Python Script Support for ESM

In This Chapter

This section describes the Python script support for Enhanced Subscriber Management (ESM).

The following topics are included:

Python Script Support for ESM

In order to provide programmable flexibility in ESM applications, the SR OS provides the following features with Python script support:

- sub-ident-policy
- radius-script-policy
- python-policy

Python in SR-OS Overview

The SR-OS python script support is based on Python version 2.4.2. Python has a set of language features (such as functions, lists and dictionaries) and a very large set of packages which provide most of the Python functionality. By keeping the language features intact and drastically reducing the number of packages available, the operator is provided with a flexible, although small, scripting language.

The only feature removed from the Python language is unicode support. The only packages provided to the operator are:

- **alc** — The SR OS-provided packages provide access to various ESM objects such as DHCPv4, DHCPv6 or RADIUS packets.
- **binascii** — Common ASCII decoding like base64.
- **re** — Regular expression support.
- **struct** — Parses and manipulates binary strings.
- **md5** — MD5 message digest algorithm.

Python Changes

Some changes have been made to Python in order to run on an embedded system:

- No files or sockets can be opened from inside Python scripts.
- No system calls can be made from inside Python scripts nor is the posix package available.
- The maximum recursion depth is fixed to twenty.
- The total amount of dynamic memory available for Python itself and Python scripts is capped at 2MB.
- The size of the script source file must be less than 16KB.

Python Support in sub-ident-policy

A Python script can be configured in sub-ident-policy to return following ESM attributes:

- sub-id
- sla-profile name
- sub-profile name
- ancp-string

The system will run the Python script configured in the sub-ident-policy against the received DHCPv4 ACK message. This is used as the input of the script. Within the script, the user can set the value with the above ESM attributes.

The alc package contains a DHCP object, and has the following members (Table 21).

Table 21: DHCP Object Members

Name	Read	Write	Class
htype	X		integer
hlen	X		integer
hops	X		integer
flags	X		integer
ciaddr	X		integer
yiaddr	X		integer
siaddr	X		integer
giaddr	X		integer
chaddr	X		string
sname	X		string
file	X		string
options	X		TLV
sub_ident		X	string
sub_profile_string		X	string
sla_profile_string		X	string
ancp_string		X	string
app_profile_string		X	string
category_map_name		X	string
int_dest_id		X	string

The TLV type provides easy access to the value part of a stream of type-length-value variables, as is the case for the DHCP option field. In the example on page 1472, the circuit-ID is accessed as alc.dhcp.options[82][1].

Some DHCP servers do not echo the relay agent option (option 82) when the DHCP message was snooped instead of relayed. For the convenience of the operator, the relay agent option from the request message is returned when alc.dhcp.options[82] is called.

Configuration

As an example consider script us5.py on page 1475 which sets the sub_ident variable based on the circuit ID of three different DSLAMs:

```
import re
import alc
import struct
# ASAM DSLAM circuit ID comes in three flavours:
#      FENT   string         "TLV1: ATM:3/0:100.33"
#      GELT   octet-stream   0x01010000A0A0A0A0000000640022
#      GENT   string         "ASAM11 atm 1/1/01:100.35"
#
# Script sets output ('subscriber') to 'sub-vpi.vci', e.g.: 'sub-100.33'. circuitid =
str(alc.dhcp.options[82][1])
m = re.search(r'(\d+\.\d+)$', circuitid)
if m:
# FENT and GENT
alc.dhcp.sub_ident = "sub-" + m.group()
elif len(circuitid) >= 3:
# GELT
# Note: what byte order does GELT use for the VCI?
# Assume network byte (big endian) order for now. vpi = struct.unpack('B', circuitid[-3:-
2])[0]
vci = struct.unpack('>H', circuitid[-2:])[0]
alc.dhcp.sub_ident = "sub-%d.%d" % (vpi, vci)
```

Configure the url to this script in a sub-ident-policy as follows:

```
-----------------------------------------------
     sub-ident-policy "DSLAM" create
          description "Parse circuit IDs from different DSLAMs"
          primary

               script-url "ftp://xxx.xxx.xxx.xx/py/us5.py"
               no shutdown
          exit
     exit
-----------------------------------------------
```

And attach this sub-ident-policy to the sub-sla-mgmt from a SAP:

```
A:dut-A>config>service>vpls>sap# info
-----------------------------------------------
     dhcp
          description "client side"
          lease-populate 50

          no shutdown
     exit
     anti-spoof ip-mac
     sub-sla-mgmt
          sub-ident-policy "DSLAM"
          no shutdown
```

```
    exit
-----------------------------------------------
```

Note that DHCP snooping/relaying should be configured properly in order for this to work.

Operator Debugging

Verbose debug output is sent to debug-trace on compile errors, execution errors, execution output and the exported result variables.

```
A:dut-A>config>subscr-mgmt>sub-ident-pol>primary# script-url "ftp://xxx.xxx.xx.xx/py/
parsefail1.py"
A:dut-A>config>subscr-mgmt>sub-ident-pol>primary# no shutdown

1 2006/07/30 01:17:33.14 UTC MINOR: DEBUG #2001 - Python Compile Error
"Python Compile Error: parsefail1.py
     File "ftp://xxx.xxx.xx.xx/py/parsefail1.py", line 2 def invalid_function():
          ^
IndentationError: expected an indented block
"

A:dut-A>config>subscr-mgmt>sub-ident-pol>primary# script-url "ftp://xxx.xxx.xx.xx/py/
dump.py"

2 2006/07/30 01:24:55.50 UTC MINOR: DEBUG #2001 - Python Output
"Python Output: dump.py htype   = 0
hlen    = 0 hops    = 0 flags   = 0
ciaddr  = '0.0.0.0' yiaddr  = '0.0.0.0' siaddr  = '0.0.0.0' giaddr  = '0.0.0.0' chaddr  =
''
sname   = ''
file    = ''
options = '5\x01\x056\x04\n\x01\x07\n3\x04\x00\x00\x00\xb4\x01\x04\xff\xff\xff
\x00\x1c\x04\n\x02\x02\xffR\x0f\x01\rdut-A|1|1/1/1\xff' "

3 2006/07/30 01:24:55.50 UTC MINOR: DEBUG #2001 - Python Result
"Python Result: dump.py
"

A:dut-A>config>subscr-mgmt>sub-ident-pol>primary# script-url "ftp://xxx.xxx.xx.xx/py/end-
less.py"

4 2006/07/30 01:30:17.27 UTC MINOR: DEBUG #2001 - Python Output
"Python Output: endless.py
"

5 2006/07/30 01:30:17.27 UTC MINOR: DEBUG #2001 - Python Error
"Python Error: endless.py
Traceback (most recent call last):
File "ftp://xxx.xxx.xx.xx/py/endless.py", line 2, in ? FatalError: script interrupted
(timeout)
"
```

Note that all the Python Result events are empty because none of the scripts set any of the output variables.

