snom 4S Proxy Scripting Interface

snom
The SIP Experts

snom 4S
SIP Proxy/Registrar
Version 2.39

snom 4S Registrar Proxy Version 2.39 Scripting

Legal Disclaimer

Table of Contents

Introduction 7

Scripting Background 7
When You Should Consider Scripting 8
Loading a script 9

Script Structure 11

Comments 11
Quotes 11
Upper and Lower Case 12

Statements 13

if 13
foreach 14
return 15

Variables 17

Names 17
Scope 17
Types 17

Operators 19

Arithmetic Operators 19
Logical Operators 19
Comparisons 19
Other Operators 20
Operator Precedence 20

Functions 21

Definition 21
Calling 22

Built-in Functions 23

String Functions 23

length(string) 23
substr(string, start, length) 23
leftstr(string, length), rightstr(string, length) 23
get_field(string, index, delimiter) 24
match(string, pattern) 24
ere_match(string, pattern) 24
parameter_name(pair), parameter_value(pair) 25
get_time(), get_date(), get_day() 25
Parsing Functions 25
parse_url(string) 25
parse_user(string), parse_host(string), parse_port(string), parse_scheme(string) 25
parse_name(string) 26
parse_cparm(string), parse_uparm(string) 26
parse_header(string) 26
SIP functions 26
create_url(user, host, parameter, header) 26
create_number(number) 27
get_header(field1, field2) 27
conv_enum(name) 28
in_domain(name) 28
Registration Related Functions 28
known(account) 28
registered(account) 28
resolvable_contact() 29
register(account) 29
Proxy Related Functions 29
proxy_dest(destination, delay) 29
proxy_user(account, delay) 29
reject_request(code, additional) 30
send_ringing(code, additional) 30
get_contacts() 31
num_branches() 31
exec_dialplan(from, to, uri, pattern) 31
Other Functions 31
create_message(from, to, method, event, type, filename, url) 31
store_message(account, message) 32
store_file(name, content) 32
load_file(name) 32
store_userfile(account, name, content) 32
load_userfile(account, name) 33

create_user(account, username, password) ... *33*
delete_user(account) ... *33*
set_parameter(account, parameter, value) ... *33*
get_parameter(account, parameter) ... *33*
log(level, message) ... *34*

Callbacks ... 35

on_request ... 35
Condition for Calling ... *35*
Default Implementation ... *36*
on_response ... 38
Condition for Calling ... *38*
Default Implementation ... *38*
on_register ... 39
Condition for Calling ... *39*
Default Implementation ... *39*
challenge ... 40
Condition for Calling ... *40*
Default Implementation ... *40*
get_user ... 41
Condition for Calling ... *41*
Default Implementation ... *41*
on_new_user ... 42
Condition for Calling ... *42*
Default Implementation ... *42*
user_directory ... 43
Condition for Calling ... *43*
Default Implementation ... *43*
require_billing ... 44
Condition for Calling ... *44*
Default Implementation ... *44*
on_denial ... 45
Condition for Calling ... *45*
Default Implementation ... *45*
on_unroutable ... 45
Condition for Calling ... *45*
Default Implementation ... *45*
on_post ... 46
Condition for Calling ... *46*
Default Implementation ... *46*

Introduction

Scripting Background

Using the script language of the snom 4S proxy, you can customize the behaviour beyond the settings which are available on the web interface.

There have been many proposals for defining the behaviour of a SIP proxy. Proposals include generic approaches and SIP specific approaches. Examples are CORBA, CPL, CTI, JAIN, Java Enhanced SIP (JES), Java Servlet API, JINI, JTAPI, OSA-PARLAY, SIP CGI-BIN, TAPI, TINA, TOPS, and VoiceXML. Looking at the http server history, we can learn that simple scripting languages like perl or PHP found a broad acceptance because of its simplicity, flexibility, stability and performance. That was our motivation to integrate a PHP-like scripting language, even if it is not a powerful as PHP4 is today.

Apache offers handling of domains and URLs. While the concept of domains has been adapted in the proxy, it does not make sense to define a script for every URL the proxy can handle. Therefore, we decided to bind the script to the domain only.

