CESNET Technical Report 02/3013

Charon – Resource access layer

Jan Mach

Received 5.9.2013

Abstract

This paper describes the architecture, design and implementation of the resource access layer referred to as Charon. It is a pluggable framework that can be used to restrict access to a set of resources or commands users may access or invoke via SSH.

Keywords

SSH, resource access layer, remote command access restriction

Introduction

When designing and implementing the project management and support server, one of the requirements was a secure hosting of multiple Git repositories. A secure access to the Git repository is best done via SSH. First we came across and tested the Gitosis[1] project. At first glance it seemed perfect, but the biggest limitation from our perspective was that all users share the same account and may authenticate via SSH keys only. One of the requirements was to enable simple password authentication, because some of our users do not use SSH keys and authenticate against company`s LDAP server. So we developed our own solution to better accomodate our needs. Thus initially, Charon was just a little bit rewritten Gitosis. It served only for restricting access to the Git repositories. Later it was extended to allow restricting access to an arbitrary set of general commands that users can remotely execute on the host system. Thus now it is not merely a Git related tool, but it can be used to perform other tasks as well. Currently, we are for example using it to enable remote users to execute some package repository management related commands (besides the Git repository access).

Design and implementation

System overview and operation

Charon is a simple access layer written in Perl. It acts as a proxy and allows remote users to execute a defined set of commands on a local host system. Figure 1 below depicts the system overview and both operation modes it is capable of. In the first mode, Charon is configured as the login shell for a particular user[2] whose access the administrator wants to restrict. This configuration is done simply in the ***/etc/passwd*** configuration file. Upon successful authentication, ***charon*** is launched instead of the shell and following checks are performed:

1. Charon restricts *USER* from using SSH interactively. If no command was entered on the command line, the user will be denied access.

2. Where a command was entered, all command line arguments are parsed and the triplet *USER – ACCESS TYPE – RESOURCE* is detected.

3. Charon looks into the configurations and determines whether the *USER* was granted a particular *ACCESS TYPE* to a particular *RESOURCE*.

4. Where the permission was given, the appropriate command is generated and executed on behalf of the user, otherwise the user request is denied.