Python Scripts

Note that the scripts in this section are test scripts and not scripts which the operator would normally use.

dump.py — `from alc import dhcp`

```
def print_field(key, value):
print '%-8s = %r' % (key, value)

def ipaddr2a(ipaddr):
return '%d.%d.%d.%d' % (
(ipaddr & 0xFF000000) >> 24, (ipaddr & 0x00FF0000) >> 16, (ipaddr & 0x0000FF00) >> 8,
(ipaddr & 0x000000FF))

print_field('htype',   dhcp.htype) print_field('hlen',    dhcp.hlen) print_-
field('hops',    dhcp.hops) print_field('flags',   dhcp.flags) print_field('ciaddr',
ipaddr2a(dhcp.ciaddr)) print_field('yiaddr', ipaddr2a(dhcp.yiaddr)) print_field('siaddr',
ipaddr2a(dhcp.siaddr)) print_field('giaddr', ipaddr2a(dhcp.giaddr)) print_field('chaddr',
dhcp.chaddr) print_field('sname',   dhcp.sname) print_field('file',    dhcp.file) print_-
field('options', str(dhcp.options))
```

us5.py — `import re import alc import struct`

```
# ASAM DSLAM circuit ID comes in three flavors:
#       FENT   string         "TLV1: ATM:3/0:100.33"
#       GELT   octet-stream   0x01010000A0A0A0A0000000640022
#       GENT   string         "ASAM11 atm 1/1/01:100.35"
#
# Script sets output ('subscriber') to 'sub-vpi.vci', e.g.: 'sub-100.33'. circuitid =
str(alc.dhcp.options[82][1])
m = re.search(r'(\d+\.\d+)$', circuitid)
if m:
# FENT and GENT
alc.dhcp.sub_ident = "sub-" + m.group()
elif len(circuitid) >= 3:
# GELT
# Note: what byte order does GELT use for the VCI?
# Assume network byte (big endian) order for now. vpi = struct.unpack('B', circuitid[-3:-
2])[0]
vci = struct.unpack('>H', circuitid[-2:])[0]
alc.dhcp.sub_ident = "sub-%d.%d" % (vpi, vci)
```

Sample Python Scripts

This section provides examples to show how the script can be used in the context of Enhanced Subscriber Management.

Note that these scripts are included for informational purposes only. The operator must customize the script to match their own network and processes.

Example

This script uses the IP address assigned by the DHCP server to derive both *sub_ident* and *sla_profile_string*.

Script:

```
1.  import alc
2.  yiaddr = alc.dhcp.yiaddr
3.  # Subscriber ID equals full client IP address.
4.  # Note: IP address 10.10.10.10 yields 'sub-168430090'
5.  # and not 'sub-10.10.10.10'
6.  alc.dhcp.sub_ident = 'sub-' + str(yiaddr)
7.  # DHCP server is configured such that the third byte (field) of the IP
8.  # address indicates the session Profile ID.
9.  alc.dhcp.sla_profile_string = 'sp-' + str((yiaddr & 0x0000FF00) >> 8)
```

Explanation:

Line 1 — Imports the library “alc” – Library imports can reside anywhere in the script as long as the items are imported before they are used.

Line 2— Assigns the decimal value of the host’s IP address to a temporary variable “yiaddr”. Line 6: The text “sub_“ followed by yiaddr is assigned to “sub_ident” string.

Line 9— The text “sp-“ followed with the third byte of the IP address is assigned to the “sla-profile” string.

If this script is run, for example, with a DHCP-assigned IP address of:

```
yiaddr = 10.10.0.2
```

The following variables are returned:

```
sub_ident: sub-168427522(hex = A0A00002 = 10.10.0.2)
sla_ident: sp-0
```

Example

This script returns the *sub_profile_string* and *sla_profile_string*, which are coded directly in the Option 82 string.

Script:

```
1.  import re
2.  import alc
3.  # option 82 formatted as follows:
4.  # "<subscriber Profile>-<sla-profile>”
5.  ident = str(alc.dhcp.options[82][1])
6.  alc.dhcp.sub_ident = ident
7.  tmp = re.match("(?P<sub>.+)-(?P<sla>.+)", str(ident))
8.  alc.dhcp.sub_profile_string = tmp.group("sub")
9.  alc.dhcp.sla_profile_string = tmp.group("sla")
```

Explanation:

Line 1-2 — Import the libraries “re” and “alc”. Library imports can reside anywhere in the script as long as the items are imported before they are used.

Line 6 — Assigns the full contents of the DHCP Option 82 field to the “sub_ident” variable.

Line 7 — Splits the options 82 string into two parts, separated by “-”.

Line 8 — Assigns the first part of the string to the variable “sub_profile_string”.

Line 9 — Assigns the second part of the string to the variable “sla_profile_string”.

If this script is run, for example, with DHCP option field:

```
options = \x52\x0D\x01\0x0Bmydsl-video
```

The following variables are returned:

```
sub_ident: mydsl-video
sub_profile_string: mydsl
sla_profile_string: video
```

Example

This script parses the Option82 “circuit-id” info inserted in the DHCP packet by a DSLAM, and returns the *sub_ident* string.

Script:

```
1.  import re
2.  import alc
3.  import struct
4.  # Alcatel 7300 ASAM circuit ID comes in three flavors:
5.  #      FENT   string         "TLV1: ATM:3/0:100.33"
6.  #      GELT   octet-stream   0x01010000A0A0A0A0000000640022
7.  #      GENT   string         "ASAM11 atm 1/1/01:100.35"
8.  #
9.  # Script sets output ('subscriber') to 'sub-vpi.vci',
10. # e.g.: 'sub- 100.33'.
11. circuitid = str(alc.dhcp.options[82][1])
12. m = re.search(r'(\d+\.\d+)$', circuitid)
13. if m:
14.    # FENT and GENT
15.    alc.dhcp.sub_ident = "sub-" + m.group()
16. elif len(circuitid) >= 3:
17.    # GELT
18.    # Note: GELT uses network byte (big endian) order for the VCI
19.    vpi = struct.unpack('B', circuitid[-3:-2])[0]
20.    vci = struct.unpack('>H', circuitid[-2:])[0]
21.    alc.dhcp.sub_ident = "sub-%d.%d" % (vpi, vci)
```

Explanation:

Line 1-2 — Import the libraries “re” and “alc” – Library imports can reside anywhere in the script as long as the items are imported before they are used. Needed if regular expressions are used.

Line 3 — Imports the “struct” library, needed if regular expressions are used.

Line 11 — Assigns the contents of the DHCP Option 82 Circuit-ID field to a temporary variable called “circuitid”.

Line 12 — Parses the “circuitid” and checks for the existence of the regular expression “digit.digit” at the end of the string.

Line 15 — If found, a string containing the text “sub-” followed by these two digits is assigned to the variable “sub-ident”.

Line 16 — If not found, and the length of circuit-id is at least 3.

Line 19 — Parses the “circuitid” and assigns the third-last byte to the temporary variable “vpi”.

Line 20 — Parses the “circuitid” and assigns the last two bytes to the temporary variable “vci”.

Line 21 — Assigns a string containing the text “sub-” followed by vpi and vci to the variable “sub-ident”.

If this script is run, for example, with DHCP option field (assigned by an ASAM with FENT card) containing:

```
options = \x52\x16\x01\x14TLV1: ATM:3/0:100.33
```

(in decimal: 80, 22, 1, 20TLV…)

The following variables are returned:

```
sub_ident: sub-100.33
```

If the above script is run, for example, with a DHCP option field (assigned by an ASAM with GELT card) containing:

```
options = \x52\x10\x01\x0E\x01\x01\x00\x00\xA0\xA0\xA0\xA0\x00\x00\x00\x64 \x00\x22
```

(in decimal: 82, 16, 1, 15, 1, 1, 0, 0, 160, 160, 160, 160, 0, 0, 0, 100, 0, 34; corresponding to VPI 100, VCI 34)

Python returns the following variables:

```
sub_ident: sub-100.34
```

If the above script is run, for example, with a DHCP option field (assigned by an ASAM with GENT card) containing:

```
options = \x52\x1A\x01\x18ASAM11 atm 1/1/01:100.35
```

The following variables are returned:

```
sub_ident: sub-100.35
```

Limitations

'%' operator — While %f is supported, %g and %e are not supported.