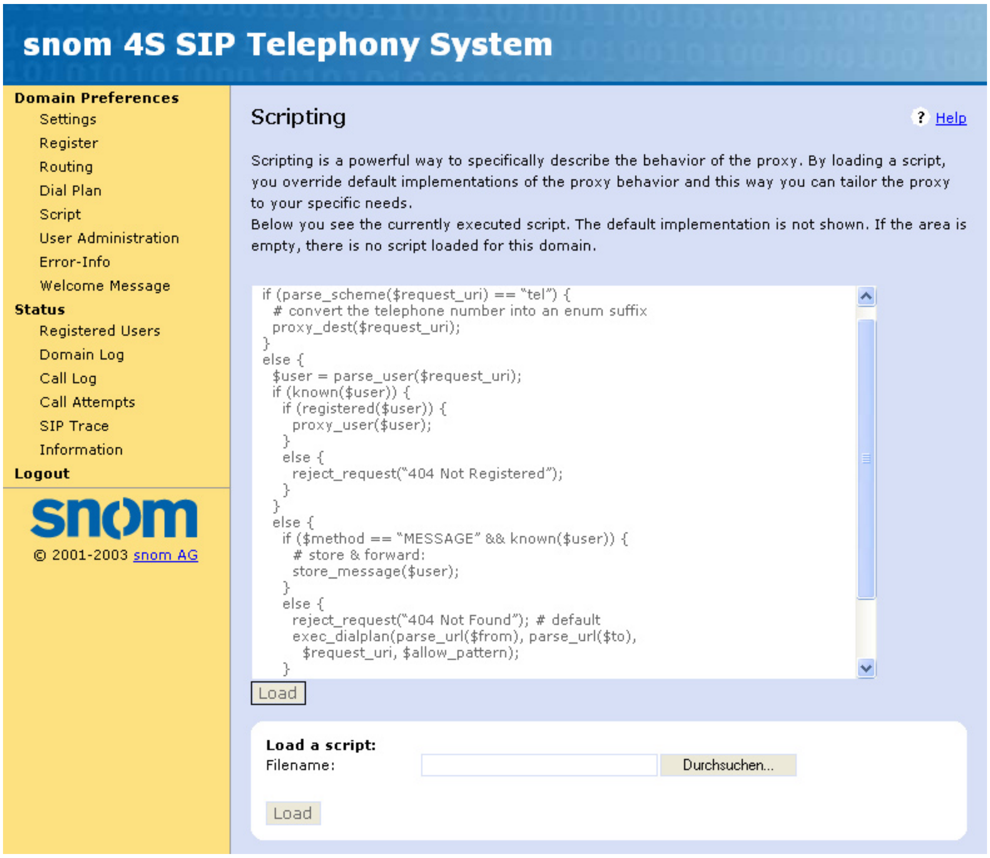
The proxy loads a default script during the start process and defines a number of functions. These functions are used unless overwritten by domain specific functions. The function overriding is done within the scope of a domain so that you can specify a completely different behaviour for different domains. This is helpful if you are operating a number of separate domains (e.g. as ITSP) or if you want to separate a productive system from a test system.

The layered architecture is also used for defining variables. Global variables can be overwritten by domain specific variables in the same way that functions are overwritten. An additional layer for overriding variables is the request layer, which exists within the domain context.

When You Should Consider Scripting

In a small office, you usually don't need to write your own script. The web interface covers most of the cases found in a typical environment.

As an operator, we recommend taking a look at scripting. In most cases, sooner or later your customers will request features that cannot be set up with the web interface or get too complicated (for example, the dial plan get very long).



Scripting

Scripting is a powerful way to specifically describe the behavior of the proxy. By loading a script, you override default implementations of the proxy behavior and this way you can tailor the proxy to your specific needs.
Below you see the currently executed script. The default implementation is not shown. If the area is empty, there is no script loaded for this domain.

```
if (parse_scheme($request_uri) == "tel") {
  # convert the telephone number into an enum suffix
  proxy_dest($request_uri);
}
else {
  $user = parse_user($request_uri);
  if (known($user)) {
    if (registered($user)) {
      proxy_user($user);
    }
    else {
      reject_request("404 Not Registered");
    }
  }
  else {
    if ($method == "MESSAGE" && known($user)) {
      # store & forward:
      store_message($user);
    }
    else {
      reject_request("404 Not Found"); # default
      exec_dialplan(parse_url($from), parse_url($to),
        $request_uri, $allow_pattern);
    }
```

Load a script:
Filename:

Loading a script

There are two ways to load a script. The first way to do this is to edit the script right on the scripting web page in the domain administrator mode. If you press the load button, the script gets directly loaded into the domain.

The second way is to upload a file into the proxy. To do this, select the scripting file with the file selection box and push the load button of the scripting interface.

If you load the script via this interface, you don‘t need to restart the proxy in order to make this script effective.

To store the currently loaded script, click on the link at the bottom of the scripting web page. This file will not include the default functions that might be running in this domain.

Script Structure

A script is made up by a list of statements and function definitions.

The script does not respect line breaks (like some BASIC implementations do). That means you can insert line breaks and indentation as you like.

There are two exceptions to this rule. The first are comments and the second are quoted text.

Comments

As in most other programming languages, you can use comments to make your code more readable. Comments start with a hash symbol ("#") and go until the end of the line. The proxy removes all comments during the reading of the script so that there is no performance drawback by comments.

Quotes

Quoted text begins with the quotation sign '"' and lasts until a terminating quotation sign is being found.

It is not allowed to use a line break within a quote. All quotations must be terminated within the same line.

To get a newline character, you can use the "\n" known from C programming language. "\r" inserts a carriage return. To get a backslash, use "\\". To get a quote character, use '\"'.

To get a number constant, you can omit the quotes around the number.

Upper and Lower Case

The script language is not case sensitive in principle. That means you can write "IF" or "if". Variable names are also not case sensitive. This is a major difference to C programming language and other programming language. Please keep this in mind when writing code.

The case sensitivity of the operator is discussed in the chapter which explains the operators.

Statements

Statements may occur inside or outside of functions and are executed sequentially. The behaviour resembles the execution of C- or PHP-code.

if

The if statement is used to execute code depending on conditions. The syntax for the if-statement is:

if (condition1) { body1 }
else if (condition2) { body2 }
else { body3 }

If the value of the expression1 is "true", the proxy executes body1 and does not execute body2 or any other body of this if-statement. It also doe not evaluate and other condition statements. This is important in cases where the other conditions have side effects.

If condition1 is not "true", the proxy checks the next condition it finds in an if-else part. The number of if-else parts if unlimited and may be zero (no if-else part). If there is a match, it executes the body for that if-else part.

If no match was found and a else part is present, it executes the body of the else part.

The comparison for conditions is done case-insensitive, so that "TRUE" also leads to execution of a body.

Note: In contrast to C, the body must be enclosed in brackets ("{" and "}"), even if you have only one statement inside of the brackets.

