

sshproxy-0.5 plugin API documentation

Index

- 1 Introduction
- 2 The core proxy API
- 3 The Config API
- 4 The command line option API
- 5 The ACL system API
- 6 The Crypto engine API
- 7 Case studies as examples
- 8 Comments

1 Introduction

sshproxy, starting with version 0.5, has a powerfull plugin API that allows an administrator to easily add features to the ACL engine, the crypto engine, the command line options, or the core proxy itself.

In this documentation, we'll see how it is done and what it can give you.

2 The core proxy API

Almost all classes in **sshproxy** can be extended and derived from a plugin. These classes inherit from the **Registry** class that takes care of keeping track of any inheritance, if it was done properly.

To inherit your own class from a *registered* class from a plugin, and make it available to the application and other plugins, you have to follow a few steps.

First, you must get the registered class with the following code:

```
from sshproxy import get_class
theBaseClass = get_class('BaseClass')
```

If the class you want to inherit from is the **Server** class, then replace '*BaseClass*' with '*Server*'.

Then you can inherit your own class from **theBaseClass**:

```
class MyClass(theBaseClass):
```

Now, if you want your class to do some initialization on instanciacion, you have to use the *__reginit__()* method instead of *__init__()*:

```
# do not use __init__() for a Register-inherited class !!!
def __reginit__(self, myarg, *args, **kw):
    self.myarg = myarg
```

```
theBaseClass.__reginit__(self, *args, **kw)
```

Don't forget to call the base class' `__reginit__()` method, unless you are certain it doesn't need it.

You may have to call also the base class method for each method you overload.

After your class definition, you have to register your class into the **Registry**:

```
MyClass.register()
```

That's it, your class will now be instantiated and used instead of the standard class you inherited from.

3 The Config API

If you want your plugin to be configurable, you may want to have your own section in the *sshproxy.ini* configuration file.

Doing this is just a matter of creating a class inherited from `config.ConfigSection`, and registering the name of the section.

First, let's see how to create your own `ConfigSection` class:

```
from sshproxy import config

class MyPluginConfig(config.ConfigSection):
    section_defaults = {
        'hello_string': 'hello world!',
        'path_to_data': '@myplugin',
        'max_entries': 10,
    }
    types = {
        'path_to_data': config.path,
        'max_entries': int,
    }
```

What we've done here is to declare all default values we want in our config section. When **sshproxycd** is run, it writes them down in the *sshproxy.ini* file if they are not already present.

The *types* attribute declares the type of each config option. Notice that the *hello_string* option has no type. This is because the default type is *str*, which is exactly what we want for this option, so we don't need to specify it.

The type of *path_to_data* is *config.path*. This is a special type that allows for the '@' sign at the beginning of the path to be expanded to the configuration directory, which is *\$HOME/.sshproxy/* by default.

Now we've created our config section, we need to give it a name, and by the same occasion, to register it to the **Config** handler:

```
config.Config.register_handler('myplugin', MyPluginConfig)
```

That's it for the declaration part.

Now we need to get the values the user may have entered into our config section. This is done with the **config.get_config()** method:

```
# the standard way
path_to_data = config.get_config('myplugin')['path_to_data']

# this form can be used for options that have no default in our
# handler
email = config.get_config('myplugin').get('email',
'david@guerizec.net')

# this is an alternate form
path_to_data = config.get_config()['myplugin']['path_to_data']
```

If you need to store values into options, you can also do it with any value that can convert to a string:

```
myconfig = config.get_config('myplugin')
myconfig['http_port'] = 8080
# write it to the file
myconfig.write()
```

4 The command line option API

To create new command line options, you have to extend the **Server** class. For this, you need to have read The core proxy API section of this manual.

To add a command line, you have to register your command:

```
from sshproxy import get_class

Server = get_class('Server')

class MyServer(Server):
    def add_cmdline_options(self, parser):
        base_class.add_cmdline_options(self, parser)
        parser.add_option("", "--list-clients", dest="action",
            help="dump the list of clients.",
            action="store_const",
            const="list_clients",
        )
```

This registers the option *--list-clients* to the command line of **pssh**.

Now, we need to write the code that will handle the action when this option is invoked. For that we need to write a method of **Server** that starts with *opt_*, and the name of the method must be the one we gave in the *const* argument of *parser.add_option*:

```
# This method return the list of clients in the database
def opt_list_clients(self, option, args):
    clients = self.pwdb.get_client().list_users()
    return '\n'.join([ c.username for c in clients ])
```

Don't forget to register your new **Server** class:

```
MyServer.register()
```

This example is just that, an example. For a production use, you may want to add an ACL check to show the option only to the users with admin rights.

5 The ACL system API

You can extend the ACL system by creating new functions, and by checking new ACL rules.

You create new ACL functions by the same mechanism as we've seen before in The command line option API.

Let's create a function called 'temp' that will give the temperature of the servers room:

```
# first import your own module to get temperature
from facility import server_room

from sshproxy import get_class

base = get_class('ACLRuleParser')

class MyACLRuleParser(base):
    def func_temp(self, *args):
        if len(args) != 1:
            log.warning("Warning: function temp takes exactly 1
argument.")
            return

        # get the server room temperature
        room = server_room.lookup(ip_address=args[0])
        if not room:
            log.warning("Warning: in acl:temp(), ip address %s
not found." % args[0])
            return

        temp = room.get_temperature()
        # return the temperature
        return temp

MyACLRuleParser.register()
```

An ACL function is defined by declaring a method prefixed by 'func' in the ACLRuleParser class.

In the ACL rules, you can use your *temp* function:

```
emergency_backup:
    temp(site.ip_address) > 40 and client.username = "backup"

remote_exec:
    acl(emergency_backup) and proxy.cmdline = "tar zcvf - /etc"
```

This is a silly example, no one would do its backups when there is an emergency, but you get the idea.

6 The Crypto engine API

TODO

7 Case studies as examples

TODO

8 Comments

Add your comments here.

SshProxy/PluginDocV0.5 (last edited 2006-09-19 12:21:21 by DavidGuerizec)