5. Every request is logged into log files.

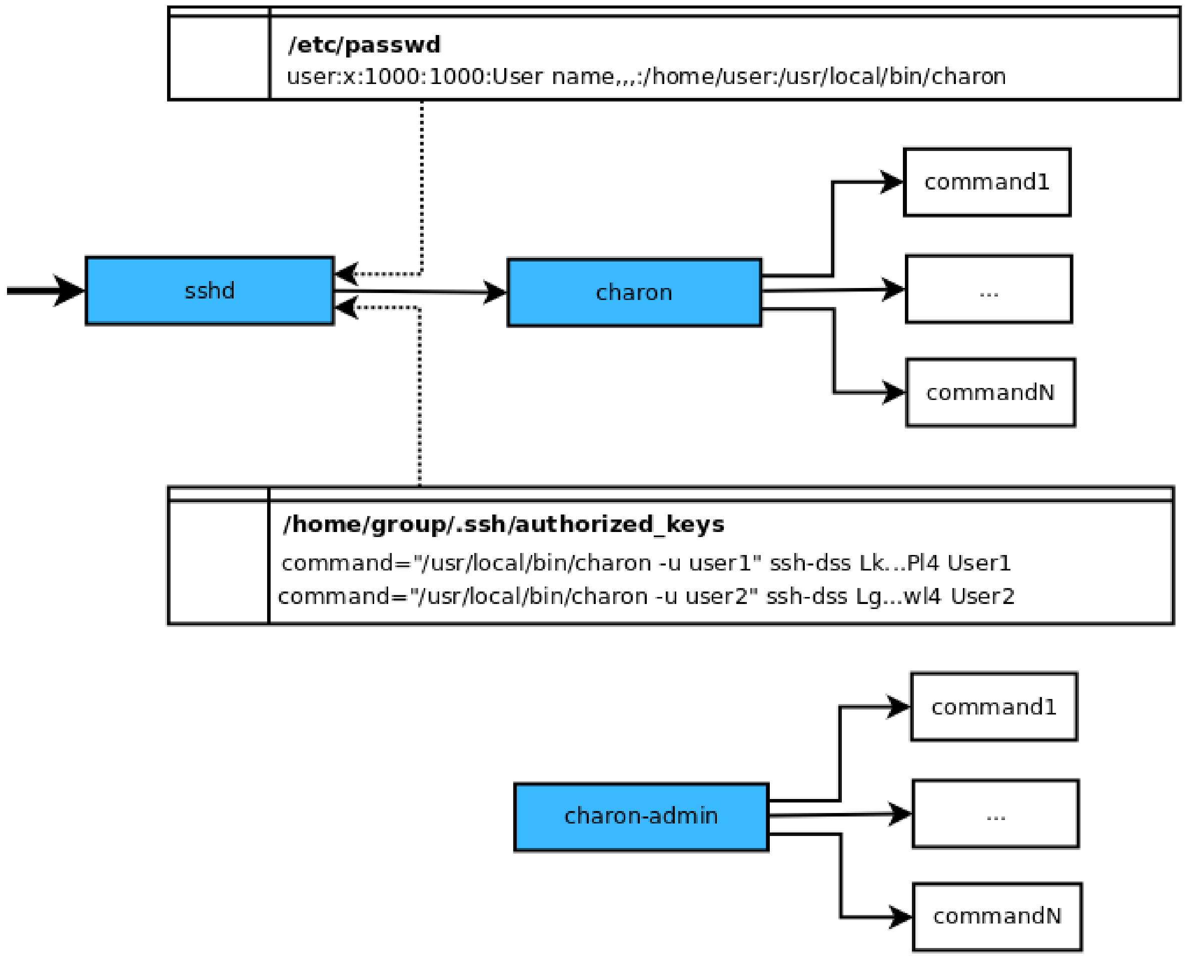

/etc/passwd
user:x:1000:1000:User name,,,:/home/user:/usr/local/bin/charon
sshd
charon
command1
...
commandN
/home/group/.ssh/authorized_keys
command="/usr/local/bin/charon -u user1" ssh-dss Lk...Pl4 User1
command="/usr/local/bin/charon -u user2" ssh-dss Lg...wl4 User2
charon-admin
command1
...
commandN

Figure 1: Charon deployment overview

The second operation mode is the same mode in which the Gitosis operates. There is an account for a group of users on the host system and users may authenticate only via SSH keys. User identities are then differentiated using the options field feature in the ***authorized_keys*** configuration file for OpenSSH server[3]. By using the ***command*** option *charon* is invoked instead of anything else that might have been entered on the command line or specified as the login shell. The user identity is passed as the argument and the original command which the user entered on the command line is stored within the designated environment variable. Upon startup the original command is retrieved and then the operation flow is the same as in the previous mode. Charon can run in both modes at the same time.

Aside from the main component of the Charon system, which is the ***charon***, stands the ***charon-admin*** utility. It is the framework for unifying all administration scripts and hiding them behind single management interface.

Charon ACL description

Charon uses simple ACL (Access Control List) rules for defining what resources a particular user may access and how. ACL rules are defined by the triplet *USER ACCOUNT – ACCESS TYPE – RESOURCE*. *RESOURCE* is defined by its unique identifier. It can be anything that the system administrator thinks is appropriate, for example the name of the Git repository or the name of the command available for execution by remote users. *ACCESS TYPE* is the type of the permission. Again, it can be anything that is needed. In most cases administrators will use the classical create-read-write-update-delete kind of permissions, but generally it can be any arbitrary string. And finally *USER ACCOUNT* is also an identifier of the user or group of users. This simplicity and someone would even say a kind of vagueness, is by design so that the ACL rules would fit as many applications as possible. By using this approach the ACL related modules can be used separately from the Charon package. The administrator first defines the ACL rules and then it is up to the application, in case of Charon it is up to the ***Charon::Command::Module***, to perform the parsing and translation of the command line given by the user and provide the Doorman with the requested ACL triplet.