Example:

```
if ($a == "17") {
  log("Found a\n");
}
```

```
else if ($b == "17") {
  log("Found b\n");
}
else if ($c == "17") {
  log("Found c\n");
}
else {
  log("Not found\n");
}
```

foreach

The proxy does not support looping like the for in C. This was done to avoid endless loops. Endless loops are catastrophic for the proxy as it sill stop service in such a case.

The alternative of looping is going through lists. Lists always have a defined length and there is no danger that the proxy will end up in an endless loop.

The foreach command has the following syntax:

foreach(variable, list) { body }

The proxy will assign every element of the list to the provided variable and execute the body. The scope of the variable of the current context, that means the value of the variable after the foreach statement will be the value of the last list element (if there was at least one list element) or the variable will be unchanged (if there was no list element).

The list is an expression that is evaluated before the execution of the foreach statement starts. The result of the execution is interpreted as string which contains the list elements separated by space.

Example:

```
foreach($i, "1 2 3 4") { log("This is " . $i . "\n"); }
```

This example will write the log messages "This is 1\n", "This is 2\n", "This is 3\n" and "This is 4\n" into the log file.

return

The return statement has the syntax

return expression;

return leaves the current function and returns as value of the function the value of the expression. If there is no expression, the return statement returns the empty string.

Variables

Names

Variable names start with the dollar sign "$" and is followed by character ‘a’-‘z’, ‘A’-‘Z’ or ‘_’. The next character may also include ‘0’-‘9’.

Examples: $abc (same as $ABC), $a0, $a_0, $_1

Invalid examples: $0, $$, $a#

Scope

The proxy keeps variables in different scopes. This is necessary to separate domains from requests and allow function arguments.

When a domain is created, it inherits to global variables of the proxy. If a request within a domain is created, the request object inherits the variables of the domain.

If a function is called (either within a request context or within a domain context), the arguments to the function are stored on a call stack and restored when the function returns. If the variable did not exist before the call, it is deleted after the function returns. If a variable is modified or created within a function, it has the context of the underlying request (if there is such a context) or the context of the domain (if there is no request).

Types

The proxy scripting language does not support the concept of type. All variables are simply treated as strings. In the case that an operator expects a number, it first determines the value of that string, performs the operation and then converts the result back into a string. This is also the case for boolean operators as seen in the "if" statement.

There is no length restriction to strings. Short strings are handled efficiently; strings with a length of more than 64 kB are allocated and freed from memory on demand. This can significantly slow down the performance of the proxy and (depending on the memory management of the operating system) take away resources.

Operators

Arithmetic Operators

Arithmetic operators are +, -, *, /, %. They take the numerical value of their left and right hand side and perform the appropriate operation. % performs modulo. All operations are done on integer numbers, floating point is not supported.

Logical Operators

Logical operators include "&&" (and), "||" (or) and "!" (not).

The "&&" operator evaluates the right hand side only if the left hand side evaluated to "true". The "||" operator evaluates the right hand side only if the left hand side evaluated not to "true". This is important when the expression has a side effect. The "!" operator returns only "false" when the right hand side was "true".

Comparisons

Comparisons always return the value "true" or "false". The proxy implements the operators "==", "!=", ">", ">=", "<" and "<=".

The operators "==" and "!=" compare for equality and non-equality. The comparison is done non-case sensitive.

The relational comparison operators ">", ">=", "<", "<=" first check if both sides of the comparisons are integer numbers. If this is the case, they compare the value of the left and right hand side. Otherwise, they perform a non-case sensitive string comparison of the two sides (using the "strcmp" C-library function).

Other Operators

The comma operator "," just evaluates both left and right hand side and return the value of the right hand side.

The dot operator "." concatenates the left string value with the right hand string value. This is helpful when messages should be created that contain variables as there is no "printf" function like in the C-library.

Operator Precedence

The following table shows the precedence of the available operators (operators with the lowest precedence are executed first):

9:	,
8:	=
7:	||
6:	&&
5:	== !=
4:	< <= > >=
3:	+ - .
2:	* / %
1:	!

Functions

Functions take a number of arguments and return a string value. You can define functions in the domain script as you wish.

Function names are similar to variables with the difference that function names do not contain the leading dollar sign. Function names are like most of the proxy functionality non case-sensitive.

The built-in functions are described later in this document.

Definition

Functions are defines on the top level of a script. That means it's not possible to define nested functions (functions within a function).

The syntax for a function definition looks like this:

Func (arg1, arg2, arg3) { body }

The name of the function must be followed by brackets enclosing a list of arguments (separated by comma). If the list is empty it still needs the brackets. The arguments must be variable names including the dollar sign and it is not allowed to use expressions in the argument list.

Tip: By providing arguments that are not initialized when the function is being called, you can set up local variables for the function. This trick is known from other scripting languages like gawk.

The sequence in which you define your functions do not matter. The functions are set up when the script is loaded. When you actually execute a function, the definition is already available, even if it appears later in the script.

Calling

Once you have defined your function, you can use it in any expression. In the simplest case, a statement consists of a single function call, which looks like a procedure call.

To avoid endless looping when a function calls itself (directly or indirectly through recursion), the number of nested function calls is limited. The default value for this limit is 50. If that limit has been reached, every function just returns an empty string, so that the processing of the current proxy action stops. In such a case, the proxy will write a log message. This will allow the operator to find these conditions.

Built-in Functions

String Functions

String functions are helpful to manipulate parts of strings and to find out if a string contains a certain patterns.

length(string)

To get the length of a string, call length with the string as argument. The result will be an integer number.

