smc-python Documentation

Release 0.6.2

David LePage

Apr 06, 2021

Contents

Contents:

CHAPTER 1

Introduction

This is the smc-python library to interface with the Stonesoft Management Center.

This acts as an front-end to simplify interactions and simplify scripting when looking to integrate automated functionality.

The smc-python library also has a CLI that provides a command completion syntax to provide guidance on commands to be run, and can be run remotely from the Stonesoft Firewalls. All actions interact with the Stonesoft Management Center (SMC), and commands specific to the FW's are proxied by the SMC to the individual deviecs.

A good place to start is the Installation and Getting Started section of the help docs.

Current versions are validated using:

- Stonesoft Management Server 5.10, 6.0, 6.1, 6.1.2 (Windows/Linux)
- Python 2.7.x,
- Python 3.4, Python 3.5 (version >= 0.4)

CHAPTER 2

Installation

Install the package by using a package manager such as pip.

```
pip install git+https://github.com/gabstopper/smc-python.git
```

Or optionally download the latest tarball (windows): smc-python, unzip and run:

python setup.py install

Dependencies on this library are:

- requests

If installation is required on a non-internet facing machine, you will have to download the smc-python tarball and dependencies manually and install by running python setup install.

Once the smc-python package has been installed, you can import the main packages into a python script:

```
import smc.elements
import smc.core.engine
import smc.core.engines
import smc.policy
import smc.elements.system
```

To remove the package, simply run:

pip uninstall smc-python

For more information on next steps, please see creating the session

CHAPTER 3

Creating the session

In order to interact with the SMC REST API, you must first obtain a valid login session. The session is generated by authenticating an API Client and the associated authentication key.

Once the login session has been retrieved successfully, all commands or controls will reuse the same session.

When exiting, call *smc.api.web.logout()* to remove the active session from the SMC.

Note: Idle API sessions will still time out after a default timeout on the SMC server.

Steps to enable API Communication on the Stonesoft Management Center:

1. Enable SMC API service on the properties of the Management Server
2. Create an API Client and obtain the 'authentication key'

3.1 Configuring credentials

Credentials to obtain a session are obtained using the following methods (in order):

- Provide credentials in session.login() constructor
- In alternate specified file path (specified in login constructor)
- In INI configured file at users ~/.smcrc
- Environment variables

Each method is described in more detail below.

3.1.1 Method parameters

Example of providing the connect information through method parameters:

```
from smc import session

session.login(url='http://1.1.1.1:8082', api_key='xxxxxxxxxxxxxxxx')
....do stuff....
session.logout()
```

If a specific API version is requested, you can add the following argument to the login constructor. Otherwise the latest API version available will be used.

To find supported versions using unauthenticated call to SMC:

```
>>> from smc.api.session import available_api_versions
>>> available_api_versions('http://1.1.1.1:8082')
[5.1, 6.1, 6.2]
```

Set up connection to specific version:

```
from smc import session
session.login(url='http://1.1.1.1:8082', api_key='xxxxxxxxxxxxxxxx',
              api_version='6.1')
```

Logging in to a specific Admin Domain:

```
session.login(url='http://1.1.1.1:8082', api_key='xxxxxxxxxxxxxxxx',
              domain='mydomain')
```

Note: If an admin domain is specified but the SMC does not have domains configured, you will be placed in the 'Shared Domain'.

In order to use SSL connections, you must first associate a private key and certificate with the SMC API server. This is done under the Management Server properties and SMC API. Obtain the certificate for use by the client. It is recommended to ensure your certificate has the subjectAltName field set per RFC 2818.

Using SSL and specify certificate for verifying:

```
from smc import session
session.login(url='https://1.1.1.1:8082', api_key='xxxxxxxxxxxxxxxx',
              verify='/Users/davidlepage/home/mycacert.pem')
```

Using SSL to the SMC without SSL validation (NOT recommended)

```
from smc import session
session.login(url='https://1.1.1.1:8082', api_key='xxxxxxxxxxxxxxxxxx',
              verify=False)
```

See also:

smc.api.session.Session.login() for constructor arguments.

3.1.2 Configuration File

It is possible to store the SMC connection information in ~/.smcrc in order to simplify the login and eliminate the need to populate scripts with api key information. Syntax for ~/.smcrc:

```
[smc]
smc_address=1.1.1.1
smc_apikey=xxxxxxxxxxxxxxxxxx
api_version=6.1
smc_port=8082
smc_ssl=True
verify_ssl=True
ssl_cert_file='/Users/davidlepage/home/mycacert.pem'
domain=mydomain
```

Then from launching scripts, you can do:

```
session.login()
session.logout()
```

Note: It is possible to override the location of .smcrc by using the 'alt_filepath' argument in the login constructor.

```
session.login(alt_filepath='/home/somedir/test')
```

3.1.3 Environment Variables

If setting environment variables, the following are supported:

```
SMC_ADDRESS=http://1.1.1.1:8082
SMC_API_KEY=123abc
SMC_CLIENT_CERT=path/to/cert
SMC_TIMEOUT = 30 (seconds)
SMC_API_VERSION = 6.1 (optional - uses latest by default)
SMC_DOMAIN = name of domain, Shared is default
```

The minimum variables that need to be present are `SMC_ADDRESS` and `SMC_API_KEY`:

```
export SMC_ADDRESS = http://1.1.1.1:8082
export SMC_API_KEY = foobarkey
```

Based on the session login constructor, you can also pass kwargs using the parameter *SMC_EXTRA_ARGS*.

Once the session has been successfully obtained, there is no reason to re-authenticate a new session unless *logout* has been called.

Note: The SMC will automatically purge idle sessions after a configurable amount of time.

3.2 Handling retries on server busy

It is possible to override the default behavior for retrying a CRUD operation based on receiving a "Service Unavailable" (HTTP 503) response. By default, no retry is attempted. You can override this behavior and allow the API to retry an operation using a backoff algorithm.

This can be enabled through the session login constructor using the *retry_on_busy* boolean or after session login by calling *set_retry_on_busy*. If called from session login, default parameters are provided for all retry related settings. If you require more granularity, call after session login.

Note: By default, the following operation types are eligible for retry (GET/POST/PUT). You can override this by calling session.set_retry_on_busy(method_whitelist=['GET', 'POST', 'DELETE'])

Calling from session login:

```
session.login(url='https://x.x.x.x:8082', api_key='xxxxxxxxxxxxxxx',
          verify=False, timeout=30, retry_on_busy=True)
```

Calling after session login:

```
session.login()
session.set_retry_on_busy(total=5, backoff_factor=0.1)
...
session.logout()
```

If you are using an preferences file, place the following into your .smcrc:

```
[smc]
retry_on_busy=True
```

You can also set this on as an environment variable using the *SMC_EXTRA_ARGS* variable:

```
os.environ['SMC_EXTRA_ARGS'] = '{"retry_on_busy": "True"}'
```

3.3 Handling proxies

To disable the use of an intermediate proxy and force the connection to go direct, you can add the following environment variable:

```
os.environ['no_proxy'] = 'my.smc.at.domain'
```

3.4 Logging helper

To enable logging from smc-python, you can utilize the standard python logger or use convenience methods provided. These are typically called before session login:

```
from smc import set_file_logger
set_file_logger(log_level=10, path='/Users/foo/smc-test.log')
...
```

Or use a stream logger and also optionally enable urllib3 messages:

```
from smc import set_stream_logger
set_stream_logger(log_level=logging.DEBUG)
set_stream_logger(log_level=logging.DEBUG, logger_name='urllib3')
```

Another logging option is to add the following lines to your script:

```
import logging
logging.getLogger()
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(
    level=logging.DEBUG, format='%(asctime)s %(levelname)s %(name)s.%(funcName)s:
↪%(message)s')
```

The `format` parameter follows the standard python logging module syntax.

CHAPTER 4

Resources

Resources in the Stonesoft Management Center are typically accessed in a couple different ways.

The first would be by using the elements collection interface to search for elements of a specific type.

For example, if you are looking for Hosts by a given IP address:

```
>>> from smc.elements.network import Host
>>> list(Host.objects.filter('192.168'))
[Host(name=aws-192.168.4.254), Host(name=host-192.168.4.135), Host(name=host-192.168.
↪4.94), Host(name=host-192.168.4.79)]
```

See *Collections* for more information on search capabilities.