Charon architecture

The most important part of the Charon system is the *charon* component. Its inner architecture is depicted on Figure 2. Upon startup Charon initializes the Doorman subcomponent, which gathers ACL information from various sources using appropriate ACL modules. Currently, there is only ***ACL::Source::File*** module for gathering ACL data from configuration files. It is bundled in the distribution but other modules can be created very easily by extending the prepared ***ACL::Source::Module*** base class and implementing its mandatory interface. For example, for our project support system we developed the Redmine and LDAP ACL modules. To put it simply, Doorman is designed to answer the question: “Was this *USER* granted this *ACCESS TYPE* to this *RESOURCE*?”.

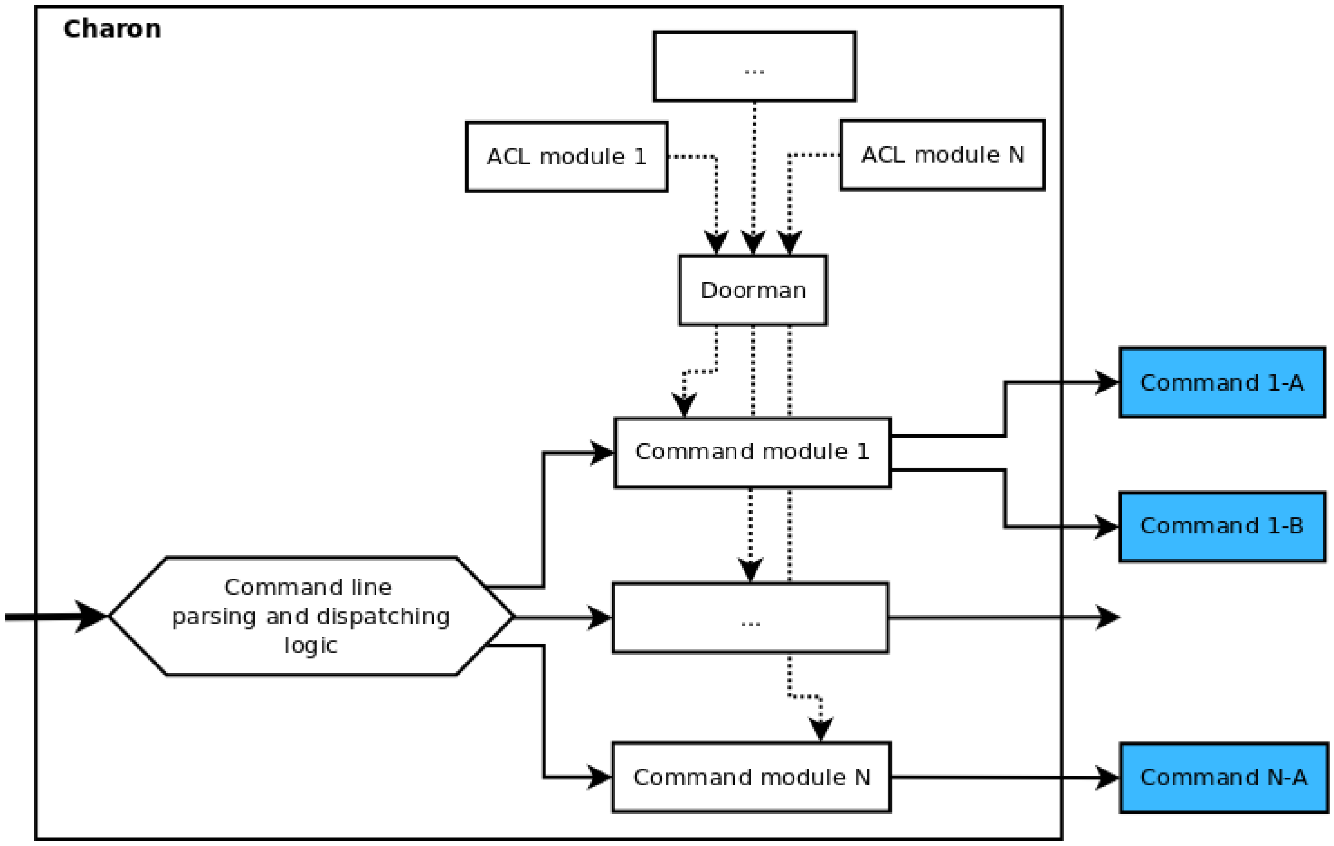


Figure 2: Inner charon architecture

After the Doorman initialization is done, *charon* dynamically detects and loads all present command modules which are responsible for the work as such. This operation results in a list of commands that are handled by a particular module. This information is fed to the command line parsing and dispatching logic. *Charon* then detects the mode, in which it is running and parses out the command name from the command line or environment variable. Where the command line is empty or the entered command is unknown, the processing is stopped immediately and an error message is returned to the user. Otherwise, the control is passed to the appropriate command module and user name, command name and other command arguments and options are passed as processing arguments. In case of single user mode, the name of the current user account is used. In case of group account mode, the name of the user from the command line argument is used.

Afterwards, the command module is responsible for performing the following actions:

1. Determining the *ACCESS TYPE* based on the command the *USER* is trying to execute.
2. Validating the command options and arguments
3. Parsing the command options and arguments and determining the name of the *RESOURCE* the user is trying to access.
4. Consulting Doorman whether the particular *USER* was granted this *ACCESS TYPE* to the *RESOURCE*.