Floating Point Arithmetic — The floating point arithmetic precision on the box is less than the precision required by the regression suites of Python. For example, pow(2., 30) equals to

1024.*1024.*1024. until five numbers after the point instead of seven and sqrt(9) equals to 3. for the first seven numbers after the point.

Using the round operator fixes these problems. For example, round(pow(2., 30)) equals round(1024.*1024.*1024.) and round(sqrt(9)) equals 3.

RADIUS Script Policy Overview

Python scripts for RADIUS AAA packets support manipulation in subscriber management application. This feature is supported on 7750 SRs and 7450 ESSes in mixed mode. A Python script can be executed in following cases:

- Before the system sends an access-request packet.
- After the system receives an access-accept packet.
- After the system receives an CoA-request packet.
- Before the system sends an accounting-request packet.

The input of the script is the corresponding original packet; and the output of packet will be used as the new corresponding packet for further ESM AAA process.

The **radius-script-policy** contains URLs of a primary and optionally a secondary Python script, which could be a local CF file path or a FTP URL. The configured radius-script-policy could be used in different ESM polices like authentication-policy or radius-accounting-policy.

The following operations are supported within the script:

- Obtain the value of an existing attribute or VSA.
- Modify the value of an existing attribute or VSA.
- Add a new attribute or VSA.
- Remove an existing attribute or VSA.

Note that the following RADIUS attributes or VSA are read-only to Python script:

- Message-Authenticator
- Alc-LI-Action
- Alc-LI-Direction
- Alc-LI-Destination
- Alc-LI-FC
- Alc-LI-Intercept-Id
- Alc-LI-Session-Id

Since R12.0R1, users should use a Python policy (instead of a RADIUS script policy) for RADIUS packet manipulation.

Python RADIUS API

The following new Python objects, alc.radius, have the following methods:

- drop(): Drop the packet.
- header(): Return the a dictionary object includes RADIUS header information.

alc.radius also provides methods to manipulate radius attributes via alc.radius.attributes, which has the following methods:

- get(type): Return the first attribute with specified type as a string.
- getTuple(type): The same as above but returns a tuple of strings.
- getVSA(vendor, type): Return the first VSA as a string.
- getVSATuple(vendor, type): The same as above but returns a tuple of strings.
- set(type, value): Set the specified attribute to the value. The value must be either a string or a tuple of strings.
- setVSA(vendor, type, value): Set the specified VSA to the value. The value must be either a string or a tuple of strings.
- clear(type): Remove the specified attribute.
- clearVSA(vendor, type): Remove the specified VSA.
- isSet(type): Return True if the specified attribute exists, False otherwise.
- isVSASet(vendor, type): Return True if the specified VSA exists, False otherwise.

Sample Script

From alc import radius

#1. Get the value of an existing Attribute

```
Username=radius.attributes.get(1)
```

#2. Modify an existing attribute

```
radius.attributes.set(1, 'Tom')
```

#3. Remove an existing attribute

```
radius.attributes.clear(1)
```

#4. Add a new attribute

```
radius.attributes.set(126,”WIFI-operator”)
```

Python Policy Overview

The Python policy represents a general framework to support all existing and new python features. A Python policy allows users to configure a Python script for specified ESM packet type (such as DHCP, RADIUS, etc.) in a specified direction (ingress/egress). The system will execute the configured script when sending or receiving the specified type of packet.

Within the script, the corresponding original packet will be used as input. The user can use the system-provided API to manipulate the input packet (such as add/change/remove option/attribute) and the changed packet is the output for further ESM processing. And in case of a DHCP transaction cache, the script could also return ESM attributes.

Python policies support following ESM packet types and application:

- RADIUS
- DHCPv4
- DHCPv6
- DHCP Transaction Cache
- Diameter
- Python cache

The following is an example configuration on a specified group interface. The system will execute cf1:/dhcpv4.py after received DHCPv4 discovery and before system forward DHCPv4 request packet.

```
config>python# info
----------------------------------------------
        python-script "dhcpv4" create
            primary-url "cf1:/dhcpv4.py"
            no shutdown
        exit
        python-policy "dhcp" create
            dhcp discover direction ingress script "dhcpv4"
            dhcp request direction egress script "dhcpv4"
        exit
----------------------------------------------
config>service>vprn>sub-if>grp-if>dhcp# info
----------------------------------------------
                        python-policy "dhcp"
                        server 9.9.9.9
                        lease-populate 100
                        gi-address 192.168.100.1
                        no shutdown
----------------------------------------------
```

Python Policy – RADIUS API

The RADIUS API in Python policy uses the same API of the radius-script-policy.

Python Policy – DHCPv4 API

- The system will provide a Python object for input DHCPv4 packet: alc.dhcpv4.
- alc.dhcpv4 has following attributes to represent the DHCPv4 header fields:

Table 22: alc.dhcpv4 Attributes

Class Attrs	DHCPv4 Header Field	Access
alc.dhcpv4.pkt_len	int, Total length of original DHCPv4 packet(UDP/IP header excluded, including pad option) in bytes	read
alc.dhcpv4.pkt_netlen	int, Total length of original DHCPv4 packet(UDP/IP header excluded, pad option in the “options” field excluded) in bytes	read
alc.dhcpv4.op	op	read
alc.dhcpv4.htype	htype	read/write
alc.dhcpv4.hlen	hlen	read/write
alc.dhcpv4.hops	hops	read/write
alc.dhcpv4.xid	xid	read
alc.dhcpv4.secs	secs	read/write
alc.dhcpv4.flags	flags	read/write
alc.dhcpv4.ciaddr	ciaddr	read/write
alc.dhcpv4.yiaddr	yiaddr	read/write
alc.dhcpv4.siaddr	siaddr	read/write
alc.dhcpv4.giaddr	giaddr	read/write
alc.dhcpv4.chaddr	chaddr	read/write
alc.dhcpv4.sname	sname	read/write
alc.dhcpv4.file	file	read/write

All attributes, except alc.dhcpv4.pkt_len and alc.dhcpv4.pkt_netlen, are string with value of the actual bytes in the header.

The following is a list all functions of alc.dhcpv4:

- alc.dhcpv4.drop()
- alc.dhcpv4. getOptionList()
- alc.dhcpv4.pad(min_size=300)
- alc.dhcpv4. get(op_code)
- alc.dhcpv4. set(op_code,valTuple)
- alc.dhcpv4. clear(op_code)
- alc.dhcpv4.get_relayagent() / alc.dhcpv4.set_relayagent(D4OL)
- alc.dhcpv4.get_vendorspecific() / alc.dhcpv4.set_vendorspecific (D4OL)

DHCPv4 allows using sname and file fields to store options. However all DHCPv4 functions will only operate with the “options” field. If a customer wants to manipulate options in the sname/file field, they need to do the parsing work in the script. (extended string.tlvxy method could help here)

- alc.dhcpv4. drop(): The system will drop the result packet
- alc.dhcpv4. getOptionList(): Returns a tuple that includes the option-code of the existing top level DHCPv4 options in the packet.
  - → The order of the element in the tuple is as same as the options that appear in the packet.
  - → If there are multiple instances of the same option, then each instance is one element in the tuple.
  - → Pad option(0) is excluded.
  - → End option(255) is included
  - → Example: A DHCP discovery packet with msg-type/lease-time/request-addr/ parameter-request-list/agent-info/end will return (53,51,50,55,82,255)
- alc.dhcpv4.pad(min_size=300): This function will pad the resulting DHCPv4 packet to the specified min_size with pad option(0) after executing the whole script. Padding will not be added if the result packet is already >=min_size. The default value of min_size is 300. Although not defined in DHCPv4 RFC, many DHCPv4 implementation has a minimal length requirement of 300 bytes. So this function could pad the result packet to the specified min_size.