Example: length("1234") will return "4".

substr(string, start, length)

To get a part of a string, you can use the substr function. It takes two or three arguments.

The first argument is the string that should be used. The second argument is the position in the string. It refers to the character counted from the left side of the string starting at position 0. If the third argument is present, it indicated how many characters should be taken from the string. If the third argument is absent, the function will return the rest of the string starting at the position.

Example: substr("abcde", 2, 2) returns "cd", substr("abcde", 2) returns "cde".

leftstr(string, length), rightstr(string, length)

The leftstr takes two arguments. The first argument is the string which should be operated on. The second argument indicated how many characters should be returned.

In the case of leftstr, the function returns the string starting from the left side, in the case of rightstr starting from the right side.

get_field(string, index, delimiter)

The get_field function extracts a field from a string. It takes three arguments. The first argument contains the string from which the field should be extracted. The second argument indicates the field number, starting at 0. The third argument is optional and indicates the field separator characters, which default to white space.

Examples: get_field("a b c", 1) returns "b", get_field("a b, c d", 1, ",") returns " c d".

match(string, pattern)

match is used for pattern matching. The first argument is the underlying string, the second argument the pattern. If there is a match, the function returns "true".

The "?" matches any character, "$" matches only digits. "*" matches any number of characters, "%" any number of digits. "~" matches the name of the domain (if the script is evaluated in domain context). "[a-z]" matches a character range. See the description for the dial plan of the proxy for details.

ere_match(string, pattern)

ere_match is also used for pattern matching. In contrast to match, it uses the "extended regular expression" matching known from NAPTR DNS resolution. However, it can also be helpful in other cases when pattern need to be replaced (that was the reason why it was chosen in NAPTR). For more information, refer to RFC 2915.

ere_match take as parameters the input string and the extended regular expression with replacement (separated by any separator symbol as described in RFC2915).

Example: ere_match("urn:cid:39CB83F7.A8450130@fake.gatech.edu", "/urn:cid:.+@([^\\.]+\\.)(.*)$/\\2/i") will return "gatech.edu".

parameter_name(pair), parameter_value(pair)

These functions retrieve the name and the value of the parameter value provided as argument. These functions are helpful when URL or contact parameters must be checked.

Example: parameter_name("bla=123") returns "bla", parameter_value("bla=123") returns "123".

get_time(), get_date(), get_day()

These functions return the current time, the current date or the current day. All date related information are relative to GMT.

The get_time function returns a string in HH:MM format, for example 14:54. The get_date function returns the current date in M.D.Y format, for example 5.28.2003. The get_day function returns the day of the week in English three letter format, e.g. Sun.

Parsing Functions

The paring functions give you easy access to URL components and to parameters and names.

parse_url(string)

This function extracts the URL of a string.

Example: parse_url("Fred Feuerstein <sip:ff@stoneage.org;parm=123>") returns "sip:ff@stoneage.org;parm=123", parse_url("sip:ff@stoneage.org;parm=123") returns "sip:ff@stoneage.org".

parse_user(string), parse_host(string), parse_port(string), parse_scheme(string)

These functions extract parts of the URL. If the URL is enclosed in "<" and ">" and part of a display-name representation, the pure URL is retrieved first. See the specification in RFC3261 for details on URL.

Example: parse_user("<sip:abc@stoneage.org>") returns "abc", parse_user("sip:host:5068") returns "", parse_scheme("tel:1234") returns "tel", parse_port("sip:abc@stoneage.org") returns "",parse_port("sip:abc@stoneage.org:5052") returns "5052", parse_host("sip:abc@stoneage.org:5061") returns "stoneage.org".

parse_name(string)

This function extracts the display-name.

Examples: parse_name("Fred \\\"F.\\\" Feuerstein <sip:ff@stoneage.org>" returns "Fred \"F.\" Feuerstein".

parse_cparm(string), parse_uparm(string)

This function extracts the contact parameters (parse_cparm) or the URL parameters (parse_uparm) of a string.

The contact parameters are not part of the URL and are only present if the url is protected by the "<" and ">" symbols. If these symbols are not present, the parameters are assumed to be part of the URL.

The result is a list of space-separated elements representing the parameters.

Example: parse_cparm("F Feuer <sip:ff@bla.com>;tag=123;parm=456") returns "tag=123 parm=456".

parse_header(string)

This function extracts the contact header arguments of a string. The header arguments are behind the "?" and separated by "&" characters (see http URL). The result is also a list of space separated elements.

Example: parse_header ("sip:ff@bla.com?p1=1&p2=2") returns "p1=1 p2=2".

SIP functions

create_url(user, host, parameter, header)

create_url is a helper function that creates a URL from its arguments. It takes four arguments. The first argument it the user name. If the URL should not contain a user name, this argument may be left empty. The second argument contains the hostname and the port, separated by a colon. If the port should be empty, the port may be omitted and the colon is not necessary. If this argument is left blank, it is replaced with the current domain name.

The third argument contains the list of parameters, each separated by space. The fourth argument contains a list of header parameters, also separated by space.

The create_url function always creates a sip URL.

Example: create_url("123", "domain.com:5062", "transport=udp line=1") returns "sip:123@domain.com:5062;transport=udp;line=1".

create_number(number)

Users are sometimes a little bit sloppy entering SIP URL. For example, they expect that a simple username without domain automatically gets converted into a complete URL. This function takes one argument that is converted into a complete URL.

If the argument is already a complete URL, this argument is returned. If the argument does not contain a "@" symbol, this function converts the argument into a URL that has the user part set to the argument and the domain to the current domain. Otherwise, it merely completes the URL with the sip scheme and returns that URL.