It is also possible to access resources directly:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('sg_vm')
>>> print(list(engine.nodes))
[Node(name=ngf-1065), Node(name=ngf-1035)]

>>> print(list(engine.routing))
[Routing(name=Interface 0,level=interface), Routing(name=Interface 1,level=interface),
↪ Routing(name=Interface 2,level=interface), Routing(name=Tunnel Interface 2000,
↪level=interface), Routing(name=Tunnel Interface 2001,level=interface)]
```

Retrieving a specific host element by name:

```
>>> from smc.elements.network import Host
>>> host = Host('kali')
>>> print(host.href)
http://172.18.1.150:8082/6.2/elements/host/978
```

When elements are referenced initially, they are lazy loaded until attributes or methods of the element are used that require the data. Once an element has been 'inflated' due to a reference being called (property, method, etc), the resultant element data is stored in a per instance cache.

Example of how elements are lazy loaded:

```
>>> from smc.elements.network import Host
>>> host = Host('kali')
>>> vars(host)
{'_meta': None, '_name': 'kali'}        #Base level attributes, only instance created
>>> host.href   # Call to retrieve this resource link reference loads instance meta␣
↪(1 SMC query)
u'http://172.18.1.150:8082/6.2/elements/host/978'
>>> vars(host)
{'_meta': Meta(name=u'kali', href=u'http://172.18.1.150:8082/6.2/elements/host/978',␣
↪type=u'host'), '_name': 'kali'}
>>> host.data   # Request to a method/attribute that requires the data attribute␣
↪inflates the instance (1 SMC query)
{u'comment': u'this is a searchable comment', u'read_only': False, u'ipv6_address': u
↪'2001:db8:85a3::8a2e:370:7334', u'name': u'kali', u'third_party_monitoring': {u
↪'netflow': False, u'snmp_trap': False}, u'system': False, u'link': [{u'href': u
↪'http://172.18.1.150:8082/6.2/elements/host/978', u'type': u'host', u'rel': u'self'}
↪, {u'href': u'http://172.18.1.150:8082/6.2/elements/host/978/export', u'rel': u
↪'export'}, {u'href': u'http://172.18.1.150:8082/6.2/elements/host/978/search_
↪category_tags_from_element', u'rel': u'search_category_tags_from_element'}], u'key
↪': 978, u'address': u'1.1.11.1', u'secondary': [u'7.7.7.7']}
>>> vars(host)
{'data': {u'comment': u'this is a searchable comment', u'read_only': False, u'ipv6_
↪address': u'2001:db8:85a3::8a2e:370:7334', u'name': u'kali', u'third_party_
↪monitoring': {u'netflow': False, u'snmp_trap': False}, u'system': False, u'link': [
↪{u'href': u'http://172.18.1.150:8082/6.2/elements/host/978', u'type': u'host', u'rel
↪': u'self'}, {u'href': u'http://172.18.1.150:8082/6.2/elements/host/978/export', u
↪'rel': u'export'}, {u'href': u'http://172.18.1.150:8082/6.2/elements/host/978/
↪search_category_tags_from_element', u'rel': u'search_category_tags_from_element'}],␣
↪u'key': 978, u'address': u'1.1.11.1', u'secondary': [u'7.7.7.7']}, '_meta':␣
↪Meta(name=u'kali', href=u'http://172.18.1.150:8082/6.2/elements/host/978', type=u
↪'host'), '_name': 'kali'}
```

At most 2 queries will be required to retrieve an element as a resource.

Cache contents can be viewed in their raw json format by calling the 'data' property.

Note: When modifications are made to a specific element, they are submitted back to the SMC using the originally retrieved ETag to ensure the element has not been modified since the original retrieval.

CHAPTER 5

Collections

Resource collections are designed to be similar to how Django query sets work and provide a similar API.

5.1 ElementCollection

ElementCollections are available on all elements that inherit from *smc.base.model.Element*, and are also available for general searching across any element with an SMC entry point.

An *ElementCollection* can be constructed without making a single query to the SMC database. No query will occur until you do something to evaluate the collection.

You can evaluate a collection in the following ways:

- **Iteration**. An ElementCollection is iterable, and it executes the SMC query the first time you iterate over it. For example, this will retrieve all host elements:

```
>>> for host in Host.objects.all():
...     print(host.name, host.address)
```

- **list()**. Force evaluation of a collection by calling list() on it:

```
>>> elements = list(Host.objects.all())
```

- *first()*. Helper collection method to retrieve only the first element in the search query:

```
>>> host = Host.objects.iterator()
>>> host.first()
Host(name=SMC)
```

If you don't need all results and only a single element, rather than getting an ElementCollection iterator, you can obtain this directly from the CollectionManager:

```
>>> Host.objects.first()
Host(name=SMC)
```

- *last()*. Helper collection method to retrieve only the last element in the search query:

```
>>> host = Host.objects.iterator()
>>> host.last()
Host(name=kali3)
```

- *exists()*. Helper collection method to evaluate whether there are results:

```
>>> hosts = Host.objects.filter('1.1.1.1')
>>> if hosts.exists():
...   for host in list(hosts):
...     print(host.name, host.address)
...
('hax0r', '1.1.1.1')
('host', '1.1.1.1')
('hostelement', '1.1.1.1')
('abcdefghijklmnop', '1.1.1.1')
```

- *count()*. Helper collection method which returns the number of results. You can still obtain the results after:

```
>>> it = Router.objects.iterator()
>>> query1 = it.filter('10.10.10.1')
>>> query1.count()
3
>>> list(query1)
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
```

- *batch()*. Iterator returning batches of results with specific by quantity. If limit() is also chained, it is ignored as batch and limit are mutually exclusive operations.

```
>>> for hosts in Host.objects.batch(2):
...   print(hosts)
...
[Host(name=SMC), Host(name=172.18.1.135)]
[Host(name=172.18.2.254), Host(name=host)]
[Host(name=host-54.76.110.156), Host(name=host-192.168.4.135)]
[Host(name=external primary DNS resolver), Host(name=host-192.168.4.94)]
...
```

5.1.1 Methods that return a new ElementCollection

There are multiple methods in an ElementCollection that allow you to refine how the query or results are returned. Each chained method returns a new ElementCollection with aggregated search parameters.

- *filter()*. Provide a filter string to narrow the search to a string value that will be used in a 'contains' match:

```
>>> host = Host.objects.filter('172.18.1')
>>> list(host)
[Host(name=172.18.1.135), Host(name=SMC)]
```

`filter` can also take a keyword argument to filter specifically on an attribute. The keyword argument should match a valid attribute for the element type, and value to match:

```
>>> list(Router.objects.filter(address='10.10.10.1'))
[Router(name=Router-10.10.10.1)]
```

Note: Two additional keyword arguments can be passed to filter, *exact_match=True* and/or *case_sensitive=False*.

- *limit()*. Limit the number of results to return.

```
>>> list(Host.objects.all().limit(3))
[Host(name=SMC), Host(name=172.18.1.135), Host(name=172.18.2.254)]
```

- *all()*. Return all results.

```
>>> list(Host.objects.all())
```

5.1.2 Basic rules on searching

- By default searches use a 'contains' logic. If you specify a filter string, the SMC API will return elements that contain that string. Therefore, if partial searches are performed, you may receive multiple matches:

```
>>> list(Router.objects.filter('10.10'))
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
```

- When the search is evaluated, the elements returned contain only meta data and not the full payload for each element matching the search. The search query is built based on provided parameters to narrow the scope and only a single query is made to SMC.
- When using a filter, the SMC API will search the name, comment and relevant field/s for the element type selected.

  Each element type will have it's own searchable fields. For example, in addition to the name and comment field, a Host element will search the address and secondary address fields. This is automatic.

  For example, the following would find Host elements with this value in any of the Host fields specified above:

```
>>> Host.objects.filter('111.111.111.111')
```

- Setting `exact_match=True` on the filter query will only match on an element's name or comment field and is a case sensitive match. The SMC is case sensitive, so unless you need an element by exact case, this field is not required. By default, `exact_match=False`.
- In v0.5.6, `case_sensitive=False` can be set on the filter query to change the behavior of case sensitive matches. If not set, case_sensitive=True.
- Using a keyword argument with 'filter' will provide element introspection against the attributes to perform an exact match. In general, using a kwarg is most effective when searching for network elements. Since the default search is a 'contains' match, a search for '10.10.10.1' may return elements with values: '10.10.10.1', '10.10.10.10', and '110.10.10.1'. Using an attribute/value would override the default search behavior and attempt to only match on the specified attribute:

```
>>> list(Router.objects.filter('10.10.10.1'))
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
```

The above query returns multiple elements contains matches. To explicitly define the attribute to make an exact match, change the filter to use a kwarg (the `address` attribute is the defined ipaddress for *smc.elements.network.Router*):

```
>>> list(Router.objects.filter(address='10.10.10.1'))
[Router(name=Router-10.10.10.1)]
```

Note: When using keyword matching with `filter`, a single query will be performed using the attribute value, returning a list of 'contains' matches. For each element match returned from the first query, an additional query is performed to retrieve the element attributes.

To reduce the number of additional queries performed when using keyword matching, use a limit on the number of return elements:

```
>>> list(Router.objects.filter(address='10.10.10.1').limit(1))
[Router(name=Router-10.10.10.1)]
```

5.1.3 Additional Examples

Obtain an iterator from the collection manager for re-use:

```
>>> iterator = Router.objects.iterator()
>>> query1 = iterator.filter('10.10.10.1')
>>> list(query1)
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
>>> query2 = query1.filter(address='10.10.10.1')
>>> list(query2)
[Router(name=Router-10.10.10.1)]
```

Access a collection directly on an Element type:

```
>>> list(Host.objects.all())
[Host(name=SMC), Host(name=172.18.1.135), Host(name=172.18.2.254), Host(name=host)]
...
>>> list(TCPService.objects.filter('HTTP'))
[TCPService(name=HTTPS_No_Decryption), TCPService(name=Squid HTTP proxy),␣
↪TCPService(name=HTTP to Web SaaS)]
```

Limit number of return entries:

```
>>> list(Host.objects.limit(3))
[Host(name=SMC), Host(name=172.18.1.135), Host(name=172.18.2.254)]
```

Limit and filter the results using a chainable syntax:

```
>>> list(Host.objects.filter('172.18.1').limit(5))
[Host(name=172.18.1.135), Host(name=SMC), Host(name=TIE Server), Host(name=172.18.1.
↪93)]
```

Get a host collection when partial IP address known:

```
>>> list(Host.objects.filter('192.168'))
[Host(name=aws-192.168.4.254), Host(name=host-192.168.4.135), Host(name=host-192.168.
↪4.94), Host(name=host-192.168.4.79)]
```

When filtering is performed, by default search queries will 'wildcard' the results. To only return an exact match of the search query, use the optional flag 'exact_match':

```
>>> list(TCPService.objects.filter('8080'), exact_match=True))
[TCPService(name=TCP_8080), TCPService(name=HTTP proxy), TCPService(name=SSH),␣
↪TCPService(name=SSM SSH)]
```

Additional convenience functions are provided on the collections to simplify navigating through results such as `count`, `first`, and `last`:

```
>>> query1 = iterator.filter('10.10.10.1')
>>> if query1.exists():
...   list(query1.all())
...
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]

>>> list(query1)
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
>>> query1.first()
Router(name=Router-110.10.10.10)
>>> query1.last()
Router(name=Router-10.10.10.1)
>>> query1.count()
3
>>> query2 = query1.filter(address='10.10.10.1')  # Add kwarg to new query
>>> list(query2)
[Router(name=Router-10.10.10.1)]
```

5.2 General Search

If a search is required for an element type that is not a pre-defined class of *smc.base.model.Element* type in the API, it is still possible to search any valid entry point using `smc.base.collections.Search`.

Search extends ElementCollection and provides additional methods:

- *entry_point()*. Entry points are top level collections available from the SMC.
- *context_filter()*. Context filters are special filters that can return more generalized results such as all engines, etc.

  Available context filters:

  - *fw_clusters* - list all firewalls
  - *engine_clusters* - all clusters
  - *ips_clusters* - ips only clusters
  - *layer2_clusters* - layer2 only clusters
  - *network_elements* - all network element types
  - *services* - all service types
  - *services_and_applications* - all services and applications
  - *tags* - element tags
  - *situations* - inspection situations
- *unused()*. Search for all unused elements:

```
>>> list(Search.objects.unused())
[RouteVPN(name=myvpn), RouteVPN(name=mygre), RouteVPN(name=avpn),␣
↪RouteVPN(name=avpn)]
...
```

- *duplicates()*. Search for all duplicate elements:

```
>>> list(Search.objects.duplicates())
[Host(name=foohost), Router(name=router-1.1.1.1)]
...
```

Using `Search` is useful if there is not a direct class representation of the element you are attempting to retrieve. If there is an entry point for the target element type, you can return any element.

First, find all available searchable objects (also known as 'entry points'):

```
>>> from smc.elements.resources import Search
>>> Search.object_types()
['elements', 'sub_ipv6_fw_policy', 'ids_alert', 'application_not_specific_tag', 'fw_
↪alert', 'virtual_ips', 'sidewinder_tag', 'os_specific_tag', 'eia_application_usage_
↪group_tag', 'external_bgp_peer', 'local_cluster_cvi_alias', 'ssl_vpn_service_profile
↪', 'active_directory_server', 'eia_golden_image_tag', 'client_gateway', 'situation_
↪tag', 'api_client', 'tls_match_situation', 'ssl_vpn_policy', 'category_group_tag',
↪'ip_list', 'vpn_profile', 'ipv6_access_list', 'appliance_information', 'single_
↪layer2', 'ei_executable', 'community_access_list']
...
```

Once the type of interest is found, the elements can be retrieved using the entry point:

```
>>> list(Search.objects.entry_point('vpn'))
[PolicyVPN(name=Amazon AWS), PolicyVPN(name=sg_vm_vpn), PolicyVPN(name=TRITON AP-WEB␣
↪Cloud VPN)]
```

And subsequently add a filter as well:

```
>>> list(Search.objects.entry_point('vpn').filter('AWS'))
[PolicyVPN(name=Amazon AWS)]
```

Additional examples:

Searching all services for port 80:

```
>>> list(Search.objects.entry_point('services').filter('80'))
[TCPService(name=tcp80443), TCPService(name=HTTP to Web SaaS),␣
↪EthernetService(name=IPX over Ethernet 802.2), UDPService(name=udp_10070-10080),␣
↪Protocol(name=HTTP8080), TCPService(name=tcp_10070-10080), TCPService(name=TCP_
↪8080), TCPService(name=tcp_3478-3480), EthernetService(name=IPX over Ethernet 802.3␣
↪(Novell)), TCPService(name=HTTP), TCPService(name=SSM HTTP), TCPService(name=HTTP␣
↪(SafeSearch)), IPService(name=ISO-IP), UDPService(name=udp_3478-3480),␣
↪TCPService(name=HTTP (with URL Logging))]
```

Only Network elements with '172.18.1':

```
>>> list(Search.objects.context_filter('network_elements').filter('172.18.1'))
[Host(name=172.18.1.135), Host(name=SMC), Network(name=Any network),␣
↪FirewallCluster(name=sg_vm), Element(name=dc-smtp), Network(name=network-172.18.1.0/
↪24), LogServer(name=LogServer 172.18.1.150), Layer3Firewall(name=testfw),␣
↪Element(name=SecurID), Element(name=Windows 2003 DHCP), AddressRange(name=net-172.
↪18.1.100-172.18.1.120), ManagementServer(name=Management Server)]
```

(continues on next page)

(continued from previous page)

Only firewall clusters:

```
>>> list(Search.objects.context_filter('fw_clusters'))
[FirewallCluster(name=sg_vm), Layer3VirtualEngine(name=ve-8),␣
↪Layer3Firewall(name=testfw), Layer3Firewall(name=i-04eec8f019adf818e (us-east-2a)),␣
↪MasterEngine(name=master)]
```

In addition to using more generic filters, with general searches, you can also specify multiple valid entry points by specifying the string filter comma separated.

For example, finding all hosts and routers:

```
>>> list(Search.objects.entry_point('router,host'))
[Host(name=172.18.2.254), Router(name=router-172.18.3.129), Host(name=All Routers␣
↪(Site-Local))]
```

Filter based on hosts and routers:

```
>>> list(Search.objects.entry_point('router,host').filter('172.18.1'))
[Host(name=172.18.1.135), Host(name=SMC), Host(name=ePolicy Orchestrator),␣
↪Router(name=router-172.18.1.225), Host(name=fw-internal-primary),␣
↪Router(name=router-172.18.1.209)]
```

Note: If an element of class *smc.base.model.Element* exists, it will be returned as that type to enable access to the objects instance methods. If there is no element defined, a dynamic class is produced from type Element.

For example, searching for object of type 'ids_alert' will produce a dynamic class as type Element and will have access to the base class methods:

```
>>> list(Search.objects.entry_point('ids_alert'))
[IdsAlertDynamic(name=Default alert), IdsAlertDynamic(name=Test alert),␣
↪IdsAlertDynamic(name=System alert)]
```

Classes deriving from *smc.base.model.Element* are found in the API reference, for example: *Administration*

CHAPTER 6

Elements

Elements are the building blocks for policy and include types such as Networks, Hosts, Services, Groups, Lists, Zones, etc.

6.1 Create

Elements within the Stonesoft Management Server are common object types that are referenced by other configurable areas of the system such as policy, routing, VPN, etc.

This is not an exhaustive list, all supported element types can be found in the API reference documentation: *Administration*

- *Hosts*
- *AddressRange*
- *Networks*
- *Routers*
- *Groups*
- *DomainName*
- *IPList* (SMC API >= 6.1)
- *URLListApplication* (SMC API >= 6.1)
- *Zone*
- *LogicalInterface*
- *TCPService*
- *UDPService*
- *IPService*
- *EthernetService*

- *ServiceGroup*
- *TCPServiceGroup*
- *UDPServiceGroup*
- *IPServiceGroup*
- *ICMPService*
- *ICMPv6Service*

Oftentimes these objects are cross referenced within the configuration, like when creating rule or NAT policy. All calls to create() will return the href of the new element stored in the SMC or will raise an exception for failure.

Examples of creating elements are as follows:

```
>> from smc.elements.network import Host, Network, AddressRange
>>> host = Host.create(name='hostelement', address='1.1.1.1')
>>> host
Host(name=hostelement)
>>> host.address
u'1.1.1.1'
>>> network = Network.create(name='networkelement', ipv4_network='1.1.1.0/24',␣
↪comment='mynet')
>>> network
Network(name=networkelement)
>>> network.ipv4_network
u'1.1.1.0/24'
>>> network.comment
u'mynet'
>>> AddressRange.create(name='myaddrrange', ip_range='1.1.1.1-1.1.1.10')
AddressRange(name=myaddrrange)
```

Check the various reference documentation for defined elements supported.

6.2 Update

Updating elements can be done in multiple ways. In most cases, making modifications to an element through methods or element attributes are the preferred way. Modifications done through existing methods/attributes are done idempotent to the elements cache. In order to commit these changes to the SMC, calling .update() is required unless explicitly documented otherwise.

Note: There are some edge cases where .update() is called automatically like when modifying interfaces where multiple areas are updated. These will be documented on the method.

Another way to update an element is by providing the kwarg values in the update() call directly.

For example, setting the address, secondary address and comment for a host element can be done in update by providing kwargs:

```
host = Host('kali')
host.update(
        address='3.3.3.3',
        secondary=['12.12.12.12'],
        comment='something about this host')
```

There is also a generic modify_attribute on *smc.base.model.Element* which is essentially the same as calling .update(kwargs) above:

```
host = Host('kali')
host.modify_attribute(
        address='3.3.3.3',
        secondary=['12.12.12.12'],
        comment='something about this host')
```

A much more low-level way of modifying an element is to modify the data in cache (dict) directly. After making the modifications, you must also call .update() to submit the change.

Modifying a service element after reviewing the element cache:

```
>>> service = TCPService.create(name='aservice', min_dst_port=9090)
>>> service
TCPService(name=aservice)
...
>>> pprint(service.data)
{u'key': 3551,
 u'link': [{u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551',
            u'rel': u'self',
            u'type': u'tcp_service'},
           {u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551/export',
            u'rel': u'export'},
           {u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551/search_
↪category_tags_from_element',
            u'rel': u'search_category_tags_from_element'}],
 u'min_dst_port': 9090,
 u'name': u'aservice',
 u'read_only': False,
 u'system': False}
 ...
>>> service.data['min_dst_port'] = 9091
>>> service.update()    # Submit to SMC, cache is refreshed
'http://172.18.1.150:8082/6.2/elements/tcp_service/3551'
...
>>> pprint(service.data)
{u'key': 3551,
 u'link': [{u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551',
            u'rel': u'self',
            u'type': u'tcp_service'},
           {u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551/export',
            u'rel': u'export'},
           {u'href': u'http://172.18.1.150:8082/6.2/elements/tcp_service/3551/search_
↪category_tags_from_element',
            u'rel': u'search_category_tags_from_element'}],
 u'min_dst_port': 9091,
 u'name': u'aservice',
 u'read_only': False,
 u'system': False}
```

Attributes supported by elements are documented in the API Reference: *Administration*

6.3 Delete

Deleting elements is done by using the base class delete method. If the element has already been fetched, the ETag of the original fetch is stored with the element cache and will be provided during the delete.

Deleting a host:

```
>>> from smc.elements.network import Host
>>> Host('kali').delete()
```

6.4 Functions or methods that modify

Some functions or element methods may make modifications to an element depending on the operation. These functions are documented and will also be decorated with and `autcommit` decorator. This allows you to queue changes locally before submitting them to the SMC by calling `update`. To override this behavior, you can either pass `autocommit=True` to these functions or set `session.AUTOCOMMIT=True` on the session. Most methods will autocommit by default with exception of methods defined in `smc.core.properties`.

CHAPTER 7

Engines

Engines are the definitions for a layer 3 FW, layer 2 FW, IPS, Cluster Firewalls, Master Engines, or Virtual Engines.

An engine defines the basic settings to make the device or virtual instance operational such as interfaces, routes, ip addresses, networks, dns servers, etc.

Creating engines are done using the Firewall specific base classes in *smc.core.engines*

Nodes are individual devices represented as properties of an engine element. In the case of single device deployments, there is only one node. For clusters, there will be at a minimum 2 nodes, max of 16. The *smc.core.node* class represents the interface to managing and sending commands individually to a node in a cluster.

By default, each constructor will have default values for the interface used for management (interface 0). This can be overridden as necessary.

Once engines are created, they can be retrieved directly by using *smc.core.engine.Engine* or directly by their engine type. The __repr__ of Engine will show a descriptive view of the engine type regardless of how the context was obtained.

```
>>> Engine('sg_vm')
FirewallCluster(name=sg_vm)
...
>>> from smc.core.engines import FirewallCluster
>>> FirewallCluster('sg_vm')
FirewallCluster(name=sg_vm)
```

Note: There is no difference between the two options. Loading from the Engine class tends to be easier as you are not required to know the engine type to obtain the context.

7.1 Create

7.1.1 Layer3 Firewall

For Layer 3 single firewall engines, the minimum requirements are to specify a name, management IP and management network. By default, the Layer 3 firewall will use interface 0 as the management port. This can be overridden in the constructor if a different interface is required.

To create a layer 3 firewall:

```
>>> from smc.core.engines import Layer3Firewall
>>> Layer3Firewall.create(name='firewall', mgmt_ip='1.1.1.1', mgmt_network='1.1.1.0/24
↪')
Layer3Firewall(name=firewall)
```

See reference for more information: *smc.core.engines.Layer3Firewall*

7.1.2 Layer 2 Firewall

For Layer 2 Firewall and IPS engines, an inline interface pair will automatically be created using interfaces 1-2 but can be overridden in the constructor to use different interface mappings. At least one inline pair or a capture interface is required to successfully create.

Creating a Layer2 Firewall with alternative management interface and DNS settings:

```
>>> from smc.core.engines import Layer2Firewall
>>> Layer2Firewall.create(name='myfirewall', mgmt_ip='1.1.1.1', mgmt_network='1.1.1.0/
↪24', mgmt_interface=5, domain_server_address=['172.18.1.20'])
Layer2Firewall(name=myfirewall)
```

See reference for more information: *smc.core.engines.Layer2Firewall*

7.1.3 IPS Engine

Similar to Layer2Firewall, at least one inline interface pair or a capture interface is required to successfully create.

Use alternative inline interface pair configuration (mgmt on interface 0):

```
>>> from smc.core.engines import IPS
>>> IPS.create(name='myips',
...            mgmt_ip='1.1.1.1',
...            mgmt_network='1.1.1.0/24',
...            inline_interface='5-6')
IPS(name=myips)
```

See reference for more information: *smc.core.engines.IPS*

7.1.4 Master Engine

A Master Engine is used to manage virtual engine nodes and provides in system virtualization. Master Engine controls administrative aspects and specifies how resources are allocated to the virtual engines.

Create a master engine with a single management interface, then add 2 more physical interface for virtual engine allocation:

```
>>> from smc.core.engines import MasterEngine
>>> engine = MasterEngine.create(name='api-master',
...                              mgmt_ip='1.1.1.1',
...                              mgmt_network='1.1.1.0/24',
...                              master_type='firewall',
...                              domain_server_address=['8.8.4.4', '7.7.7.7'])
>>> print(engine)
>>> MasterEngine(name=api-master)
>>> engine.physical_interface.add(1)     # add intefaces
>>> engine.physical_interface.add(2)
>>> for intf in engine.interface.all():
...   print(intf)
...
PhysicalInterface(name=Interface 1)
PhysicalInterface(name=Interface 0)
PhysicalInterface(name=Interface 2)
```

See *smc.core.engines.MasterEngine* for more details.

7.1.5 Layer3Virtual Engine

A virtual engine is a host that resides on a Master Engine node used for multiple FW contexts. Stonesoft maps a 'virtual resource' to a virtual engine as a way to map the master engine interface to the individual instance residing within the physical device.

In order to create a virtual engine, you must first manually create the Master Engine from the SMC, then create the interfaces that will be used for the virtual instances.

The first step in creating the virtual engine is to create the virtual resource and map that to a physical interface or VLAN on the master engine. Once that has been created, add IP addresses to the virtual engine interfaces as necessary.

First create the virtual resource on the already created Master Engine:

```
>>> from smc.core.engines import MasterEngine
>>> engine = MasterEngine('api-master')
>>> engine.virtual_resource.create('ve-1', vfw_id=1)
'http://1.1.1.1:8082/6.1/elements/master_engine/62629/virtual_resource/756'
```

See *smc.core.engine.VirtualResource.create()* for more information.

Creating a layer 3 virtual engine with two single physical interfaces:

```
>>> from smc.core.engines import Layer3VirtualEngine
>>> Layer3VirtualEngine.create(name='myvirtual',
...                            master_engine='api-master',
...                            virtual_resource='ve-1',
...                            interfaces=[{'address':'5.5.5.5','network_value':'5.5.
↪5.0/24','interface_id':0},
...                                        {'address':'6.6.6.6','network_value':'6.6.
↪6.0/24','interface_id':1}]
Layer3VirtualEngine(name=myvirtual)
```

Note: Virtual engine interface numbering takes into account the dedicated interface for the master engine. For example, if the master engine is using physical interface 0 for management, the virtual engine may be assigned physical interface 1 for use. From an indexing perspective, the naming within the virtual engine configuration will start at

interface 0 but be using physical interface 1.

See reference for more information: *smc.core.engines.Layer3VirtualEngine*

7.1.6 Firewall Cluster

Creating a layer 3 firewall cluster requires additional interface related information to bootstrap the engine properly. With NGFW clusters, a "cluster virtual interface" is required (if only one interface is used) to specify the cluster address as well as each engine specific node IP address. In addition, a macaddress is required for packetdispatch functionality (recommended HA configuration).

By default, the FirewallCluster class will allow as many nodes as needed (up to 16 per cluster) for the singular interface. The node specific interfaces are defined by passing in the 'nodes' argument to the constructor as follows:

Create a 3 node cluster:

```
>>> from smc.core.engines import FirewallCluster
>>> FirewallCluster.create(name='mycluster',
...                        cluster_virtual='1.1.1.1',
...                        cluster_mask='1.1.1.0/24',
...                        cluster_nic=0,
...                        macaddress='02:02:02:02:02:02',
...                        nodes=[{'address':'1.1.1.2','network_value':'1.1.1.0/24',
↪'nodeid':1},
...                               {'address':'1.1.1.3','network_value':'1.1.1.0/24',
↪'nodeid':2},
...                               {'address':'1.1.1.4','network_value':'1.1.1.0/24',
↪'nodeid':3}],
...                        domain_server_address=['8.8.8.8'])
FirewallCluster(name=mycluster)
```

See *smc.core.engines.FirewallCluster* for more info

7.1.7 MasterEngine Cluster

Create a master engine cluster for redundancy. Master Engine clusters support active/standby mode.

Create the cluster and add a second interface for each cluster node:

```
>>> MasterEngineCluster.create(name='engine-cluster',
...                            master_type='firewall',
...                            macaddress='22:22:22:22:22:22',
...                            nodes=[{'address':'5.5.5.2','network_value':'5.5.5.0/24
↪','nodeid':1},
...                                   {'address':'5.5.5.3','network_value':'5.5.5.0/24
↪','nodeid':2}])
MasterEngine(name=engine-cluster)
```

Adding an interface after creation:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('engine-cluster')
>>> engine.physical_interface.add_cluster_interface_on_master_engine(
...                                        interface_id=1,
...                                        macaddress='22:22:22:22:22:33',
```

(continues on next page)

(continued from previous page)

```
...                                                    nodes=[{'address':'6.6.6.2','network_value
↪':'6.6.6.0/24','nodeid':1},
...                                                           {'address':'6.6.6.3','network_value
↪':'6.6.6.0/24','nodeid':2}])
```

See *smc.core.engines.MasterEngineCluster* for more info

7.2 Nodes

Managed engines have many options for controlling the behavior of the device or virtual through the SMC API. Once an engine has been created, The engine is represented with 'nodes' that map to the individual firewall/IPS's. For example, a cluster will have 2 or more nodes.

Engine hierarchy resembles the following:

```
Engine
   | - ---> Node1
   | - ---> Node2
   | - ---> Node3
   \ - .... (up to 16)
```

Engine level commands allow operations like refresh policy, upload new policy, generating snapshots, export configuration, blacklisting, adding routes, route monitoring, and add or delete a physical interfaces

Some example engine level commands:

```
>>> engine = Engine('testfw')
>>> for node in engine.nodes:
>>> engine.generate_snapshot() #generate a policy snapshot
>>> engine.export(filename='/Users/davidlepage/export.xml') #generate policy export
>>> engine.refresh() #refresh policy
>>> engine.routing_monitoring()          #get route table status
....
```

For all available commands for engines, see *smc.core.engine.Engine*

Node level commands are specific commands targeted at individual nodes directly. In the case of a cluster, you can control the correct node by iterating *smc.core.engine.Engine.nodes* list.

Node level commands allow actions such as fetch license, bind license, initial contact, appliance status, go online, go offline, go standby, lock online, lock offline, reset user db, diagnostics, reboot, sginfo, ssh (enable/disable/change pwd), and time sync.

View nodes and reboot a node by name:

```
>>> engine = Engine('testfw')
>>> print(engine.nodes)
[Node(name=testfw node 1)]
...
>>> for node in engine.nodes:
...   if node.name == 'testfw':
...     node.reboot()
```

Bind license, then generate initial contact for each node for a specific engine:

```
>>> for node in engine.nodes:
...   node.initial_contact(filename='/Users/davidlepage/engine.cfg')
...   node.bind_license()
```

For all available commands for node, see *smc.core.node.Node*

7.3 Interfaces

After your engine has been successfully created with the default interfaces, you can add and remove interfaces as needed.

From an interface perspective, there are several different interface types that are have subtle differences. The supported physical interface types available are:

- Single Node Dedicated Interface (Single Layer 3 Firewall)
- Node Dedicated Interface (Used on Clusters, IPS, Layer 2 Firewall)
- Inline Interface (IPS / Layer2 Firewall)
- Capture Interface (IPS / Layer2 Firewall)
- Cluster Virtual Interface
- Virtual Physical Interface (used for Layer 3 Virtual Engines)
- Tunnel Interface

The distinction is subtle but straightforward. A single node interface is used on a single layer 3 firewall instance and represents a unique interface with dedicated IP Address.

A node dedicated interface is used on Layer 2 and IPS engines as management based interfaces and may also be used as a heartbeat (for example).

It is a unique IP address for each machine. It is not used for operative traffic in Firewall Clusters, IPS engines, and Layer 2 Firewalls. Firewall Clusters use a second type of interface, Cluster Virtual IP Address (CVI), for operative traffic.

IPS engines have two types of interfaces for traffic inspection: the Capture Interface and the Inline Interface. Layer 2 Firewalls only have Inline Interfaces for traffic inspection.

Note: When creating your engine instance, the correct type/s of interfaces are created automatically without having to specify the type. However, this may be relavant when adding interfaces to an existing device after creation.

To access interface information on existing engines, or to add to an existing engine, you must obtain the engine context object. It is not required to know the engine type (layer3, layer2, ips) as you can load by the parent class `smc.core.engines.Engine`.

For example, if I know I have an engine named 'myengine' (despite the engine 'role'), it can be obtained via:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('sg_vm')
>>> print(engine.nodes)
[Node(name=ngf-1065), Node(name=ngf-1035)]
```

It is not possible to add certain interface types based on the node type. For example, it is not possible to add inline or capture interfaces to layer 3 FW engines. This is handled automatically and will raise an exception if needed.

Adding interfaces are handled by property methods on the engine class.

To add a single node interface to an existing engine as Interface 10:

```
>>> engine = Engine('sg_vm')
>>> engine.physical_interface.add_single_node_interface(10, '33.33.33.33', '33.33.33.
↪0/24')
```

Node Interface's are used on IPS, Layer2 Firewall, Virtual and Cluster Engines and represent either a single interface or a cluster member interface used for communication.

To add a node interface to an existing engine:

```
>>> engine = Engine('sg_vm')
>>> engine.physical_interface.add_node_interface(10, '32.32.32.32', '32.32.32.0/24')
```

Inline interfaces can only be added to Layer 2 Firewall or IPS engines. An inline interface consists of a pair of interfaces that do not necessarily have to be contiguous. Each inline interface requires that a 'logical interface' is defined. This is used to identify the interface pair and can be used to simplify policy. See *smc.elements.other.LogicalInterface* for more details.

To add an inline interface to an existing engine:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('sg_vm')
...
>>> from smc.elements.helpers import logical_intf_helper
>>> logical_interface = logical_intf_helper('MyLogicalInterface') #get logical␣
↪interface reference
>>> engine.physical_interface.add_inline_interface('5-6', logical_interface_
↪ref=logical_intf)
```

Note: Use `smc.elements.helpers.logical_intf_helper('name')()` to find the existing logical interface reference by name or create it automatically

Capture Interfaces are used on Layer 2 Firewall or IPS engines as SPAN interfaces.

To add a capture interface to a layer2 FW or IPS:

```
>>> logical_interface = logical_intf_helper('MyLogicalInterface')
>>> engine = Engine('myengine')
>>> engine.physical_interface.add_capture_interface(10, logical_interface_ref=logical_
↪interface)
```

Cluster Virtual Interfaces are used on clustered engines and require a defined "CVI" (sometimes called a 'VIP'), as well as node dedicated interfaces for the engine initiated communications. Each clustered interface will therefore have 3 total address for a cluster of 2 nodes.