- alc.dhcpv4. get(op_code): Returns a tuple that includes all instances of the specified top level option as a string. The value of this string is the exact bytes of the option as it appears in packet(excludes option-code and option-len).
  - → If the specified option does not exist, then the function will return ()
  - → If the certain instance of specified option does not have the value (len=0 or doesn't have len and value part), then the function will return "" for that instance in the tuple.
  - → Example: There is an address lease time option(51) in the packet, with value 60, then alc.dhcpv4.get(51) should return: ('\x00\x00\x00\x3c',)
- alc.dhcpv4. set(op-code,valTuple): This function will in fact remove the all existing instances of specified top level option and insert a list of new options. Each element in valTuple is a string, representing one instance of the new option to be inserted; For each new option, the option-code is specified in op-code. The option-len is the length of the element. The rest of option is the element itself.
  - → Example: To insert an address lease time option(51) in the packet, with the value 60; use alc.dhcpv4.set(51, ('\x00\x00\x00\x3c',))
- alc.dhcpv4. clear(op-code): This function will remove the all existing instances of specified top level option.
- Although alc.dhcpv4.get() and alc.dhcpv4.set() provide a generic way to manipulate DHCPv4 top level options, but some DHCPv4 options have a complex/hierarchical structure like option82 and option43. To provide a friendly access to these kinds of options, the system provides the following options' specific functions:
  - → alc.dhcpv4.get_relayagent() / alc.dhcpv4.set_relayagent(D4OL)
  - → alc.dhcpv4.get_vendorspecific() / alc.dhcpv4.set_vendorspecific (D4OL)

  All alc.dhcpv4.get_XXX() will return a data structure:"D4OL" (DHCPv4 Option List)
  - → D4OL is a list. Each element in the list represents an instance of that option. For example, if there are 3 option82 in the "options" field of packet, then get_relayagent() will return a list of 3 elements. Each element represents one instance of the option in the packet.
  - → Each element in D4OL is a dict (called dict as "D4O" in this example):
    - – The key of D4O is the sub-option- code. The value is a list of strings of sub-option-value of all instance of the sub-option.

  All alc.dhcpv4.set_XX(OPDL) will accept D4OL as the parameter. Remove all existing instances of the corresponding options and then insert the new options represented by specified D4OL.

Examples:

For a packet with an option82 like following

```
Option: (82) Agent Information Option
     Length: 22
     Option (82) Suboption: (1) Agent Circuit ID
          Length: 8
          Agent Circuit ID: 4a616e737656e73
     Option 82 Suboption: (2) Agent Remote ID
          Length: 10
          Agent Remote ID 62617369632364617461
```

The option-data is (hex formatted)
“01:08:4a:61:6e:73:73:65:6e:73:02:0a:62:61:73:69:63:23:64:61:74:61”

The following is an example script:

```
import alc

option82_list=alc.dhcpv4.get_relayagent()
#option82_list will be
[{1:['\x4a\x61\x6e:73\x73\x65\x6e\x73',],2:['\x62\x61\x73\x69\x63\x23\x64\x61\x74\x61',]}
,]
Option82_list[0][2][0]='basic#video' #change remote-id to 'basic#video'
alc.dhcpv4.set_relayagent(option82_list)#update the option82
```

Python Policy – DHCPv6 API

- The system provides a Python object for input DHCPv6 packet: alc.dhcpv6
- alc.dhcpv6 has following attributes to represent the DHCPv6 header fields:

Table 23: DHCPv6 Header Fields

Class Attrs	DHCPv6 Header Field	Client/ Server Msg	Relay Msg	Access
alc.dhcpv6.pkt_len	int, Total length of original DHCPv4 packet(UDP/IP header excluded) in bytes	●	●	Read
alc.dhcpv6.msg_type	msg-type	●	●	Read
alc.dhcpv6.transaction_id	transaction-id	●		Read
alc.dhcpv6.hop_count	hop-count		●	read/write
alc.dhcpv6.link_addr	link-address		●	read/write
alc.dhcpv6.peer_addr	peer-address		●	read/write

All header fields (as the attribute of alc.dhcpv6 class) are strings(except pkt_len) with exact bytes as it appears in the packet.

If certain attribute does not exist in the given msg-type, for example if the link_attr does not exist in client/server message(C/S msg), then its value should be None.

- The following is a list of all functions in the class:
  - → alc.dhcpv6.drop()
  - → alc.dhcpv6. getOptionList()
  - → alc.dhcpv6. get(op-code)
  - → alc.dhcpv6. set(op-code,valTuple)
  - → alc.dhcpv6. clear(op-code)
  - → alc.dhcpv6.get_iana() / alc.dhcpv6.set_iana(OPDL)
  - → alc.dhcpv6.get_iata() / alc.dhcpv6.set_iata(OPDL)
  - → alc.dhcpv6.get_vendoropts() / alc.dhcpv6.set_vendoropts(OPDL)
  - → alc.dhcpv6.get_iapd() / alc.dhcpv6.set_iapd(OPDL)
  - → alc.dhcpv6.get_relaymsg()
  - → alc.dhcpv6.set_relaymsg(packet)
- alc.dhcpv6. drop(): The system will drop the resulting packet.

- alc.dhcpv6. getOptionList(): Returns a tuple that includes the option-code of the existing top level DHCPv6 options in the packet. The order of the element in the tuple is as same as the options appear in the packet. If there are multiple instances of same option, then each instance is one element in the tuple. For example:

  → A C/S Msg with Elapsed Time/Client Identifier/IANA/FQDN/Vendor Class/Option Request will return (8,1,3,39,16,6).

  → A Relay Msg with Relay Message option only will return (9)

- alc.dhcpv6. get(op-code): Returns a tuple that includes all instances of the specified top level option as string The value of this string is the exact bytes of the option as it appears in packet(excludes option-code and option-len). If the specified option does not exist in the input packet, then it will return ().

  Examples:

  → If there is a Status Code option in the packet, status-code 0 and status-msg:"Address Assigned"; then alc.dhcpv6.get(13) should return: ('\x00\x00Address Assigned',)

- alc.dhcpv6. set(op-code,valTuple): This function will remove the all existing instances of the specified top level option and insert a list of new options. Each element in valTuple is a string, representing one instance of the new option to be inserted. For each new option, the option-code is specified in op-code, the option-len is the length of the element, reset of option is the element itself.

  → To insert a Status Code options with status-code 0 and status-msg:"Address Assigned"; use alc.dhcpv6.set(13, ('\x00\x00Address Assigned',))

- alc.dhcpv6. clear(op-code): This function will remove the all existing instances of specified top level option.