Examples (in the domain "domain.com"): create_number("sip: bla@abc.com") returns "sip:bla@abc.com"), create_number("123") returns "sip:123@domain.com" and create_number("123@ff.com") returns "sip:123@ff.com".

get_header(field1, field2)

get_field retrieves a SIP message header of the request of the current open request. This function is helpful when the script needs information directly from the SIP request. Examples include User-Agent or proprietary information.

The function takes one or two arguments. The first name is the normal name of the header, the optional second argument is the short name of the header.

Example: get_header ("From", "f") returns "Fred Feuerstein <ff@stoneage.org>;tag=123456".

conv_enum(name)

RFC2916 defines an algorithm that translates a telephone number in the tel URL style into a DNS name. conv_enum executes that algorithm. It takes the number as first argument (without the resource identifier) and optionally the domain which is being searched. This option defaults to "e164.arpa".

Example: conv_enum("+49-30-39833-0") returns "0.3.3.8.9.3.0 .3.9.4.e164.arpa".

in_domain(name)

Sometimes it is important to know if a URL belongs to the domain where a request is being processed in (for example, to check if the source also belongs to the domain). For this purpose, the function in_domain takes as argument a URL or a contact (as from the From-header) and returns true if that URL or contact belongs to the current domain.

Registration Related Functions

known(account)

The function known determines if the account name provided as argument has an assigned username. This is the case if the account has been set up on the proxy. It is not necessary that the password is set.

Example: known("1234") returns "true"

registered(account)

The function registered is similar to the known function. However, in this case the proxy checks if at least one registration exists for the provided account name. That means that it will be possible to proxy a request to this user.

Example: registered("1234") returns "false"

resolvable_contact()

The function resolvable_contact does not take any parameters.

It simply checks if the top Via header of the received request matches the IP address where the request came from. Typically, if a client is behind NAT, this is not the case. If the proxy itself is operating on a NAT address, this function always returns true.

The proxy does not perform DNS resolution to check if an address is a private address. The primary goal with this function is to avoid stupid registration attempts from behind NAT so that customers receive a notice when their network is not setup correctly for handling NAT.

register(account)

The function register takes as parameter an account name. It registers the contacts provided in the current request with the current account and sends a success response back to the user agent client.

See the explanations on registrations on details on Path and registration duration limitation.

Proxy Related Functions

proxy_dest(destination, delay)

This is one of the core functions of the proxy. It forks the current request to the location provided as the first argument. If present and the request is an INVITE request, the second argument delays the fork. The delay is measures in seconds.

Forking to a destination can be done without a domain context.

proxy_user(account, delay)

The proxy_user uses the database of registered users for determining the destination of the forking process. The user account is provided as first argument.

As with proxy_dest, the second argument delays the forking process. However, the delay is combined with the registration probability, so that users with a probability will receive the request immediately; users with a probability of zero will receive the request after the delay. Users having a probability between these extremes will receive the request with a delay that scales linear between the extremes (sequential forking). If you use a negative delay value, the proxy does not use the probability value of the registration and instead initiates the request exactly after the absolute value of the delay.

If no user is registered with this account, this function sends a 404 response to the request. If there are no other requests forked, this will be the end result of the request.

reject_request(code, additional)

Sometimes it is helpful to explicitly add a reject code to a request. For instance, when a user cannot be found, the proxy might want to signal "404 Not Found". The reject_request takes as argument the reject code. The optional second argument may contain additional headers that are inserted into the response. The additional parameters **must** be terminated with a CRLF pair.

Example: reject_request("404 Not Found", "Error-Information: <sip:notfound@media.company.com>\r\n")

send_ringing(code, additional)

send_ringing is similar to reject_request and it is called in the same way. Instead of adding a final response to the open request, it immediately sends out a provisional response on behalf of the proxy. This is helpful when the calling party should have the impression that the other side is ringing. This command makes especially sense when used during redirection with a 181 response code.

The send_ringing command may also include additional headers like the reject_request command.

get_contacts()

The get_contacts function returns a list (separated by spaces) containing the Contact URL of the current response. This is helpful when the proxy should take care about redirecting calls in early media state.

num_branches()

This function returns the number of branches open on the current request. This is helpful in situations when the proxy needs to know if a request is pending to a request.

For example, when you try different low cost gateways to terminate a call, you want to redirect the call to a expensive gateway only if all of the low cost gateways did not pick up the call.

exec_dialplan(from, to, uri, pattern)

The exec_dialplan function exists to simplify the usage of the web-based dial plan. Normally, when programming scripts for the proxy, this function is not needed. However, in some cases it may be easier and more convenient to use the dial plan in addition to the script language.

The exec_dialplan function takes four arguments. The first argument is the actual destination part, the second argument the actual pattern part and the third argument the actual URI of the request. The fourth argument is the dial plan in the format generated by the web interface.

Normally, the proxy calls the function like this: exec_dialplan(parse_url($from), parse_url($to), $request_uri, $allow_pattern).

Other Functions

create_message(from, to, method, event, type, filename, url)

The create_message function generates a request string that can be stored in the store-and-forward buffer of a user account. This function is helpful for generating welcomes messages and other information messages.

The function takes seven arguments. The first argument identifies the From-Header, the second the To-Header of the request. Argument number three indicates the method (typically "MESSAGE" or "NOTIFY"), the fourth argument the event type of the request (this argument can be left empty in some cases). The argument number five indicates the Content-Type of the request. The sixth argument indicates the file that contains the actual attachment, the seventh argument indicates the request_uri. This uri can be left blank in most cases as it will be overwritten by the sending procedure.

store_message(account, message)

To put a message into the store-and-forward buffer of an account, you may use the store_message function.