To add a cluster virtual interface on a layer 3 FW cluster with a zone:

```
>>> engine = Engine('myengine')
>>> engine.physical_interface.add_cluster_virtual_interface(
...                           interface_id=1,
...                           cluster_virtual='5.5.5.1',
...                           cluster_mask='5.5.5.0/24',
...                           macaddress='02:03:03:03:03:03',
...                           nodes=[{'address':'5.5.5.2', 'network_value':'5.5.5.0/
↪24', 'nodeid':1},
```

(continues on next page)

(continued from previous page)

```
...                                     {'address':'5.5.5.3', 'network_value':'5.5.5.0/
↪24', 'nodeid':2},
...                                     {'address':'5.5.5.4', 'network_value':'5.5.5.0/
↪24', 'nodeid':3}],
...                                zone_ref=zone_helper('Heartbeat'))
```

Warning: Make sure the cluster virtual netmask matches the node level networks

Nodes specified are the individual node dedicated addresses for the cluster members.

VLANs can be applied to layer 3 or inline interfaces. For inline interfaces, these will not have assigned IP addresses, however layer 3 interfaces will require addressing.

To add a VLAN to a generic physical interface for single node (layer 3 firewall) or a node interface, independent of engine type:

```
>>> engine = Engine('myengine')
>>> engine.physical_interface.add_vlan_to_node_interface(23, 154)
>>> engine.physical_interface.add_vlan_to_node_interface(23, 155)
>>> engine.physical_interface.add_vlan_to_node_interface(23, 156)
```

This will add 3 VLANs to physical interface 23. If this is a layer 3 routed firewall, you may still need to add addressing to each VLAN.

Note: In the case of Virtual Engines, it may be advisable to create the physical interfaces with VLANs on the Master Engine and allocate the IP addressing scheme to the Virtual Engine.

To add layer 3 interfaces with a VLAN and IP address:

```
>>> engine = Engine('myengine')
>>> engine.physical_interface.add_ipaddress_to_vlan_interface(
...                                         interface_id=2,
...                                         address='3.3.3.3',
...                                         network_value='3.3.3.0/24',
...                                         vlan_id=3,
...                                         zone_ref=zone_helper('Internal')
```

Note: The physical interface will be created if it doesn't already exist

When adding VLANs to a cluster interface, there are multiple options. Adding a VLAN, then adding a CVI interface, adding a VLAN and only NDI interfaces, adding VLAN with CVI and NDI or adding a simple VLAN with no interfaces.

Add a cluster interface with id 2, vlan 2, with no interfaces:

```
engine.physical_interface.add_ipaddress_and_vlan_to_cluster(
                                interface_id=2, vlan_id=2)
```

Add a cluster interface with id 2, vlan 2 and a single CVI interface with no macaddress (exempts this interface from load balancing:

```
engine.physical_interface.add_ipaddress_and_vlan_to_cluster(
                                    interface_id=2, vlan_id=2,
                                    cluster_virtual='3.3.3.1',
                                    cluster_mask='3.3.3.0/24',
                                    macaddress=None)
```

Add a cluster interface with id 2, vlan 2, single CVI interface and macaddress to allow load balancing. Set cluster mode to 'packetdispatch':

```
engine.physical_interface.add_ipaddress_and_vlan_to_cluster(
                                    interface_id=2, vlan_id=2,
                                    nodes=None, cluster_virtual='22.22.22.22',
                                    cluster_mask='22.22.22.0/24',
                                    macaddress='02:02:02:02:02:02',
                                    cvi_mode='packetdispatch')
```

Add a cluster interface with id 2, vlan 2, a CVI, NDI interfaces along with an assigned macaddress and zone:

```
engine.physical_interface.add_ipaddress_and_vlan_to_cluster(
                                    interface_id=2, vlan_id=2,
                                    nodes=[{'address': '4.4.4.4', 'network_value': '4.
↪4.4.0/24', 'nodeid':1},
                                           {'address': '4.4.4.5', 'network_value': '4.
↪4.4.0/24', 'nodeid':2}]
                                    cluster_virtual='4.4.4.1',
                                    cluster_mask='4.4.4.0/24',
                                    macaddress='02:02:02:02:02:02',
                                    cvi_mode='packetdispatch',
                                    zone_ref=zone_helper('thiszone'))
```

To add VLANs to layer 2 or IPS inline interfaces:

```
>>> logical_interface = logical_intf_helper('default_eth') #find logical intf or␣
↪create it
...
>>> engine = Engine('myengine')
>>> engine.physical_interface.add_vlan_to_inline_interface(interface_id='5-6',
...                                                        vlan_id=56,
...                                                        logical_interface_
↪ref=logical_interface)
...
>>> engine.physical_interface.add_vlan_to_inline_interface(interface_id='5-6',
...                                                        vlan_id=57,
...                                                        logical_interface_
↪ref=logical_interface)
...
>>> engine.physical_interface.add_vlan_to_inline_interface(interface_id='5-6',
...                                                        vlan_id=58,
...                                                        logical_interface_
↪ref=logical_interface)
```

Note: The physical interface will be created if it doesn't already exist

To see additional information on interfaces, *smc.core.interfaces* reference documentation

7.3.1 Sub-Interface and VLAN

Top level interface types hold basic settings about the interface, and sub-interfaces define the actual configuration itself, such as IP Addresses, Netmask, which node the interface is assigned to, etc. To obtain more information about a given interface such as sub-interfaces or vlans, use the interface vlan_interfaces() and sub_interfaces() resources.

To show all vlan interfaces:

```
>>> for interface in engine.interface.all():
...   if interface.has_vlan:
...     print(interface.vlan_interfaces())
[PhysicalVlanInterface(address=None,vlan_id=14), PhysicalVlanInterface(address=45.45.
↪45.50,vlan_id=13)]
```

Interfaces that have IP addresses assigned are considered 'sub interfaces'. There may be multiple sub interfaces on a given physical interface if multiple IP's are assigned.

Display addresses for a specific interface (showing the sub-interfaces):

```
>>> for interface in engine.interface.all():
...   if interface.name == 'Interface 0':
...     print(interface.sub_interfaces())
[SingleNodeInterface(name=172.18.1.55)]
```

It is not required to traverse the physical or sub-interface hierarchy to view properties of an interface.

Show IP addresses and networks for all interfaces:

```
>>> for interface in engine.interface.all():
...   print(interface.name, interface.addresses)
('Tunnel Interface 2001', [('169.254.9.22', '169.254.9.20/30', '2001')])
('Tunnel Interface 2000', [('169.254.11.6', '169.254.11.4/30', '2000')])
('Interface 2', [('192.168.1.252', '192.168.1.0/24', '2'), ('192.168.1.253', '192.168.
↪1.0/24', '2')])
('Interface 1', [('10.0.0.254', '10.0.0.0/24', '1'), ('10.0.0.253', '10.0.0.0/24', '1
↪'), ('10.0.0.252', '10.0.0.0/24', '1')])
('Interface 0', [('172.18.1.254', '172.18.1.0/24', '0'), ('172.18.1.252', '172.18.1.0/
↪24', '0'), ('172.18.1.253', '172.18.1.0/24', '0')])
```

See *smc.core.interfaces.Interface* for more info.

7.3.2 Modifying Interfaces

To modify an existing interface, you will first need to obtain a reference to the interface. There are some modifications that may have dependencies on other settings. For example, when an interface is configured with an IP address, the SMC will automatically create a route entry mapping that physical interface to the directly connected network. Changing the IP will leave the old network definition from the previously assigned interface and would also need to be removed.

Note: Save must be called on the interface itself or changes will only be made to a local copy of the element.

Example of changing the IP address of an existing single node interface (for layer 3 firewalls):

```
>>> for interface in engine.interface.all():
...   if interface.name == 'Interface 0':
```

(continues on next page)

(continued from previous page)

```
...     for intf in interface.sub_interfaces():
...       intf.address = '172.18.1.60'
...     interface.save()
...
>>> intf = engine.interface.get(0)
>>> print(intf.addresses)
[('172.18.1.60', '172.18.1.0/24', '0')]
```

Change the zone on the top level Physical Interface:

```
>>> intf = engine.interface.get(0)
>>> intf.zone_ref=zone_helper('My New Zone')
>>> intf.save()
```

Change a VLAN on a single FW node under Interface 2:

```
>>> intf = engine.interface.get(2)
>>> for vlan in intf.vlan_interfaces():
...   if vlan.vlan_id == '14':
...     vlan.vlan_id = '15'
...     intf.save()
```

7.3.3 Deleting Interfaces

Deleting interfaces by referencing the interface from the engine context.

Once you have loaded the engine, you can display all available interfaces by calling using the engine level property interface: *smc.core.engine.Engine.interface()* to view all interfaces for the engine.

The name of the interface is the name the NGFW gives the interface based on interface index. For example, physical interface 1 would be "Interface 1" and so on.

