

Exscript Release 0.9.16

User Documentation

A scripting language and framework for terminal based protocols

Samuel Abels

July 26, 2009

Contents

1	Introduction	2
1.1	Why Exscript?	2
1.2	Legal Information	2
1.3	Contact Information & Feedback	2
2	Overview	3
2.1	Quick Introduction	3
2.2	Talking To Multiple Devices At The Same Time	3
2.3	Advanced Command Templates	3
3	Command Line Syntax	4
3.1	Overview	4
3.2	Using Account Pooling	4
3.3	Using a CSV file as input	4
4	Language Syntax	5
4.1	Sending Commands To The Remote Host	5
4.2	Comments	5
4.3	Using Variables	5
4.4	Adding Variables To A List	6
4.5	Using Built-in Variables	6
4.6	Using Expressions	6
4.6.1	Priority 1 Operators	7
4.6.2	Priority 2 Operators	7
4.6.3	Priority 3 Operators	7
4.6.4	Priority 4 Operators	7
4.6.5	Priority 5 Operators	7
4.6.6	Priority 6 Operators	7

4.7	Using Hexadecimal Or Octal Numbers	8
4.8	Using Regular Expressions	8
5	<i>Exscript</i> Commands	8
5.1	Extracting Data From A Response	8
5.1.1	extract ... into	8
5.1.2	extract ... into ... from	9
5.1.3	extract ... as	9
5.2	Using If-Conditions	9
5.2.1	if ... end	9
5.2.2	if ... else ... end	9
5.2.3	if ... else if	10
5.3	Loops	10
5.4	Loops And Lists	11
5.5	Using Functions	11
5.6	Exiting A Script	13
5.6.1	fail "message"	13
5.6.2	fail "message" if	13
5.7	Error Handling	13
6	Trouble Shooting	14
6.1	Common Pitfalls	14
6.2	Deadlocks	14
6.3	A Command Is Sent Too Soon	15
6.4	The Connection Is Closed Too Soon	16

1 Introduction

1.1 Why Exscript?

Exscript is a script and template language for automating network connections over protocols such as Telnet or SSH. *Exscript* is targeted at non-developers and developers alike.

Exscript is often used to automate sessions with routers from Cisco, Juniper, Huawei, and others. It may be used by an administrator who often configures machines running Linux/Unix, IOS, IOS-XR, JunOS, VRP, or any other operating system that can be used with a terminal.

Exscript is in some ways comparable to Expect, but has some unique features that make it a lot easier to use and understand for non-developers.

1.2 Legal Information

Exscript and this handbook are distributed under the terms and conditions of the GNU GPL (General Public License) Version 2. You should have received a copy of the GPL along with *Exscript*. If you did not, you may read it here:

<http://www.gnu.org/licenses/gpl-2.0.txt>

If this license does not meet your requirements you may contact us under the points of contact listed in the following section. Please let us know why you need a different license - perhaps we may work out a solution that works for either of us.

1.3 Contact Information & Feedback

If you spot any errors, or have ideas for improving *Exscript* or this documentation, your suggestions are gladly accepted. We offer the following contact options:

Google Groups: <http://groups.google.com/group/exscript/>
Bug tracker: <http://code.google.com/p/exscript/issues/list>
Phone: +49 176 830 40288
Jabber: knipknap@jabber.org

2 Overview

2.1 Quick Introduction

With *Exscript* you can quickly automate a conversation with a device over Telnet or SSH. For example, to execute the “ls” command on three different hosts, create a file with the following content:

```
ls
```

and then run it using

```
exscript my_template host1 host2 host3
```

2.2 Talking To Multiple Devices At The Same Time

With *Exscript* you can automatically parallelize your connections, such that multiple sessions are opened at the same time. This can speed up the time in which a specific command is propagated within your network.

For example, imagine you want to execute the “clear ip bgp * soft” command on twenty different Cisco routers. Start by creating a text file with the following content:

```
clear ip bgp * soft
```

Save this file as `commands.exscript`. Also, create a text file that contains the list of hostnames to which the command should be sent:

```
host1  
host2  
...  
host20
```

Save this file as `hosts.txt`. To send this change to all routers at the same time, type the following command:

```
exscript —hosts hosts.txt -c15 commands.exscript
```

Note that the `-c15` option causes *Exscript* to open a maximum of fifteen connections at the same time. Once the first host out of these 15 is completed, *Exscript* opens the connection to the next host, until the “clear ip bgp * soft” command has been sent to all hosts.