- Although alc.dhcpv6.get() and alc.dhcpv6.set() provide a generic way to manipulate DHCPv6 top level options, but some DHCPv6 options have more complex/hierarchical structure like IA_NA/IA_TA, etc. To provide a friendly access to these kinds of options, the system provides the following options specific functions:

  → alc.dhcpv6.get_iana() / alc.dhcpv6.set_iana(OPDL)

  → alc.dhcpv6.get_iata() / alc.dhcpv6.set_iata(OPDL)

  → alc.dhcpv6.get_vendoropts() / alc.dhcpv6.set_vendoropts(OPDL)

  → alc.dhcpv6.get_iapd() / alc.dhcpv6.set_iapd(OPDL)

All alc.dhcpv6.get_XXX() will return a data structure:“OPDL” (Option Data structure List)

→ OPDL is a list. Each element in the list represents an instance of that option. For example, if there are 3 IANA in the packet, then get_iana() will return a list of 3 elements, each element represent one instance of IANA option in the packet.

→ Each element in OPDL is a list, referred to as “OPD” in this list.

- Each element in OPD represent one field in the option(option-code and option-len are not included), the order of the element is as same as the fields appear in the option
- For field that could be parsed into sub-option by RFC, then the element is a dict, the key of this dict is the sub-option type, if sub-option is one of following supported-sub-option, the value to the key is a sub-option_OPDL represent the list of that specific sub-option
  - IAADDR(5)
  - Status Code(13)
  - IAPREFIX(26)

Else, if the sub-option is **not** one of above, then the value to the key is a list of string of sub-option-data, each string represent one instance of the sub-option.

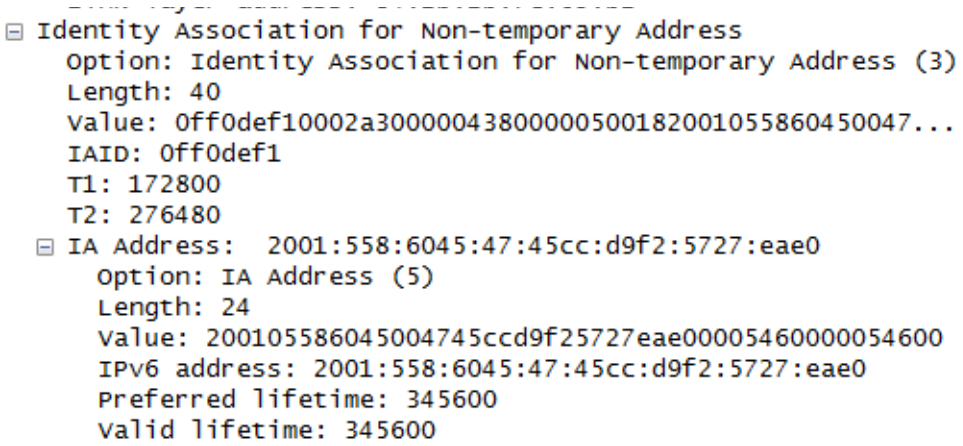
- The structure of sub-option_OPDL of IAADDR is: [[v6_addr,prefer_lifetime, valid_lifetime,sub-option_OPDL], etc.]
- The structure of sub-option_OPDL of Status Code is: [[status-code,status-msg], etc.]
- The structure of sub-option_OPDL of IAPREFIX is: [[prefer_lifetime,valid_lifetime,prefix-len,v6prefix,sub-option_OPDL],..]
- For the field (by RFC definition) could be parsed into sub-options, but it does not actually exist, then the dict will be empty {}
- For field that cannot be parsed into sub-option by RFC, the element is a string of exact bytes of that field

All alc.dhcpv6.set_XX(OPDL) will accept an OPDL as the parameter. Remove all existing instances of the corresponding options and then insert new options represented by the specified OPDL.

- alc.dhcpv6.get_iana()/alc.dhcpv6.get_iana(OPDL)
- The general OPDL structure for these two functions is: [[IAID_val,T1_val,T2_val, sub-option_dict]]
- The structure of sub-option_dict is:{sub-option-type:sub-option_val}
- If sub-option is supported-sub-option, then sub-option_val is a sub-option_OPDL
- For all other sub-options, the sub-option_val is a list of string of sub-option-data

Examples:

For a packet with an IANA option like following:



The option-data is (hex formatted)

```
"0f:f0:de:f1:00:02:a3:00:00:04:38:00:00:05:00:18:20:01:05:58:60:45:00:47:45:cc:d9:f2:57:2
7:ea:e0:00:05:46:00:00:05:46:00"
```

The following is an example script:

```
import alc

iana_list=alc.dhcpv6.get_iana()
#iana_list will be
[['\x0f\xf0\xde\xf1','\x00\x02\xa3\x00','\x00\x04\\x38\x00',{5:[['\x20\x01\x05\x58\x60\x4
5\x00\x47\x45\xcc\xd9\xf2\x57\x27\xea\xe0','\x00\x05\x46\x00','\x00\x05\x46\x00',{}]]}]]
iana_list[0][1]='\x00\x00\x04\xb0' #change T1 to 1200
alc.dhcpv6.set_iana(iana_list)#update the iana
```

- alc.dhcpv6.get_iata()/alc.dhcpv6.get_iata(OPDL)

  → The general OPDL structure for these two functions is: [[IAID_val, sub-option_dict]]

  → The structure of sub-option_dict is:{sub-option-type:sub-option_val}

  → If sub-option is supported-sub-option, then sub-option_val is a sub-option_OPDL

  → For all other sub-options, the sub-option_val is a list of string of sub-option-data

  Examples: These two function are very similar with IANA, so the examples are skipped here.
- alc.dhcpv6.get_iapd()/alc.dhcpv6.get_iapd(OPDL)
- The general OPDL structure for these two functions is: [[IAID_val,T1_va1,T2_val, sub-option_dict]]
- The structure of sub-option_dict is:{sub-option-type:sub-option_val}
- If sub-option is supported-sub-option, then sub-option_val is a sub-option_OPDL

- For all other sub-options, the sub-option_val is a list of string of sub-option-data

Examples: For a packet with IA_PD like following:

```
Identity Association for Prefix Delegation
  Option: Identity Association for Prefix Delegation (25)
  Length: 41
  Value: 00000001000007080000b40001a001900000e1000015180...
  IAID: 00000001
  T1: 1800
  T2: 2880
  IA Prefix
    Option: IA Prefix (26)
    Length: 25
    Value: 00000e10000151803820010db80002000000000000000000...
    Preferred lifetime: 3600
    Valid lifetime: 86400
    Prefix length: 56
    Prefix address: 2001:db8:2::
```

The option-data is (hex formatted)

```
"00:00:00:01:00:00:07:08:00:00:0b:40:00:1a:00:19:00:00:0e:10:00:01:51:80:38:20:01:0d:b8:0
0:02:00:00:00:00:00:00:00:00:00:00"
```

Following is an example script:

```
import alc

iapd_list=alc.dhcpv6.get_iapd()
#iapd_list will be
[['\x00\x00\x00\x01','\x00\x00\x07\x08','\x00\x00\x0b\x40',{26:[['\x00\x00\x0e\x10','\x00
\x01\x51\x80','\x38', '\x20\x01\x0d\xb8\x00\x02\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00',{}]]}]]
iapd_list[0][1]='\x00\x00\x04\xb0' #change T1 to 1200
alc.dhcpv6.set_iapd(iapd_list)#update the iapd
```

- alc.dhcpv6.get_vendoropts()/alc.dhcpv6.get_vendoropts (OPDL)
  - → The general OPDL structure for these two functions is: [[enterpriseid_val, sub-option_dict]]
  - → The structure of sub-option_dict is: {sub-option-type:sub-option_val}
  - → If sub-option is supported-sub-option, then sub-option_val is a sub-option_OPDL
  - → For all other sub-options, the sub-option_val is a list of string of sub-option-data

