate the gem with Puppet. We also need a DSL (Domain Specific Language) so the end user can ask the information he wants from Puppet.

```
$hash = cern_lan_db_func({ "method" => "get_device_info",
                        "method_arguments" => "PCITCS57",
                        "responce_info" => [["device_info", "responsible_person", "name"], ["device_info", "responsible_person", "email"]]
                      })
notify { $hash: }
```

## Implementation

### Ruby gem

The project is consisted of two major classes. The LandbClient and the LandbResponse.

#### *LandbClient*

The LandbClient is a dynamic class that reads a WSDL document and creates its instance methods from the SOAP actions of the WSDL document. For instance, at landb SOAP server there is an action called GetAuthToken. This ruby gem is creating an instance method called `get_auth_token` from the above SOAP action.

```
token = client.get_auth_token ["username", "password", "NICE"]
```

The whole mechanism of creating the methods dynamically is in the 'initialize' method of the class. The names of the SOAP actions are taken from a supplementary gem called 'Savon'.

As the developers of 'Savon' (<http://savourb.com>) say: 'Savon helps you talk to SOAP services. It abstracts a lot of insanity.'

One more interesting thing that landb gem can make, is to understand how many arguments each SOAP action has, through the WSDL document. There are two methods 'help\_all\_operations' and 'help\_arguments\_for\_operation(operation)' which can help the user find all the actions and the arguments of each action.

#### *LandbResponse*

The LandbResponse is a wrapper class for the SOAP responses of landb. This class takes the SOAP response as a hash and constructs ruby objects with methods and variables with the same names as the retrieved information.

Example

```
cl = LandbClient.instance
r = cl.get_device_info(['PCITCS57'])
r.device_info.location.room
⇒ "0005"
```

At this example, r is a LandbResponse class.

### Puppet function

As a second requirement for my project, I created a Puppet function that uses the above gem and retrieves information from landb.

We needed a custom DSL for the Puppet manifests to be able to handle and use the gem. An example of this DSL is:

```
$hash = cern_lan_db_func({ "method" => "get_device_info",  
                        "method_arguments" => "PCITCS57",  
                        "response_info" => [["device_info", "responsible_person", "name"], ["device_info", "responsible_person", "email"]]  
                      })  
notify { $hash: }
```

As you can see the function accepts a hash as an argument. This hash can contain 3 keys and values for each key.

The first key 'method' is responsible for the SOAP action we want to call. In our example we call 'get\_device\_info' method.

The second key 'method\_arguments' accepts the arguments for the SOAP action. If the arguments are more than one, the Puppet template should use an array to pass the arguments.

The third key 'response\_info' is responsible for collecting the information from the response of the above SOAP action. It accepts an array of strings that represents a path of chained methods of the LandbResponse class. In our example we request two different information. The return values will be inserted in an array and will be saved in \$hash Puppet variable.

## Conclusion

The code and gem are all at this url: (<https://gitgw.cern.ch/gitweb/?p=gem-landb.git;a=summary;js=1>) for more information there are lots of REAME files inside the repository.

## References

1. <http://www.puppetlabs.com>
2. <http://docs.puppetlabs.com>
3. <http://forge.puppetlabs.com>
4. <http://forge.puppetlabs.com/puppetlabs/firewall>
5. <https://github.com/puppetlabs/puppetlabs-firewall>
6. <https://github.com/andronat/puppetlabs-firewall>
7. <http://guides.rubygems.org>
8. <http://www.ruby-lang.org>
9. <https://gitgw.cern.ch/gitweb/?p=gem-landb.git;a=summary;js=1>
10. <ssh://gitgw.cern.ch:10022/gem-landb>
11. <https://network.cern.ch/sc/soap/4/>
12. <http://openlab.web.cern.ch>