It takes as arguments the account and the message string, which is typically generated by the create_message function.

store_file(name, content)

This function writes the content into the file with the filename "name". The filename is relative to the current working directory. Absolute filenames and filenames using the ".." are not allowed. This is to avoid accidental writing into other domains files.

load_file(name)

This function returns the content of the file with the filename "name". The filename is relative to the current working directory. Absolute filenames and filenames using the ".." are not allowed. This is to avoid peeking into other domains files.

store_userfile(account, name, content)

This function writes the content into the file with the filename "name". The filename is relative to the directory of the account. Absolute filenames and filenames using the ".." are not allowed.

load_userfile(account, name)

This function returns the content of the file with the filename "name". The filename is relative to the directory of the account. Absolute filenames and filenames using the ".." are not allowed.

create_user(account, username, password)

The proxy does not create a user by default; it needs an explicit call to do so. The create_user takes as argument the name of the account which will contain the user, the name for challenging and the password. Both the name and the account must be provided, the password is optional.

delete_user(account)

This function deletes a account as if it has been removed from the web interface.

set_parameter(account, parameter, value)

Accounts may have parameters that are stored permanently. The set_parameter takes three arguments. The first argument is the name of the account, the second parameter the name of the variable and the third parameter the value that should be assigned to the variable.

The names "user" (user name), "pass" (password), "single" (single registration) have a special meaning and overwrite their predefined value. The name "domain" is reserved and cannot be overwritten.

Example: set_parameter($user, "trial_calls", 3);

get_parameter(account, parameter)

To read out a setting the get_parameter function may be used. It reads out both the predefined variables as well as the custom variables.

Example: if (get_parameter($user, "trial_calls") > 1) { ... }

log(level, message)

Log explicitly writes a message into the log file. It takes an unlimited number of arguments, but it must have at least two arguments. The first argument is evaluated to the log level. The other arguments are concatenated and printed to the log file if the log level is high enough.

Example: log(5, "This is a log\n")

Callbacks

The proxy scripting is primary determined by overriding the default implementation of callbacks.

Callbacks run either under a domain context or under a request context. Requests have their own scope, that means variables created in this scope remain inside the request and do not affect the domain variables. The following requests run under domain context: on_post, user_directory and on_new_user. All other are executed under request context.

on_request

Condition for Calling

Whenever the proxy receives a new request, it sets up an object representing that request and the associated responses, timeouts etc.

The request object has its own scope with variables. This objects receives a copy of the variables of the domain context where the request in being processed in. Additionally, some request specific variables are set: $method is set to the method of the request (e.g. INVITE, NOTIFY, etc.), $request_uri is set to the request URI of the request (this is the word after the method), $from is set to the value of the From header, including display-name and tags, $to is set to the value of the To header, similar to the $from variable.

After the variables have been copied, the proxy calls the on_request function without any arguments.

Default Implementation

The default implementation of the on_request function implements the behaviour describes in the web based configuration.

First of all, the script checks if a ENUM number is being called. If this is the case, the proxy just forwards the call. The transport mechanism will take care about the tel-url to ENUM conversion and the resolution of the SIP URL for the telephone number.

If the scheme is not a tel URL, the proxy checks if the request is a message to a user which is offline. In this case, it stores the message in the user account for later delivery. Otherwise it determines the user which is being called by looking at the request_uri. If that user is registered with the proxy, it forks the request to that user using the sequential forking delay. Otherwise, it calls the exec_dialplan function that takes care about the dial plan. In case the dial plan does not generate an error message, the proxy takes the 404 Not Found stored as default answer.

```
on_request() {
  if (in_domain($from) && get_parameter($account, “disabled”)) {
    # someone disabled this account
    reject_request(“404 Account Disabled”); # default
  }
  else if (parse_scheme($request_uri) == “tel”) {
    # convert the telephone number into an enum suffix
    proxy_dest($request_uri);
  }
  else {
    $user = parse_user($request_uri);
    if (known($user)) {
      if ($method == “INVITE”) {
        # check mailbox:
        $mb_target = get_parameter($user, “mb_target”);
        $mb_timeout = get_parameter($user, “mb_timeout”);
        $red_location = get_parameter($user, “red_location”);
          if ($mb_timeout == “”) { $mb_timeout = 20; } # 20 s is
default
        if ($red_location == “offline” || !registered($user)) { $mb_
timeout = 0; }
        if ($mb_target != “”) {
          if (registered($mb_target)) {
            proxy_user($mb_target, -$mb_timeout);
          }
          else {
            proxy_dest(create_number($mb_target), $mb_timeout);
          }
        }

        # check redirection:
        if ($red_location == "offline") {
          # no further redirections, mailbox already included
        }
        else if ($red_location == "home") {
          $red_target = get_parameter($user, "red_target");
          if ($red_target != "") {
            proxy_dest(create_number($red_target));
          }
        }
        else if ($red_location == "road") {
          $red_road = get_parameter($user, "red_road");
          if ($red_road != "") {
            proxy_dest(create_number($red_road));
          }
        }
        else { # Office, default
          # default action
          if (registered($user)) {
            proxy_user($user, $seqfork_delay);
          }
          else {
             reject_request("404 Not Registered"); # mb and red will
override this
          }
        }
      } # method == INVITE
      else if ($method == "MESSAGE" && !registered($user)) {
        # store & forward:
        store_message($user);
        reject_request("200 Message Delivered"); # default
      }
      else {
        # default action
        if (registered($user)) {
          proxy_user($user, $seqfork_delay);
        }
        else {
          reject_request("404 Not Registered");
        }
      } # other methods
    } # known user
    else {
      # no for a known user:
      reject_request(“404 Not Found”); # default
        exec_dialplan(parse_url($from), $request_uri, $request_uri,
$allow_pattern);
    }
  }
}
```

on_response

Condition for Calling

When a response is received by the proxy, it matches this response to an open request.