Examples: For a packet with vendor options like following:

```
Vendor-specific Information
  Option: Vendor-specific Information (17)
  Length: 40
  Value: 0000197f0001000969612d6e615f3030310002000969612d...
  Enterprise ID: Panthera Networks, Inc. (6527)
  option
    option code: 1
    option length: 9
    option-data
  option
    option code: 2
    option length: 9
    option-data
  option
    option code: 3
    option length: 1
    option-data
  option
    option code: 4
    option length: 1
    option-data
```

The option-data is (hex formatted)

```
"00:00:19:7f:00:01:00:09:69:61:2d:6e:61:5f:30:30:31:00:02:00:09:69:61:2d:70:64:5f:30:30:3
1:00:03:00:01:38:00:04:00:01:40"
```

The following is an example script:

```
import alc

vendoropts_list=alc.dhcpv6.get_vendoropts()
# vendoropts_list will be [['\x00\x00\x19\x7f',
{1:['\x69\x61\x2d\x6e\x61\x5f\x30\x30\x31'],2:[
'\x69\x61\x2d\x70\x64\x5f\x30\x30\x31'],3:['\x38'], 4:['\x40']}]]
iapd_list[0][1][4][0]='\x60' #change sub-option 4's value to 0x60
alc.dhcpv6.set_vendoropts(vendoropts_list)#update the vendor options
```

- For DHCPv6 relay message, the "Relay Message" option embedded a full DHCPv6 packet and the embedded packet could itself have a "Replay Message" option which embedded another DHCPv6 packet.

  To provide direct access to embedded DHCPv6 packet, the system provides following functions:

  → alc.dhcpv6.get_relaymsg()

  → alc.dhcpv6.set_relaymsg(packet)

- alc.dhcpv6. get_relaymsg(): This function will return a populated alc.dhcpv6 object. which means the returned object was initialized with the DHCPv6 packet embedded in "Relay Message" option as the input.
- alc.dhcpv6. set_relaymsg(packet): This function will accept an alc.dhcpv6 object as a parameter. This object will replace existing "Relay Message" option.

- Example-1 script for single relay message:

```
import alc
#input packet is a relay-reply msg
embed_dhcpv6_packet=alc.dhcpv6.get_relaymsg()
iana_list=embed_dhcpv6_packet.get_iana()
iana_list[0][1]='\x00\x00\x04\xb0' #change T1 to 1200
embed_dhcpv6_packet.set_iana(iana_list)#update the iana of the embedded packet
alc.dhcpv6.set_relaymsg(embed_dhcpv6_packet)#update the Relay Message option
```

- Example-2 script for double relay message (relay of relay):

```
import alc
#input packet is a relay-reply msg
embed_lv1_packet=alc.dhcpv6.get_relaymsg() #get the level=1 embedded packet
embed_lv2_packet= embed_lv1_packet.get_relaymsg()#get level-2 packet embedded in level-1
packet
iana_list=embed_dhcpv6_packet.get_iana()
iana_list[0][1]='\x00\x00\x04\xb0' #change T1 to 1200
embed_dhcpv6_packet.set_iana(iana_list)#update the iana
embed_lv1_packet.set_relaymsg(embed_lv2_packet)#update the Relay Message option of lv1 msg
alc.dhcpv6.set_relaymsg(embed_lv1_packet)#update the Relay Message option of the top level
msg
```

Python Policy – Diameter API

The alc.diameter Python module provides an API for Diameter message manipulation.

Terminology used in the API description:

- **top-level-AVP** — AVP appearing at the top level in a Diameter message, i.e. not embedded in the Data field of a grouped AVP
- **embedded-AVP** — AVP embedded in the Data field of a grouped AVP. An embedded AVP can be a grouped AVP. This is called nesting.
- **AVP-tuple —** Python tuple with following values:
  (AVP code, vendor id, flags)
  AVP code : integer, AVP header field
  vendor id : integer, AVP header field
  flags : string, AVP header field
- **AVP-value-tuple —** Python tuple with following values:
  (flags, data)
  flags: string, AVP header field
  data: string, AVP data field
- **AVP-key-tuple —** Python tuple with following values:
  (AVP code, vendor id)
  AVP code: integer, AVP header field
  vendor id : integer, AVP header field
- **grouped-AVP-value-tuple —** Python tuple with following values:
  (flags, grouped-AVP-dictionary)
  flags: string, AVP header field
- **grouped-AVP-dictionary** - Python dictionary with following key:value pairs:

  {AVP-key-tuple : [AVP-value-tuple or grouped-AVP-value-tuple, …], ... }
  key = AVP-key-tuple
  value = list of AVP-value-tuples or list of grouped-AVP-value-tuples
- **grouped-AVP-decode-tuple —** Python tuple with following values:

  (AVP-key-tuple, …)
  tuple of AVP-key-tuples
- **AVP-order-tuple —** Python tuple with following values:

  (AVP-key-tuple, …)
  tuple of AVP-key-tuples

Table 24 displays attributes available in **alc.diameter** module providing access to the Diameter message header:

Table 24: Diameter Message Header alc.diameter Attributes

Attribute	Description	Type	Access
application_id	Diameter message header field	string	Read/Write
code	Diameter message header field	string	Read/Write
end_to_end_id	Diameter message header field	string	Read/Write
flags	Diameter message header field	string	Read/Write
hop_by_hop_id	Diameter message header field	string	Read/Write
msg_length	Diameter message header field. The value is the message length of the original diameter message.	string	Read
version	Diameter message header field	string	Read/Write

Table 25 displays methods available in **alc.diameter** module providing message and AVP manipulation functionality:

Table 25: Message and AVP Manipulation Functionality alc.diameter Methods

Method	Description
clear_avps(AVP code, vendor id) AVP code, vendor id = top-level-AVP	Remove all instances of the specified AVP from the message. Applies to top-level-AVP's only. If the specified AVP is not present, no python error is generated. Vendor id value zero matches top-level-AVP's without Vendor Id field. Return value: None For example: diameter.clear_avps(256, 12645)
drop()	Drop the Diameter message. Packet is consumed at TCP level (ack send). A drop will trigger retransmits on Diameter level. Return value: None For example: diameter.drop()

Table 25: Message and AVP Manipulation Functionality alc.diameter Methods (Continued)

Method	Description
get_avps(AVP code, vendor id) AVP code, vendor id = top-level-AVP	Returns a list of AVP-value-tuples. Each AVP-value-tuple represents an instance of the specified AVP in the message. Applies to top-level-AVP's only. The position in the list corresponds with the position of the AVP instance in the message at that stage in the script. When executed before any clear or set AVP method, the list order corresponds with the AVP order in the received message. If the specified AVP is a grouped AVP, then the data will contain all the embedded-AVP's. An empty list is returned if the specified AVP is not present. Vendor id value zero matches top-level-AVP's without Vendor Id field. For example: diameter.get_avps(263, 0) [('@', 'bng.alcatel-lucent.com;1398156449;28')]
get_avps_list()	Returns a list of AVP-tuples. Each AVP-tuple represents an instance of an AVP in the message. Applies to top-level-AVP's only. The position in the list corresponds with the position of the AVP in the message at that stage in the script. When executed before any clear or set AVP method, the list order corresponds with the AVP order in the received message. When multiple instances of an AVP are present in the message, then there will be multiple instances in the list. The Vendor Id has value zero when not present. Grouped AVPs cannot be distinguished from other AVPs in the list. For example: diameter.get_avps_list() [(263, 0, '@'), (264, 0, '@'), (296, 0, '@'), (258, 0, '@'), (416, 0, '@'), (415, 0, '@'), (268, 0, '@'), (55, 0, '@'), (456, 0, '@'), (456, 0, '@'), (456, 0, '@'), (293, 0, '@'), (256, 12645, '\x80')]