5. In case the previous step is successful, the command is prepared to be executed. The original command specified by the user can now be translated into something else. It can be a completely different command generated according to some rules, or only sections of the user command such as paths or options can be modified.
6. Executing the command is executed on the host system.

Command modules can be easily created by extending the prepared ***Charon::Command::Module*** base class. The mandatory interface, that needs to be implemented for the module to be pluggable into the *charon*, is described on Figure 3.

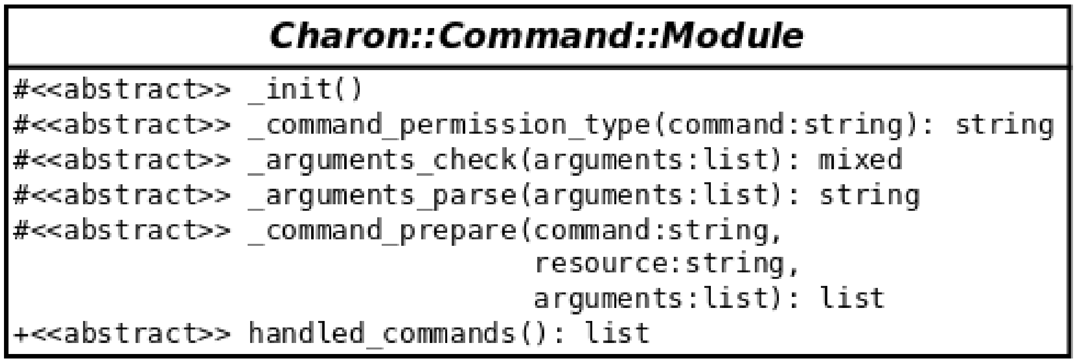

Charon::Command::Module
#<<abstract>> _init()
#<<abstract>> _command_permission_type(command:string): string
#<<abstract>> _arguments_check(arguments:list): mixed
#<<abstract>> _arguments_parse(arguments:list): string
#<<abstract>> _command_prepare(command:string, resource:string, arguments:list): list
+<<abstract>> handled_commands(): list

Figure 3: Charon::Command::Module abstract interface

The *_init* method can perform any initializations necessary for a proper module operation. The *_command_permission_type* method must return the proper ACCESS TYPE value for the given command name. The *_arguments_check* method should perform the command line validation and *_arguments_parse* method must parse the command line and return the RESOURCE name. The *_command_prepare* method is a place for the mentioned translation or change of the original command, that was specified by the user. And finally, the *handled_commands* must return a list of all commands that are handled by that particular module. All previous methods are hook methods and are called automatically at a proper time.

Charon-admin architecture

The second big part of the Charon system is called ***charon-admin***. The architecture of this component is depicted on Figure 4. The internal architecture is similar to the *charon* component but much simpler. Basically, *charon-admin* is just a smart command line parser, which dynamically detects and loads all administration modules, parses the command line arguments and then invokes the appropriate module method. Its purpose is to unify all administration scripts that the administrator must use to manage one or more services, that are somehow coupled together. It is a framework administrators can use to facilitate the management tasks.