The proxy automatically determines the best response to a request. The function on_response does not need take care about this. The response is sent only of all call legs returned an error code or one of the call legs returned a success code.

The return code indicates weather the proxy should consider the response as an answer to the original request.

Default Implementation

The default implementation takes care about the redirection codes 300-399. The proxy initiates the sending of a 181 provisional response and then forks a new request for each of the contacts given in the redirect response. In this case, the response is not taken as a response code to the request.

```
on_response() {
  # handle redirect codes
  if ($code >= 300 && $code < 400) {
    send_ringing("181 Call Being Forwarded");
    foreach($dest, get_contacts()) {
      proxy_dest($dest);
    }
    return false;
  }
  else {
    return true; # this is a valid response
  }
}
```

on_register

Condition for Calling

When the proxy receives a REGISTER request, it would normally call on_request as the REGISTER is also a request. However, because normally REGISTER requests must be treated differently that other requests, the proxy calls the more special function on_request.

Normally, the function would check if the user exists in the database and then selectively register that user. However, on_request is called after the authorization checking, which is done in the challenge callback.

Default Implementation

The default implementation checks if the contact can be resolved or if that feature has been turned off. If the contact is accepted but the user account does not exist, the user account is created (which may trigger the on_new_user function). Otherwise, if the contact is not accepted, the request is rejected with a message that the contact is not ok.

```
on_register($user) {
  if (!$reject_nat_register || resolvable_contact()) {
    if (!known($user)) {
      log(2, "Create user " . $user . " without password\n");
      create_user($user, $user);
    }
    # if (!get_parameter($user, "disabled"))
    register($user);
  }
  else {
    reject_request("406 Bad Contact (NAT)");
  }
}
```

challenge

Condition for Calling

Challenging is a process in SIP which asks the user agent client to answer a question that can only be answered if the user agent has a secret password. The password is not transmitted and from the answer it is practically impossible to guess the password.

This challenging is used to protect destinations and authorize registrations in the standard web interface settings. However, you can also implement your own policy by overwriting the function challenge.

When a request arrives at the proxy, the proxy checks if the request contains credentials from a recently generated question (nonce). If that is the case the proxy will process the requests without further authorization checking.

If this is not the case, it calls the function "challenge" and if that function returns "true" it generates a new nonce and rejects the request with the code 407 (Proxy Authorization Required). If the user agent has the matching password, it will send another request which matches the generated response and then the request will be processed.

Default Implementation

The default implementation differentiates between REGISTER and other methods, just like the web interface does it.

For REGISTER requests, it asks for credentials only of general authorization has been turned on or the user us known.

For other requests, it goes through the list of protected destinations and checks if the pattern matches the request URI.

```
challenge() {
  if ($method == "REGISTER") {
     if ($force_authorization || known(parse_user($from))) { return
true; }
  }
  else {
    foreach($i, $auth_exception) {
      if (match($i, $request_uri)) { return true; }
    }
  }
  return false; #default
}
```

get_user

Condition for Calling

To be able to check the answer from a challenge, the proxy must determine which user account was used. In order to do this, the proxy calls the function get_user.

Most of the time, the user part of the URL in the From header will identify the user. However, you might want to specify another policy or add special cases (e.g. for gateways).

If the get_user function returns an empty string, the proxy will look for the user parameter in the challenge response. In this case the user agent may decide which account it uses to answer the question.

Default Implementation

The default implementation retrieves the user part of the URL in the From header.

```
get_user() {
  if (in_domain($from)) {
    return parse_user($from);
  }
  else {
    return;
  }
}
```

on_new_user

Condition for Calling

When a new user is created via the web interface or through explicit command in the scripting interface, the proxy calls a function "on_new_user". The function receives the user accounts as argument.

This function can be used nicely for welcome messages or other actions that need to be done during the set up of a user account. Examples include setting up of mailbox accounts, initializing billing parameters and so on.

The function does not have any influence on the fact that the new user account is being set up.

Default Implementation

The default implementation first checks if the settings for welcome messages are sufficient to generate a welcome message. It is assumed that this is the case if a welcome file has been specified and the method is either MESSAGE or an event type has been specified. An event type is always necessary when the NOTIFY message type has been selected.

Because the message type is set to "plain/text" by default, the user just has to specify the file name to trigger sending welcome messages.

If the default implementation determines that the message can be sent, it creates a new message with the parameters set up by the web interface and puts that message into the store and forward offer of the respective account.

```
on_new_user($user) {
  if ($welcome_file != "" &&
      ($welcome_event != "" || $welcome_method == "MESSAGE") &&
      $welcome_type != "") {
        store_message($user, create_message(create_url(), # From
          create_url($user), # To
          $welcome_method, # method
          $welcome_event, # event
          $welcome_type, # type
          $welcome_file)); # file
  }
}
```

user_directory

Condition for Calling

For a large number of registered users it is important to have a hash function that puts the respective user into a bin. This way the proxy can set up a layered file system structure where the hash results may serve as first tier index for finding the user account.