Table 25: Message and AVP Manipulation Functionality alc.diameter Methods (Continued)

Method	Description
get_grouped_avps(AVP code, vendor id, grouped-AVP-decode-tuple) AVP code, vendor id = top-level-AVP	Returns a list of grouped-AVP-value-tuples with each grouped-AVP-dictionary entry representing an embedded AVP. Each grouped-AVP-value-tuple represents an instance of the specified AVP in the message. Applies to top-level-AVP's of type grouped only. The position in the list corresponds with the position of the grouped AVP instance in the message at that stage in the script. When executed before any clear or set AVP method, the list order corresponds with the AVP order in the received message. If the grouped-AVP-decode-tuple is empty, only the specified top-level-AVP is expanded in a grouped-AVP-value-tuple, with each grouped-AVP-dictionary entry representing an embedded AVP and all dictionary values of type "list of AVP-value-tuples" To expand nested AVPs (grouped AVPs embedded in a grouped AVP), the grouped top-level-AVP and grouped embedded-AVP to expand must be added to the grouped-AVP-decode-tuple. All grouped AVP's in the grouped-AVP-decode-tuple will be expanded in a list of grouped-AVP-value-tuples provided that their embedding AVP is also in the list. The position of the embedded AVPs in the grouped-AVP-dictionary does not correspond with the position in the grouped AVP. If the specified top-level-AVP is not a grouped AVP, then a Python error is generated: 'ValueError: malformed diameter message". For example: To expand the Multiple Services Credit Control (456) grouped top level AVP: diameter.get_grouped_avps(456,0,()) [('@', {(432, 0): [('@', '\x00\x00\x00\x01')], (431, 0): [('@', '\x00\x00\x01\xa4@\x00\x00\x0c\x00\x00\x00d')], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]}), ('@', {(432, 0): [('@', '\x00\x00\x00\x02')], (431, 0): [('@', '\x00\x00\x01\xa4@\x00\x00\x0c\x00\x00\x03\x84')], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]}), ('@', {(432, 0): [('@', '\x00\x00\x00\x03')], (431, 0): [('@', '\x00\x00\x01\xa4@\x00\x00\x0c\x00\x00\x00<')], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]})]

Table 25: Message and AVP Manipulation Functionality alc.diameter Methods (Continued)

Method	Description
	To expand the nested Granted-Service-Unit AVP (code 431) in the grouped Multiple Services Credit Control top level AVP (code 456): diameter.get_grouped_avps(456,0,((456,0),(431,0))) [('@', {(432, 0): [('@', '\x00\x00\x00\x01')], (431, 0): [('@', {(420, 0): [('@', '\x00\x00\x00d')]})], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]}), ('@', {(432, 0): [('@', '\x00\x00\x00\x02')], (431, 0): [('@', {(420, 0): [('@', '\x00\x00\x03\x84')]})], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]}), ('@', {(432, 0): [('@', '\x00\x00\x00\x03')], (431, 0): [('@', {(420, 0): [('@', '\x00\x00\x00<')]})], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]})]
set_avps(AVP code, vendor id, list of AVP-value-tuples) AVP code, vendor id = top-level-AVP	Remove all instances of the specified top-level-AVP from the message. For each entry in the AVP-value-tuple list, a top-level-AVP instance is inserted. If the specified vendor id value is zero, then no vendor id field is inserted and setting the Vendor-Specific bit in the flags field of the AVP value tuple will then result in a Python error: "ValueError: no vendor ID but vendor flag set". If the specified vendor id value is non-zero, then a vendor id field is inserted and not setting the Vendor-Specific bit in the flags field of the AVP value tuple will result in a Python error: "ValueError: vendor ID but vendor flag not set". Padding between AVPs, AVP length and Diameter message length are adapted accordingly by the system. Return value is None. For example: diameter.set_avps(461,0,[('\x40', 'Pytho.n-1'), ('\x40', 'Python-2')])
set_fixed_position_avps(AVP-order-tuple)	Put the specified top-level-AVPs at the beginning of the message in the order as specified in the AVP-order-tuple. This method overrides the order of the top-level-AVPs in the resulting Diameter message. If for example the session-id AVP must appear as first in the message, then the corresponding AVP-key-tuple must be included in the first position of the AVP-order-tuple. AVPs not present in the message but specified in the AVP-order-tuple are ignored.

Table 25: Message and AVP Manipulation Functionality alc.diameter Methods (Continued)

Method	Description
	AVPs present in the message and not specified in the AVP-order-tuple will be included in the final message after the AVPs listed in the AVP-order-tuple. The order is deterministic but implementation specific. This method can appear at any point in the script. The last call will override the previous one. From a black box viewpoint, this method is executed at the end of the script: the result of the call is not reflected in the list returned by a subsequent get_avp_list() call. Return value: None For example: diameter.set_fixed_position_avps(((263,0), (264,0), (296,0), (268,0)
set_grouped_avps(AVP code, vendor id, list of grouped-AVP-value-tuples) AVP code, vendor id = top-level-AVP	Remove all instances of the specified grouped top-level-AVP from the message. For each entry in the grouped-AVP-value-tuples list, a grouped top-level-AVP instance is inserted. The order of the embedded-AVPs in the grouped AVP cannot be specified.If the specified vendor id value is zero, then no vendor id field is inserted and setting the Vendor-Specific bit in the flags field of the AVP value tuple will then result in a Python error: “ValueError: no vendor ID but vendor flag set”. If the specified vendor id value is non-zero, then a vendor id field is inserted and not setting the Vendor-Specific bit in the flags field of the AVP value tuple will result in a Python error: “ValueError: vendor ID but vendor flag not set”. Padding between AVPs, AVP length and Diameter message length are adapted accordingly. Return value is None. For example: diameter.set_grouped_avps(456,0,[('@', {(432, 0): [('@', '\x00\x00\x00\x01')], (431, 0): [('@', {(420, 0): [('@', '\x00\x00\x00\x2b')]})], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]}), ('@', {(432, 0): [('@', '\x00\x00\x00\x03')], (431, 0): [('@', {(420, 0): [('@', '\x00\x00\x00\x53')]})], (448, 0): [('@', '\x00\x00\x04\xb0')], (268, 0): [('@', '\x00\x00\x07\xd1')]})])

To enable Diameter message manipulation using Python, a python-policy must be configured in the diameter-peer-policy. For example:

```
configure
    aaa
        diameter-peer-policy "diameter-peer-policy-1" create
            description "Diameter peer policy"
            applications gy
            origin-host "bng.alcatel-lucent.com"
            origin-realm "alcatel-lucent.com"
            python-policy "py-policy-diam-1"
            source-address 192.0.2.5
            peer "peer-1" create
                address 172.16.1.1
                destination-host "server.alcatel-lucent.com"
                destination-realm "alcatel-lucent.com"
                no shutdown
            exit
        exit
    exit
```

The python-policy specifies the python-script to use for each Diameter message type received or transmitted on a Diameter peer. In the ingress direction, the Python script is executed when the corresponding packet type is received on the Diameter peer but prior to further processing in the system. In the egress direction, the Python script is executed prior to sending the corresponding packet type on the Diameter peer. For example:

```
configure
    python
        python-policy "py-policy-diam-1" create
            description "Python policy"
            diameter ccr direction egress script "diameter-1"
            diameter cca direction ingress script "diameter-2"
        exit
    exit
```

The python-script specifies the location of the script and optional protection mechanism. For example:

```
configure
    python
        python-script "diameter-1" create
            primary-url "ftp://usr:pwd@192.0.2.1/./py/diam-1.py"
            no shutdown
        exit
    exit
```

As an example, the diam-1.py script, clears the M-bit from the Event-Timestamp AVP (code 55):

```
 from alc import diameter
 avp55=diameter.get_avps(55,0)
 diameter.set_avps(55,0,[('\x00',avp55[0][1])])
```

Python Policy – DHCP Transaction Cache API

A DHCP transaction cache (DTC) is a short-lived cache during DHCPv4/v6 transaction. The cache could be used to store user-chosen information or return ESM attributes via a Python script. The DTC's lifetime is only during a single DHCP transaction (for example, only between discovery-offer, request-reply). This includes both alc.dtc.store() data and alc.dtc.setESM() data. DTC is also a transaction specific cache, which means the cached information could only be accessed by the Python script running in same DHCP transaction.