2.3 Advanced Command Templates

Exscript templates support many more commands. For example, to automate a session with a Cisco router, the following template may be used:

```
show version {extract /^(cisco)/ as vendor}  
{if vendor is "cisco"}  
  show ip interface brief {extract /^(S+)\s/ as interfaces}  
  {loop interfaces as interface}  
    show running interface $interface  
  configure terminal
```

```
interface $interface
no shut
end
{end}
copy running-config startup-config
{end}
```

Exscript provides additional methods for interacting with the remote host and for receiving information from it. The following chapters include a more complete overview of the template language.

3 Command Line Syntax

3.1 Overview

You can pass parameters (or lists of parameters) into the templates and use them to drive what happens on the remote host. *Exscript* easily supports logging, authentication mechanisms such as TACACS and takes care of synchronizing the login procedure between multiple running connections. These features are enabled using simple command line options. The following options are currently provided:

3.2 Using Account Pooling

It is possible to provide an account pool from which *Exscript* takes a user account whenever it needs to log into a remote host. Depending on the authentication mechanism used in your network, you may significantly increase the speed of parallel connections by using more than one account in parallel. The following steps need to be taken to use the feature:

1. Create a file with the following format:

```
[account-pool]
user=password
other_user=another_password
somebody=yet_another_password
```

2. Save the file. It is assumed that you are aware of the security implications of saving your login passwords in a text file.
3. Start *Exscript* with the "--account-pool FILE" option. For example:

```
exscript --account-pool /home/user/my_accounts my.exscript host4
```

3.3 Using a CSV file as input

By providing the `-csv-hosts` option you may pass a list of hosts to *Exscript* while at the same time providing a number of variables to the script. The CSV file should have the following format:

```
hostname my_variable another_variable
myhost value another_value
yourhost hello world
```

Note that fields are separated using the tab character, and the first line must start with the string "hostname" and is followed by a list of column names.

In the Exscript, you may then access the variables using those column names:

```
ls -l $my_variable
touch $another_variable
```

4 Language Syntax

4.1 Sending Commands To The Remote Host

The simplest possible template is one that contains only the commands that are sent to the remote host. For example, the following *Exscript* can be used to retrieve the response of the `ls -l` and `df` commands from a unix host:

```
ls -l
df
```

Save this file as "my.exscript" and execute it using the following command:

```
exscript my.exscript localhost
```

where "localhost" is the name of the host on which the "ls -l" and "df" commands are executed.

4.2 Comments

Lines starting with a hash ("#") are interpreted as comments and ignored. For example:

1. *# This line is ignored...*
2. `{if hostname is "test"}`
3. *# ...and so is this one.*
4. `{end}`

4.3 Using Variables

The following template uses a variable to execute the `ls` command with a filename as an argument:

```
ls -l $filename
```

Execute:

```
exscript -d filename=.profile my.exscript localhost
```

Note that the `-d` switch allows passing variables into the *Exscript*. The example executes the command `ls -l .profile`. You can also assign a value to a variable within an *Exscript*:

```
{filename = ".profile"}
```

```
ls -l $filename
```

You may also use variables in strings by prefixing them with the “\$” character:

1. {test = "my test"}
2. {if "my test one" is "\$test one"}
3. # *This matches!*
4. {end}

In the above template line 3 is reached. If you don't want the “\$” character to be interpreted as a variable, you may prefix it with a backslash:

1. {test = "my test"}
2. {if "my test one" is "\\$test one"}
3. # *This does not match*
4. {end}

4.4 Adding Variables To A List

In *Exscript* every variable is a list. You can also merge two lists by using the “append” keyword:

1. {
2. test1 = "one"
3. test2 = "two"
4. append test2 to test1
5. }

This results in the “test1” variable containing two items, “one” and “two”.

4.5 Using Built-in Variables

The following variables are available in any *Exscript* template, even if they were not explicitly passed in:

1. **hostname** contains the hostname that was used to open the current connection.
2. **response** contains the response of the remote host that was received after the execution of the last command.

Built-in variables are used just like any other variable. You can also assign a new value to a built-in variable in the same way.

4.6 Using Expressions

An expression is a combination of values, variables, operators, and functions that are interpreted (evaluated) according to particular rules and that produce a return value. For example, the following code is an expression:

```
name is "samuel" and 4 * 3 is not 11
```

In this expression, *name* is a variable, *is*, *is not*, and *** are operators, and "samuel", 4, 3, and 11 are values. The return value of this particular expression is *true*.

In *Exscript*, expressions are used in many places, such as if-conditions or variable assignments. The following operators may be used in an expression.