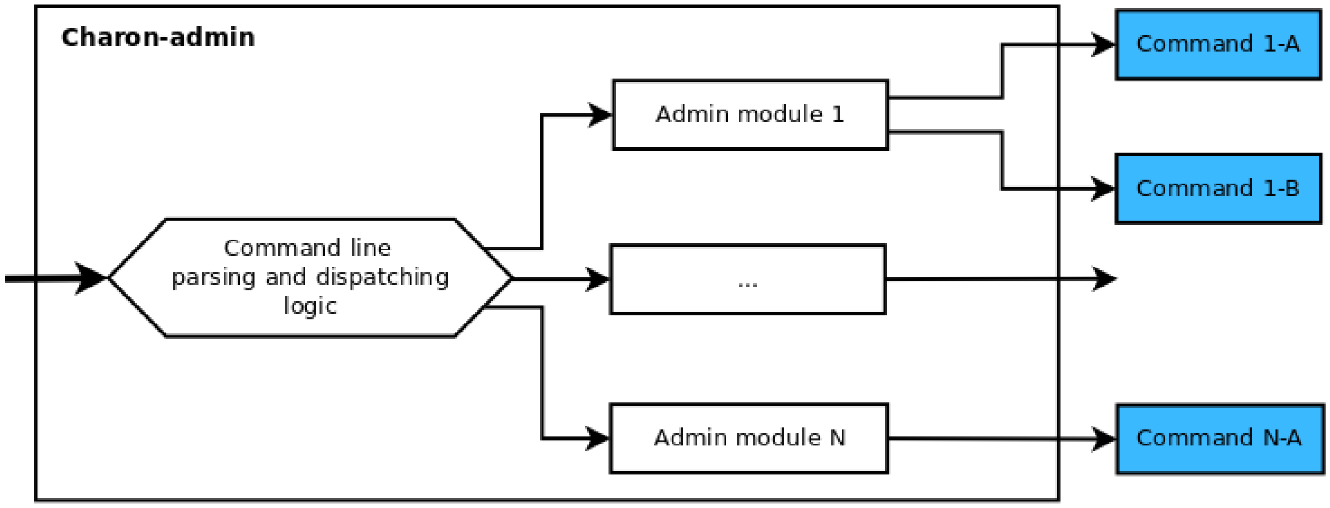
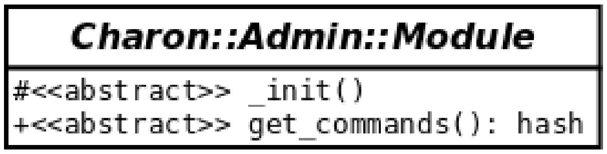


Figure 4: Inner charon-admin architecture

For example on our project management support server we are use *charon-admin* to dynamically generate the configuration files for services such as HTTP and FTP, to create more complex user accounts, view the overall system status, etc. And all these tasks are conveniently hidden under one administration interface which executes the command line processing, validates parameters and invokes appropriate administration script or tool. Again, creating a custom admin module is just the matter of extending the prepared ***Charon::Admin::Module*** base class. The mandatory interface that needs to be implemented for the module to be pluggable into *charon-admin* is described on Figure 5.



```
#<<abstract>> _init()
+<<abstract>> get_commands(): hash
```

Figure 5: Charon::Admin::Module abstract interface

This interface is much simpler. Aside from the *_init()* method, it is only necessary to implement the *get_commands()* method which must return a hash structure describing the command tree provided by the module. A detailed description of this structure's syntax can be found in the documentation. Parsing the command line arguments is then done according to this structure and the control together with any additional arguments is passed to the appropriate module method.