The following are the DTC APIs:

- alc.dtc.derivedId: A string used as a LUDB lookup key
- alc.dtc.store(cache-key, cache-value): Store the value with the specified cache-key in DTC, both key and value are string.
- alc.dtc.retrieve(cache-key): Returns the cached value string according to the specified cache-key, raise exception if specified key does not exist.
- alc.dtc.setESM(ESM-key, value): Sets the specified ESM attribute, which could be used when system creating the ESM host. Note that due to the short-live nature of DTC, setESM should be used in final DHCP transaction before system create ESM host., such as DHCPv4 REQUEST-ACK.

- The following is a list of supported ESM-key and its corresponding python type:
  - → alc.dtc.accountingPolicy:str
  - → alc.dtc.ancpString:str
  - → alc.dtc.appProfileString:str
  - → alc.dtc.catMapString:str
  - → alc.dtc.defGw:str
  - → alc.dtc.dhcpv4GIAddr:str
  - → alc.dtc.dhcpv4Pool:str
  - → alc.dtc.dhcpv4ServerAddr:str
  - → alc.dtc.dhcpv6LinkAddr:str
  - → alc.dtc.dhcpv6ServerAddr:str
  - → alc.dtc.intDestId:str
  - → alc.dtc.ipAddress:str
  - → alc.dtc.ipv4LeaseTime:int
  - → alc.dtc.ipv4PrimDns:str
  - → alc.dtc.ipv4SecDns:str
  - → alc.dtc.ipv6Address:str
  - → alc.dtc.ipv6DelegatedPrefix:str
  - → alc.dtc.ipv6DelegatedPrefixLength:int
  - → alc.dtc.ipv6PrefixPool:str
  - → alc.dtc.ipv6PrimDns:str
  - → alc.dtc.ipv6SecDns:str
  - → alc.dtc.ipv6SlaacPrefix:str
  - → alc.dtc.ipv6WanPool:str
  - → alc.dtc.msapGroupInterface:str
  - → alc.dtc.msapPolicy:str
  - → alc.dtc.msapServiceId:str
  - → alc.dtc.primNbns:str
  - → alc.dtc.retailServiceId:str
  - → alc.dtc.secNbns:str
  - → alc.dtc.slaProfileString:str
  - → alc.dtc.subIdent:str
  - → alc.dtc.subProfileString:str
  - → alc.dtc.subnetMask:str

Python Cache Support

Python cache support allows information to be shared across different run times of the same python script or even different python scripts in a programmatic way. It essentially provides a central memory cache and a set of APIs which let the user store and retrieve strings. For example, a DHCP python-script could store a DHCP option into cache and later a RADIUS python-script could retrieve stored string and add it into access-request.

Each cached entry in the cache is a tuple of (**key**, **val**). **key** is used as entry id. **val** is the string to be cached. Both **key** and **val** are strings. The max length of the key is 512 bytes. Future more, the combine length of key+val is limited by the configured value of **entry-size** *size* command in the python-policy.

The Python cache can be enabled per python-policy. Each python policy has its own cache memory which script in other python-policy cannot access. This also implies that the key of a cached entry in different a python policy could overlap.

The user can also specify the max number cache entry per python policy the command **max-entries** command. System has a global limit for python cache memory of 256MB.

The cached entries could be made persistent by saving it to CF card. This can be enabled with the **persistence** command in the python policy.

Note: From memory consumption point of view, with MCS enabled, each cached entry will have a corresponding MCS record, so each entry will consume twice amount of memory.

The system also supports syncing the python cache across chassis with MCS. This can be configured per python policy with the **mcs-peer** command in the python policy.

Each cached entry has a remaining lifetime. If it decreases over time, the system will remove the cached entry if its remaining lifetime is 0. The remaining lifetime can be changed using a system-provided API. The initial lifetime of a newly created cache entry is 600 seconds.

The following are the python cache APIs in a module alc.cache:

- alc.cache.save(val,key): Saves the val identified by the key into the cache. If there is an existing cache entry with same key, then it will be overwritten with the new val. An exception will be raised if the save failed (for example, due to exceeding the max number of entries).
- alc.cache.retrieve(key): Returns the stored entry’s val identified by the key. A KeyError exception will be raised if the specified entry does not exist.
- alc.cache.clear(key): Removes the cached entry identified by the key. Raise KeyError exception if the specified entry does not exist.
- cache.get_lifetime(key): The system will return a integer as seconds of remaining lifetime of the specified entry. It will return none if the specified entry does not exist. An exception will be raised for any other error.

- cache.set_lifetime(key,new_lifetime): The new_lifetime value is an integer. The system will set the remaining lifetime of the specified entry to the number of seconds of the new_lifetime. An exception will be raised for any error including specified entry does not exist. If the new_lifetime>=max_lifetime(configurable using the **max-entry-life** command in the python policy), then the system will set the actual lifetime to the max_lifetime.

Applying a Python Policy

The following is a list of places that a Python policy could be applied:

- Under capture SAP — Apply to the DHCPv4/v6 packets sent/received on the capture SAP.
- Under group-interface — Apply to DHCPv4/v6 packets sent/received on the group-interface.
- Under subscriber-interface — Apply to DHCPv4 packets on the retail subscriber interface.
- In the radius-server-policy — Apply to the RADIUS packets sent/received to/from the RADIUS servers configured in the radius-server-policy.
- In the radius-proxy-server — Apply to the RADIUS packets on the client side of proxy.
- In the diameter-peer-policy — Apply to the Diameter packets send/received on the Diameter peers configured in the policy.

Python Script Protection

In order to protect the Python script from unintended changes, the SR-OS supports a new Python script file format:SRPY. Since 12.0R1, SRPY includes a key based hash(HMAC) of the original script content. When the system loads a script with SRPY format, a hash will be computed by using a configured key and script content. The result hash will be compared to the embedded hash. If it is the same, then this script is considered valid. Otherwise, the system will abort with a warning message.

Users can configure **protection hmac-sha256 key <***key***>** within a Python script. To mandate, all configured scripts must be in SRPY format.

The system provides a tool command (**tool perform python-script protect**) to convert a Python script into SRPY format.

Tips and Tricks

- Use xrange() instead of range().
- Avoid too many string operations. The following scripts provide the same output:

```
# This script takes 2.5 seconds. s = ""
for c in 'A'*50000:
s += str(ord(c)) + ', ' print '[' + s[:-2] + ']'

# This script takes 0.1 seconds. print map(ord, 'A'*50000)
```