4.6.1 Priority 1 Operators

1. *** multiplies the operators (numerically).
2. */* divides the operators (numerically).

4.6.2 Priority 2 Operators

1. *+* adds the operators (numerically).
2. *-* subtracts the operators (numerically).

4.6.3 Priority 3 Operators

1. *.* concatenates two strings.

4.6.4 Priority 4 Operators

1. *is* tests for equality. If both operators are lists, only the first item in the list is compared.
2. *is not* produces the opposite result from *is*.
3. *in* tests whether the left string equals any of the items in the list given as the right operator.
4. *not in* produces the opposite result from *in*.
5. *matches* tests whether the left operator matches the regular expression that is given as the right operator.
6. *ge* tests whether the left operator is (numerically) greater than or equal to the right operator.
7. *gt* tests whether the left operator is (numerically) greater than the right operator.
8. *le* tests whether the left operator is (numerically) less than or equal to the right operator.
9. *lt* tests whether the left operator is (numerically) less than the right operator.

4.6.5 Priority 5 Operators

1. *not* inverts the result of a comparison.

4.6.6 Priority 6 Operators

1. *and* combines two tests such that a logical AND comparison is made. If the left operator returns FALSE, the right operator is not evaluated.
2. *or* combines two tests such that a logical OR comparison is made. If the left operator returns TRUE, the right operator is not evaluated.

4.7 Using Hexadecimal Or Octal Numbers

Exscript also supports hexadecimal and octal numbers using the following syntax:

```
{
  if 0x0a is 012
    sys.message("Yes")
  else
    sys.message("No")
  end
}
```

4.8 Using Regular Expressions

At some places *Exscript* uses Regular Expressions. These are NOT the same as the expressions documented above, and if you do not know what regular expressions are it is recommended that you read a tutorial on regular expressions first.

Exscript regular expressions are similar to Perl and you may also append regular expression modifiers to them. For example, the following is a valid regular expression in *Exscript* :

```
/^cisco \d+\s+\w/i
```

Where the appended “i” is a modifier (meaning case-insensitive). A full explanation of regular expressions is not given here, because plenty of introductions have been written already and may be found with the internet search engine of your choice.

5 *Exscript* Commands

By default, any content of an *Exscript* is sent to the remote host. However, you can also add instructions with special meanings. Such instructions are enclosed by curly brackets (and). The following commands all use this syntax.

5.1 Extracting Data From A Response

Exscript lets you parse the response of a remote host using regular expressions. If you do not know what regular expressions are, please read a tutorial on regular expressions first.

5.1.1 extract ... into ...

If you already know what regular expressions are, consider the following template:

```
ls -l {extract /^(\d.*)/ into directories}
```

The extract command matches each line of the response of “ls -l” against the regular expression `/^(\d.*)/` and then appends the result of the first match group (a match group is a part of a regular expression that is enclosed by brackets) to the list variable named directories.

You can also extract the value of multiple match groups using the following syntax:

```
ls -l {extract /^(d\S+)\s.*\s(\S+)\$/ into modes, directories}
```

This extracts the mode and the directory name from each line and appends them to the modes and directories lists respectively. You can also apply multiple matches to the same response using the following syntax:

```
ls -l {
  extract /^[^d].*\s(\S+)/ into files
  extract /^[^d].*\s(\S+)/ into directories
}
```

There is no limit to the number of extract statements.

5.1.2 extract ... into ... from ...

When used without the “from” keyword, “extract” gets the values from the last command that was executed. You may however also instruct *Exscript* to extract the values from a variable. The following example shows how this may be done.

```
ls -l {
  extract /^(.*)/ into lines
  extract /^(d.*)/ into directories from lines
}
```

5.1.3 extract ... as ...

The “as” keyword is similar to “into”, the difference being that with as, the destination variable is cleared before new values are appended.

```
ls -l {extract /^(d.*)/ as directories}
```

“as” may be used anywhere where “into” is used.

5.2 Using If-Conditions

You can execute commands depending on the runtime value of a variable or expression.

5.2.1 if ... end

The following *Exscript* executes the “ls” command only if “ls -l .profile” did not produce a result:

```
ls -l .profile {extract /(\.profile)/ as found}
{if found is not ".profile"}
  ls
{end}
```

5.2.2 if ... else ... end

You can also add an else condition:

```
ls -l .profile {extract /(\.profile)/ as found}
{if found is not ".profile"}
```

```

    ls
  {else}
    touch .profile
  {end}

```

5.2.3 if ... else if ...