Charon configuration file syntax

Charon uses simple configuration file syntax for tweaking inner processing settings. Syntax is similar to the INI file syntax, but there are few differences. The following code is the example of a configuration file:

```
# Comment
some_key         = some value containing spaces
hash_key.first   = valueA
hash_key.second  = valueB
{include /path/to/config.conf}
```

Syntax does not support multi line declarations. One line must contain exactly one configuration declaration. Anything after the '#' character at the beginning of the line is treated as a comment. The configuration keyword is expected at the beginning This keyword must not begin with '#', '[', or '{' character and must not include any spaces. The '.' character in the keyword name may be used as a separator and hierarchical structure (hash of hashes in Perl) is then created instead. After the keyword, any number of whitespace characters may be used for readability purposes, then the '=' character is expected, again any number of whitespace characters and anything else to the end of line is treated as a value. Syntax also supports including other files, this is the last line in the example above. Parsing the configuration from the example will result in the following Perl hash structure (without the contents of the unknown */path/to/config.conf* file).

```
$VAR1 = {
          'some_key' => 'some value containing spaces',
          'hash_key' => {
                          'first'  => 'valueA',
                          'second' => 'valueB' ,
                        }
        };
```

Doorman ACL::Source::File configuration file syntax

Syntax of ***ACL::Source::File*** configuration file syntax is similar to the syntax of *charon* main configuration file. Below is the example of a configuration file:

```
[general]

perm write = userA
perm read  = __ALL__
```

```
[resource res_id1]
attr has_git_repo      = true
attr gpg_key           = ABC123
perm write             = group1

[resource res_id2]
attr public      = true
perm read        = user3

[group group1]
members    = user1,user4,user5

[aliases]
user5 = user1
```

There are four types of sections that may appear in the ACL configuration file. In the '*general*' section, the administrator may give global access type permissions, that will apply to all resources. In the example above, '*user1*' may write to any resource and, by using '*__ALL__*' keyword, all users may read all resources. The configuration file may then contain one or more '*resource*' sections in which arbitrary attributes and access permissions can be set for the resource with an appropriate identifier. Attributes can then be used by a developer in the module source code. Access permissions can be given either to the accounts, or to the groups. Groups can be defined within the '*group*' sections. The last type of configuration file section is the '*aliases*' section. Doorman may use more than one ACL source modules. In most cases, some sort of account mapping must be used because a single user may have different account names in different sources (LDAP, database, local system account, etc.).

Charon configuration file contents

Some of the configuration directives in the configuration file are hard coded and used by Charon itself, but developers of the Charon modules may define any number of arbitrary keywords, which can be then easily used in the module source code via provided interface.

Below is the list of Charon specific configurations:

```
log_threshold    = DEBUG
log_file         = /var/log/charon.log

re_resource_name       = [-_a-zA-Z0-9]+
re_attribute_name      = [-_a-zA-Z0-9]+
re_account_name        = [-_a-zA-Z0-9.@]+
re_alias_name          = [-_a-zA-Z0-9.@]+

acl_all_accounts       = __ALL__
acl_all_resources      = __ALL__
```

```
perms_list             = create, read, write, delete
perms_order            = create, read < write, delete

acls.file  = /etc/charon/doorman.conf
```

Configurations beginning with '*log_*' set up the logging feature. Configurations beginning with '*re_*' define regular expressions that will be used to validate resource, attribute, account and alias names in the Doorman file ACL configuration file. Configurations beginning with '*acl_all_*' define the keyword for catchall account and resource. This keyword can be then used in Doorman ACL configuration file to assign all users some access permission to some resource, or all users some access permission to some resource. In the '*perms_list*', the administrator must define a list of valid access permissions. Anything else will be treated as invalid. In the '*perms_order*', the administrator may define that some permissions include others. In the example above, the write permission automatically grantsthe read permission. And finally, a list of ACL modules to use for Doorman initialization must be given.

Conclusions

Charon is an easy-to-use proxy system (or perhaps it would be better described as a framework) for restricting access to local resources and invoking local commands by remote users via SSH. It is designed to be used either instead of the login shell for single user accounts, or invoked via *command* option from *authorized_keys* of the group user accounts. The system is composed of flexible and modular ACL control system for defining access rules and pluggable modules which then perform validation, preparation and execution of a given command. Another part of the system is a pluggable administration tool for hiding all necessary administration scripts behind one interface.

Acknowledgements

This work is supported by the research intent MSM6383917201 of the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1]: Gitosis, http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way

[2]: Debian Manual, http://www.debian.org/doc/manuals/debian-reference/ch04.en.html

[3]: OpenSSH Documentation, http://www.openssh.org/manual.html