For example, by using hash function returning thousand bins it would be possible to set up approximately one million users without having to bother about performance of the file system. If every registration takes approximately 4 KB, registering 1 million users would require 40 GB storage, which can be easily set up with a normal PC.

For instance, if the hash function would return the first digit of a telephone number, the proxy with set up ten subdirectories each of them containing the telephone numbers starting with the first digit. That would increase the performance of the proxy almost by a factor ten already.

However, practically it is usually not so easy to find such a simple hash function. In cases where most of the users start for example with the number 2, that hash function would not be effective at all. Therefore, the proxy allows defining a customized hash function.

Good hash functions could be taking the last digit or the last two digits of the user accounts if they are equally distributed. More sophisticated-functions like a MD5 or check sum can be used in cases where an equally distributed hash function cannot be found easily.

Default Implementation

The default implementation returns the first character of the user name. This gives the proxy at least a minimum level of hierarchy.

```
user_directory($user) {
  return leftstr($user, 1);
}
```

require_billing

Condition for Calling

Before the proxy starts billing a request with RADIUS, it calls the require_billing function. This is necessary because RADIUS assumes that before a call can start the RADIUS server from its access. If the function returns the value "true" and the radius parameters have been set up, the proxy will first initiate a RADIUS admission request before it starts forking requests by calling the on_request function.

If RADIUS admission request is being denied, the proxy will call a function on_denial. This function then can inform the user that the trend is not sufficient by returning an appropriate response.

Practically, not all requests require billing. Depending on the policy of the operator, instant messages might be delivered without any billing, while initial INVITE requests may only be forwarded if the respective user account has positive credit. It also depends on the policy of the operator if for instance presence messages are subject to billing. It is very hard to predict in beforehand which packets require_billing and which not. That is the reason why the proxy calls the require billing function to determine if it should do billing or not.

Requests which already contain a route are never subject to billing. These requests are usually part of an ongoing call which did not need any further billing checks. This approach does not introduce new security problems, because if user agents is able to preset a route without previous proxy interaction, but will also be able to bypass the proxy at all.

Default Implementation

The default implementation merely checks if the request is an INVITE request and if this is the case returns true. All other requests are assumed not to require billing.

```
require_billing() {
  return $method == "INVITE";
}
```

on_denial

Condition for Calling

When the RADIUS admission request has been denied, the proxy calls the on_denial function. This function will usually generate an error response explaining that the credit on the billing server has expired. It may also include additional information like error information that will make the user agents call the media server, which will explain what happened.

Default Implementation

The default implementation merely returns in an error code "403 Forbidden".

```
on_denial() {
  reject_request("403 Request Denied");
}
```

on_unroutable

Condition for Calling

When a SIP URL cannot be resolved via DNS, the proxy calls this callback. This includes DNS NAPTR, DNS SRV, DNS A and ENUM searches.

Default Implementation

The default implementation merely returns in an error code "404 Not Found".

```
on_unroutable() {
  reject_request("404 Not Found");
}
```

on_post

Condition for Calling

In many cases, operators want to control the proxy automatically. The web interface was designed for user interaction and the session-key management complicates the automatic access to the proxy. Because of this, the proxy supports an automatic access method.

To ensure basic security and identify the, domain, two arguments are mandatory: domain must be set to the domain name and pass t the password for that domain. All other arguments are passed as variables to the on_post function. The page name is hard coded to "post.htm".

The proxy will return a short html message that embeds the return string of the on_post function. This is to indicate a web interface user that this page is normally not available. But you can see the return code embedded in the answer between the H1 tags.

To address the interface, you may use the program "curl" or even directly connect to the http TCP port and request the URL. For example:

```
curl "http://url.org/post.htm?domain=abc.de&pass=abc&action=create_u
ser&username=theo&password=secret&account=theo"
```

(please note the quotes around the URL that avoid the shell from forking several commands) will return:

```
<HTML><HEAD>
<TITLE>snom proxy: Error</TITLE>
</HEAD><BODY>
<H1>Ok</H1>
Please ask your system administrator to check the log file.<P>
</BODY></HTML>
```

Default Implementation

The default implementation checks for the action variable and allows four basic functions: creating, deleting, disabling and enabling an account. From the reading above, the code should be self-explanatory.

```
on_post() {
  if ($action == "create_user" && $account != "" && $username != "")
{
    create_user($account, $username, $password);
    return "Ok";
  }
  else if ($action == "delete_user" && $account != "") {
    delete_user($account);
    return "Ok";
  }
  else if ($action == "disable_user" && $account != "") {
    set_parameter($account, "disabled", "true");
    return "Ok";
  }
  else if ($action == "enable_user" && $account != "") {
    set_parameter($account, "disabled", "false");
    return "Ok";
  }
  else {
    return "Unhandled Request";
  }
}
```

Europe & ROW:

snom technology Aktiengesellschaft
Pascalstr. 10B, 10587 Berlin, Germany
Phone: +49 (30) 39833-0
mailto:info@snom.de
http://www.snom.com
sip:info@snom.com

India and SAARC:

snom technology (India) Pvt Ltd.
No. 417, International Trade Tower
Nehru Place, New Delhi-110019
Phone: +91 11 26234097
Fax: +91 11 26234079
http://www.snomindia.com
mailto:info@snomindia.com
sip:india@snom.com

USA and Americas:

snom USA Representation
ABP International, Inc.
1203 Crestside Dr.
Coppell, Texas 75019, USA
Phone: +1-972-831-0280
sip:usa@snom.com
mailto:usa@snom.de

snom
The SIP Experts