You can perform multiple matches using else if:

```

    ls -l .profile {extract /(*.profile)$/ as found}
  {if found is ".profile"}
    ls
  {else if found matches /my_profile/}
    ls -l p*
  {else}
    touch .profile
  {end}

```

5.3 Loops

You can execute commands multiple times using the “loop” statement. The following *Exscript* executes the “ls” command three times:

```

{number = 0}
{loop until number is 3}
  {number = number + 1}
  ls $directory
{end}

```

Similarly, the while statement may be used. The following script is equivalent:

```

{number = 0}
{loop while number is not 3}
  {number = number + 1}
  ls $directory
{end}

```

Another alternative is using the “loop from ... to ...” syntax, which allows you to specify a range of integers:

```

# Implicit "counter" variable.
{loop from 1 to 3}
  ls $directory$counter
{end}

# Explicit variable name.
{loop from 1 to 3 as number}
  ls $directory$number
{end}

```

5.4 Loops And Lists

The following *Exscript* uses the `ls` command to show the content of a list of subdirectories:

```
ls -l {extract /^d.*\s(\S+)/ as directories}
{loop directories as directory}
  ls $directory
{end}
```

You can also walk through multiple lists at once, as long as they have the same number of items in it:

```
ls -l {extract /^(d\S+)\s.*\s(\S+)/ as modes, directories}
{loop modes, directories as mode, directory}
  echo Directory has the mode $mode
  ls $directory
{end}
```

List loops can also be combined with the `until` or `while` statement seen in the previous section:

```
ls -l {extract /^d.*\s(\S+)/ as directories}
{loop directories as directory until directory is "my_subdir"}
  ls $directory
{end}
```

5.5 Using Functions

Exscript provides builtin functions with the following syntax:

```
type.function(EXPRESSION [EXPRESSION ...])
```

For example, the following function instructs *Exscript* to wait for 10 seconds:

```
{sys.wait(10)}
```

The following functions are currently provided:

1. `connection.authenticate(user = [None], password = [None])`
2. `connection.authorize(password = [None])`
3. `connection.auto_authorize(password = [None])`
4. `connection.close()`
5. `connection.execline(data)`
6. `connection.exec(data)`
7. `connection.sendline(data)`
8. `connection.send(data, wait = None)`
9. `connection.set_error(error_re = None)`

10. `connection.set_prompt(prompt = None)`
11. `connection.set_timeout(timeout)`
12. `connection.wait_for(prompt)`
13. `crypt.otp(password, seed, seqs)`
14. `device.guess_os()`
15. `device.model(force = None)`
16. `device.os(force = None)`
17. `device.vendor(force = None)`
18. `file.chmod(filenamees, mode)`
19. `file.clear(filename)`
20. `file.exists(filename)`
21. `file.mkdir(dirnames, mode = None)`
22. `file.read(filename)`
23. `file.rm(filenamees)`
24. `file.write(filename, lines, mode = ['a'])`
25. `ipv4.in_network(prefixes, destination, default_mask = [24])`
26. `ipv4.mask2pfxlen(masks)`
27. `ipv4.mask(ips, mask)`
28. `ipv4.pfxlen2mask(pfxlen)`
29. `ipv4.pfxmask(ips, pfxlens)`
30. `ipv4.remote_ip(local_ips)`
31. `list.new()`
32. `list.unique(source)`
33. `string.replace(strings, source, dest)`
34. `sys.message(string)`
35. `sys.run(host, filename)`
36. `sys.tacacs_lock(user = [None])`
37. `sys.tacacs_unlock(user = [None])`
38. `sys.wait(seconds)`

5.6 Exiting A Script

5.6.1 fail “message”

The “fail” keyword may be used where a script should terminate immediately.

```
show something
{fail "Error: Failed!"}
show something else
```

In this script, the “show something else” line is never reached.

5.6.2 fail “message” if ...

It is also possible to fail only if a specific condition is met. The following snippet terminates only if a Cisco router does not have a POS interface:

```
show ip int brie {
  extract /^(POS)\S+/ as pos_interfaces
  fail "No POS-Interface found!" if "POS" not in pos_interfaces
}
```

5.7 Error Handling

Exscript attempts to detect errors, such as commands that are not understood by the remote host. By default, *Exscript* considers any response that includes one of the following strings to be an error:

```
invalid
incomplete
unrecognized
unknown command
[^\r\n]+ not found
```

If this default configuration does not suit your needs, you can override the default, setting it to any regular expression of your choice using the following function:

```
{connection.set_error(/[Ff]ailed/)}
```

Whenever such an error is detected, the currently running *Exscript* is cancelled on the current host. For example, when the following script is executed on a Cisco router, it will fail because there is no `ls` command:

```
ls -l
show ip int brief
```

The “show ip int brief” command is not executed, because an error is detected at “ls -l” at runtime. If you want to execute the command regardless, you can wrap the “ls” command in a “try” block:

```
{try}ls -l{end}
show ip int brief
```

You can add as many commands as you like in between a try block. For example, the following will also work:

```
{try}
  ls -l
  df
  show running-config
{end}
show ip int brief
```

6 Trouble Shooting

6.1 Common Pitfalls

Generally, the following kinds of errors that may happen at runtime:

1. **A script deadlocks.** In other words, *Exscript* sends no further commands even though the remote host is already waiting for a command. This generally happens when a prompt is not recognized.
2. **A script executes a command before the remote host is ready.** This happens when a prompt was detected where none was really included in the response.
3. **A script terminates before executing all commands.** This happens when two (or more) prompts were detected where only one was expected.

The following sections explain when these problems may happen and how to fix them.

6.2 Deadlocks

Exscript tries to automatically detect a prompt, so generally you should not have to worry about prompt recognition. The following prompt types are supported:

```
[sam123@home ~]$
sam@knip:~/Code/exscript$
sam@MyHost-X123$
MyHost-ABC-CDE123$
MyHost-A1$
MyHost-A1(config)$
FA/0/1/2/3$
FA/0/1/2/3(config)$
admin@s-x-a6.a.bc.de.fg:/$
```

Note: The trailing “\$” may also be any of the following characters: “\$#;%”

However, in some rare cases, a remote host may have a prompt that *Exscript* can not recognize. Similarly, in some scripts you might want to execute a special command that triggers a response that does not include a prompt *Exscript* can recognize.

In both cases, the solution includes defining the prompt manually, such that *Exscript* knows when the remote host is ready. For example, consider the following script:

1. show ip int brief
2. write memory
3. {enter}
4. show configuration

Say that after executing line 2 of this script, the remote host asks for a confirmation, saying something like this:

```
Are you sure you want to overwrite the configuration? [confirm]
```

Because this answer does not contain a standard prompt, *Exscript* can not recognize it. We have a deadlock. To fix this, we must tell *Exscript* that a non-standard prompt should be expected. The following change fixes the script:

1. show ip int brief
2. {connection.set_prompt(/\[confirm\]/)}
3. write memory
4. {connection.set_prompt()}
5. {enter}
6. show configuration

The second line tells *Exscript* to wait for “[confirm]” after executing the following commands. Because of that, when the write memory command was executed in line 3, the script does not deadlock (because the remote host’s response includes “[confirm]”). In line 4, the prompt is reset to it’s original value. This must be done, because otherwise the script would wait for another “[confirm]” after executing line 5 and line 6.

6.3 A Command Is Sent Too Soon

This happens when a prompt was incorrectly detected in the response of a remote host. For example, consider using the following script:

```
show interface descriptions{extract /^(S+\d)/ as interfaces}
show diag summary
```

Using this script, the following conversation may take place:

1. router> show interface descriptions
2. Interface Status Protocol Description
3. Lo0 up up Description of my router>
4. PO0/0 admin down down
5. Serial1/0 up up MyWAN link
6. router>

Note that line 3 happens to contain the string “Router¿”, which looks like a prompt when it really is just a description. So after receiving the “¿” character in line 3, *Exscript* believes that the router is asking for the next command to be sent. So it immediately sends the next command (“show diag summary”) to the router, even that the next prompt was not yet received.

Note that this type of error may not immediately show, because the router may actually accept the command even though it was sent before a prompt was sent. It will lead to an offset however, and may lead to errors when trying to capture the response. It may also lead to the script terminating

too early.

To fix this, make sure that the conversation with the remote host does not include any strings that are incorrectly recognized as prompts. You can do this by using the “`connection.set_prompt(...)`” function as explained in the sections above.

6.4 The Connection Is Closed Too Soon

This is essentially the same problem as explained under “A Command Is Sent Too Soon”. Whenever a prompt is (correctly or incorrectly) detected, the next command is sent to the remote host. If all commands were already executed and the next prompt is received (i.e. the end of the script was reached), the connection is closed.

To fix this, make sure that the conversation with the remote host does not include any strings that are incorrectly recognized as prompts. You can do this by using the “`connection.set_prompt(...)`” function as explained in the sections above.