Viewing all interfaces and removing one by id:

```
>>> engine = Engine('testfw')
>>> for interface in engine.interface.all():
...     print(interface)
...
PhysicalInterface(name=Interface 12)
TunnelInterface(name=Tunnel Interface 2000)
PhysicalInterface(name=Interface 10)
TunnelInterface(name=Tunnel Interface 1001)
TunnelInterface(name=Tunnel Interface 1000)
PhysicalInterface(name=Interface 20)
PhysicalInterface(name=Interface 11)
PhysicalInterface(name=Interface 40)
...
>>> intf = engine.interface.get(20)     #Get interface 20
>>> print(intf.name)
Interface 20
...
>>> intf.delete()       #Delete interface
```

To see additional information on interfaces, *smc.core.interfaces* reference documentation

7.4 Routing

Adding routes to routed interfaces is done by loading the engine and providing the next hop gateway and destination network as parameters. It is not necessary to specify the interface to place the route, the mapping will be done automatically on the SMC based on the existing IP addresses and networks configured on the engine.

Show routes, and view specific interface details:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('testfw')
>>> for routes in engine.routing.all():
...   print(routes)
...
Routing(name=Interface 1,level=interface)
Routing(name=Tunnel Interface 1000,level=interface)
Routing(name=Interface 11,level=interface)
Routing(name=Tunnel Interface 2000,level=interface)
Routing(name=Interface 10,level=interface)
```

Details of interface 1 routes:

```
>>> for routes in engine.routing.all():
...   if routes.name == 'Interface 1':
...     print(routes.all())
...
[Routing(name=network-1.1.1.0/24,level=network), Routing(name=network-2.2.2.0/24,
↪level=network)]
```

Add a route. It is not required to specify the interface in which to add the route, the gateway will determine the interface as it is required to be directly connected:

```
>>> engine = Engine('master-eng')
>>> engine.add_route(gateway='172.18.1.200', network='192.168.17.0/24')
```

7.5 Licensing

Stonesoft engine licensing for physical appliances is done by having the SMC 'fetch' the license POS from the appliance and auto-assign the license. If the engine is running on a platform that doesn't have a POS (Proof-of-Serial) such as a virtual platform, then the fetch will fail. In this case, it is possible to do an auto bind which will look for unassigned dynamic licenses available in the SMC.

Example of attempting an auto-fetch and falling back to auto binding a dynamic license:

```
>>> engine = Engine('testfw')
>>> for node in engine.nodes:
...   node.bind_license()
```

CHAPTER 8

Policies

Policies are available for all 3 firewall roles, Firewall, Layer2 and IPS. The only initial requirement to create a policy is to reference a policy template. The policy template is a pre-configured set of best practice rules that provide connectivity and enables basic features such as stateful inspection, etc.

Obtaining available templates can be achieved through the collections interface:

```
>>> from smc.policy.layer3 import FirewallTemplatePolicy
>>> FirewallTemplatePolicy.objects.all()
>>> print(list(FirewallTemplatePolicy.objects.all()))
[FirewallTemplatePolicy(name=Firewall Inspection Template),␣
↪FirewallTemplatePolicy(name=Firewall Template)]
```

Example of creating a basic layer 3 policy; reference template by name:

```
>>> from smc.policy.layer3 import FirewallPolicy
>>> FirewallPolicy.create('newpolicy', template='Firewall Template')
FirewallPolicy(name=newpolicy)
```

Loading an existing policy is similar to obtaining other elements:

```
>>> policy = FirewallPolicy('newpolicy')
>>> policy.template
FirewallTemplatePolicy(name=Firewall Template)
```

Once a policy instance has been obtained, rules (policy or NAT) can be added, viewed, or removed.

Example of creating a rule for a firewall policy:

```
>>> policy.fw_ipv4_access_rules.create(name='newrule', sources='any', destinations=
↪'any', services='any', action='permit')
'http://1.1.1.1:8082/6.1/elements/fw_policy/265/fw_ipv4_access_rule/2099472'

#View all rules
>>> for rule in policy.fw_ipv4_access_rules.all():
...   print(rule.name, rule.sources, rule.destinations, rule.services)
```

(continues on next page)

(continued from previous page)

```
...
('newrule', <smc.policy.rule_elements.Source object at 0x1050d3b50>, <smc.policy.rule_
↪elements.Destination object at 0x1050d3dd0>, <smc.policy.rule_elements.Service␣
↪object at 0x1050d3f50>)
```

NAT can be applied as dynamic source NAT, static source NAT, or static destination NAT.

Example of creating a dynamic source NAT rule:

```
>>> from smc.policy.layer3 import FirewallPolicy
>>> from smc.elements.network import Host
>>> policy = FirewallPolicy('newpolicy')
>>> policy.fw_ipv4_nat_rules.create(name='mynat',
...                                 sources=[Host('kali')],
...                                 destinations='any',
...                                 services='any',
...                                 dynamic_src_nat='1.1.1.1',
...                                 dynamic_src_nat_ports=(1024,65535))
'http://1.1.1.1:8082/6.1/elements/fw_policy/265/fw_ipv4_nat_rule/2099475'
```

Example of creating a destination NAT rule where the destination is to Host('3.3.3.3') and will be translated to '1.1.1.1':

```
>>> policy.fw_ipv4_nat_rules.create(name='mynat',
...                                 sources='any',
...                                 destinations=[Host('3.3.3.3')],
...                                 services='any',
...                                 static_dst_nat='1.1.1.1')
'http://1.1.1.1:8082/6.1/elements/fw_policy/265/fw_ipv4_nat_rule/2099476'
```

Create an any/any no NAT rule (no value for NAT field):

```
>>> policy.fw_ipv4_nat_rules.create(name='nonat',sources='any',destinations='any',
↪services='any')
'http://1.1.1.1:8082/6.1/elements/fw_policy/265/fw_ipv4_nat_rule/2099477'
```

For additional NAT related options, see: *smc.policy.rule_nat.IPv4NATRule*

CHAPTER 9

VPN

It is possible to create VPN policy, all gateway elements and configurations related to Policy Based VPN. Gateway's in the VPN configuration can be either managed engines or remote gateawys (ExternalGateway).

There are several components or terminology required to set up a VPN.

- External Gateway: Non-SMC managed VPN endpoint
- External Endpoint: VPN Endpoint/s defined in external gateway (IP addresses, profiles)
- Sites: sites define the protected network/s for both sides of the VPN
- Internal Gateway: SMC managed layer 3 engine.

When creating a VPN to a non-managed device, an external gateway is required. This is a container object used to encapsulate the remote endpoints where the VPN will terminate:

```
>>> gateway = ExternalGateway.create('remoteside')
```

An external endpoint specifies the IP address settings and other VPN specific settings for the external gateway.

Create the external endpoint from the gateway resource:

```
>>> gateway.external_endpoint.create(name='remoteendpoint', address='2.2.2.2')
'http://1.1.1.1:8082/6.1/elements/external_gateway/22961/external_endpoint/26740'
```

Lastly, 'sites' need to be configured that identify the network/s for the external gateway side of the VPN. You can use pre-existing network elements, or create new ones as in the example below.

```
>>> network = Network('internal-network')
>>> print(network.href)
http://1.1.1.1:8082/6.1/elements/network/17911
...
>>> gateway.vpn_site.create('remote-site', [network.href])
'http://1.1.1.1:8082/6.1/elements/external_gateway/22961/vpn_site/22994'
```

Retrieve the engine internal gateway resource for the managed engine by obtaining the engine context.

```
>>> engine = Engine('testfw')
>>> print(engine.internal_gateway.href) #Internal gateway resource
http://1.1.1.1:8082/6.1/elements/single_fw/39550/internal_gateway/11476
```

Create the VPN Policy and apply the internal gateway as the 'Central Gateway' and the ExternalGateway as the 'Satellite Gateway':

```
>>> vpn = PolicyVPN.create(name='myVPN', nat=True)
>>> print(vpn.name, vpn.vpn_profile)
(u'myVPN', u'http://172.18.1.150:8082/6.1/elements/vpn_profile/2')
...
>>> vpn.open()
>>> vpn.add_central_gateway(engine.internal_gateway.href)
>>> vpn.add_satellite_gateway(external_gateway.href)
>>> vpn.save()
>>> vpn.close()
```

Note: You must call *smc.vpn.policy.PolicyVPN.open()* before modifications can be made. You also must call *smc.vpn.policy.PolicyVPN.save()* and *smc.vpn.policy.PolicyVPN.close()*

See API Reference documentation for more details.

CHAPTER 10

Administration

Administration provides an interface to system level admin tasks such as creating administrators, updating SMC with dynamic updates or engine upgrades, running tasks, etc.

10.1 Administrators

Creating administrators and modifying settings can be done using the *smc.elements.user.AdminUser* class.

For example, to create a user called 'administrator' and modify after creation, do:

Create admin:

```
AdminUser.create('administrator')
```

To modify after creation by setting a password and making a superuser:

```
admin = AdminUser('administrator') # Load an admin user called administrator
admin.change_password('mynewpassword')
admin.update(superuser=True) # ad-hoc update of attribute
admin.enable_disable() #enable or disable account
```

10.2 Tasks

Tasks may be generated by methods within certain classes, for example, many classes support an export() method. This is an asynchronous task that generates a 'follower' link to the task.

It is possible to monitor those asynchronous operations seperately from the direct method call by getting the follower href and using `smc.actions.tasks.TaskMonitor` or `smc.actions.tasks.TaskDownload` classes.

For example, fire off a policy update on an engine and get the asynchronous follower href:

```
engine = Engine('myfw')
task_follower = engine.refresh(wait_for_finish=True) #This isn't required as engine␣
↪will still refresh
while not task_follower.done():
    task_follower.wait(3)
print("Did task succeed: %s" % task_follower.success)
print("Last message from task: %s" % task_follower.last_message)
```

10.3 System

System level tasks include operations such as checking for and downloading a new dynamic update, engine upgrades, last activated package, SMC version, SMC time, emptying the trash bin, viewing all license details, importing, exporting elements and submitting global blacklist entries.

To view any available update packages:

```
    from smc.administration.system import System
    system = System()
available_packages = system.update_package()
print(list(available_packages))
```

To fully download and activate a dynamic update:

```
    system = System()
    available_packages = system.update_package()

my_dynup = available_packages.get_contains('1097')

if my_dynup.state.lower() == 'available':
    download_task = my_dynup.download(wait_for_finish=True)
    while not download_task.done():
        download_task.wait(3)
        print(download_task.last_message())
    if download_task.success:
        print("Success!")

# We are now downloaded, so activate
activation = my_dynup.activate(wait_for_finish=True)
while not activation.done():
    activation.wait(3)
    print(activation.last_message())

if activation.success:
    print("We are now activated")
else:
    print("Something bad went wrong: %s" % activation.last_message())
```

Empty the trash bin:

```
system = System()
system.empty_trash_bin()
```

CHAPTER 11

Logging

The smc-python API uses python logging for INFO, ERROR and DEBUG logging levels. If needed, add the following to your classes:

```
import logging
logging.getLogger()
logging.basicConfig(level=logging.ERROR, format='%(asctime)s %(levelname)s:
↪%(message)s')
```

Note: This is a recommended setting initially as it enables detailed logging of each call as it is processed through the API. It also includes the backend web based calls initiated by the requests module.

If you simply require stream logging to console for scripts, from your script import the smc module set_stream_logger, debug level, and optional format string conforming to the logging module:

```
from smc import set_stream_logger
set_stream_logger(level=logging.DEBUG, format_string=None)
```

CHAPTER 12

Extensions

smc-python provides additional extensions to extend the base library. Extensions are installed as separate packages and will have the dependency on the base smc-python library.

Available extensions:

- smc-python-monitoring

12.1 smc-python-monitoring

smc-python-monitoring API provides a monitoring interface to the SMC to perform queries for dynamic engine components such as blacklists, connections, routes, vpn's, users and logs.

Capabilities in the API implement the functionality found in the SMC Log Viewer and engine level monitoring.

12.1.1 Query

A Query is the top level object used to construct parameters to make queries to the SMC.

Query is the parent class for all monitors in package *smc_monitoring.monitors*

Each monitor type will have it's own predefined set of log fields that are considered 'default' for the query type. These will correlate closely to the default fields you will see in the SMC when viewing the same information (Connections, VPN SAs, Blacklist, etc).

Each query also has a specific formatter which defines how the data is returned from the query. Formatters are defined in *smc_monitoring.models.formats*.

Each formatter type allows customization of the field_format and allows a value of 'pretty', 'name' or 'id'. By default 'pretty' is used as the format which aligns with the column names in the SMC monitoring views.

class smc_monitoring.models.query.**Query**(*definition=None*, *target=None*, *format=None*, ***sockopt*)

Query is the top level structure for controlling requests over the SMC websocket protocol. Any keyword argu-

ments are passed through from inheriting classes are passed through as socket options for `smc_monitoring.wsocket.SMCSocketProtocol`.

Variables

- **request** (*dict*) – built request, eventually sent to socket
- **format** (`TextFormat`) – format settings for query

add_and_filter(**values*)

Add a filter using "AND" logic. This filter is useful when requiring multiple matches to evaluate to true. For example, searching for a specific IP address in the src field and another in the dst field.

See also:

smc_monitoring.models.filters.AndFilter for examples.

Parameters **values** – optional constructor args for *smc_monitoring.models.filters.AndFilter*. Typically this is a list of InFilter expressions.

Type list(QueryFilter)

Return type *AndFilter*

add_defined_filter(**value*)

Add a DefinedFilter expression to the query. This filter will be considered true if the `smc.monitoring.values.Value` instance has a value.

See also:

smc_monitoring.models.filters.DefinedFilter for examples.

Parameters **value** (`Value`) – single value for the filter. Value is of type *smc_monitoring.models.values.Value*.

Type list(QueryFilter)

Return type *DefinedFilter*

add_in_filter(**values*)

Add a filter using "IN" logic. This is typically the primary filter that will be used to find a match and generally combines other filters to get more granular. An example of usage would be searching for an IP address (or addresses) in a specific log field. Or looking for an IP address in multiple log fields.

See also:

smc_monitoring.models.filters.InFilter for examples.

Parameters **values** – optional constructor args for *smc_monitoring.models.filters.InFilter*

Return type *InFilter*

add_not_filter(**value*)

Add a filter using "NOT" logic. Typically this filter is used in conjunction with and AND or OR filters, but can be used by itself as well. This might be more useful as a standalone filter when displaying logs in real time and filtering out unwanted entry types.

See also:

smc_monitoring.models.filters.NotFilter for examples.

Parameters **values** – optional constructor args for *smc_monitoring.models.filters.NotFilter*. Typically this is a list of InFilter expressions.

Type list(QueryFilter)

Return type *OrFilter*

add_or_filter(**values*)

Add a filter using "OR" logic. This filter is useful when matching on one or more criteria. For example, searching for IP 1.1.1.1 and service TCP/443, or IP 1.1.1.10 and TCP/80. Either pair would produce a positive match.

See also:

smc_monitoring.models.filters.OrFilter for examples.

Parameters **values** – optional constructor args for *smc_monitoring.models.filters.OrFilter*. Typically this is a list of InFilter expressions.

Type list(QueryFilter)

Return type *OrFilter*

add_translated_filter()

Add a translated filter to the query. A translated filter syntax uses the SMC expressions to build the filter. The simplest way to see the syntax is to create a filter in SMC under Logs view and right click->Show Expression.

See also:

smc_monitoring.models.filters.TranslatedFilter for examples.

Parameters **values** – optional constructor args for *smc_monitoring.models.filters.TranslatedFilter*

Type list(QueryFilter)

Return type *TranslatedFilter*

execute()

Execute the query with optional timeout. The response to the execute query is the raw payload received from the websocket and will contain multiple dict keys and values. It is more common to call query.fetch_XXX which will filter the return result based on the method. Each result set will have a max batch size of 200 records. This method will also continuously return results until terminated. To make a single bounded fetch, call *fetch_batch()* or *fetch_raw()*.

Parameters **sock_timeout** (*int*) – event loop interval

Returns raw dict returned from query

Return type dict(list)

fetch_as_element()

Each inheriting class will override this method if supported.

fetch_batch(*formatter=<class 'smc_monitoring.models.formatters.TableFormat'>*, ***kw*)

Fetch and return in the specified format. Output format is a formatter class in *smc_monitoring.models.formatters*. This fetch type will be a single shot fetch unless providing max_recv keyword with a value greater than the default of 1. Keyword arguments available are kw in *fetch_raw()*.

Parameters **formatter** – Formatter type for data representation. Any type in *smc_monitoring.models.formatters*.

Returns generator returning data in specified format

Note: You can provide your own formatter class, see *smc_monitoring.models.formatters* for more info.

---

fetch_live (*formatter=<class 'smc_monitoring.models.formatters.TableFormat'>*)
Fetch a live stream query. This is the equivalent of selecting the "Play" option for monitoring fields within the SMC UI. Data will be streamed back in real time.

Parameters **formatter** – Formatter type for data representation. Any type in *smc_monitoring.models.formatters*.

Returns generator yielding results in specified format

fetch_raw (***kw*)
Fetch the records for this query. This fetch type will return the results in raw dict format. It is possible to limit the number of receives on the socket that return results before exiting by providing max_recv.

This fetch should be used if you want to return only the result records returned from the query in raw dict format. Any other dict key/values from the raw query are ignored.

Parameters **max_recv** (*int*) – max number of socket receive calls before returning from this query. If you want to wait longer for results before returning, increase max_iterations (default: 0)

Returns list of query results

Return type list(dict)

static resolve_field_ids (*ids*, ***kw*)
Retrieve the log field details based on the LogField constant IDs. This provides a helper to view the fields representation when using different field_formats. Each query class has a default set of field IDs that can easily be looked up to examine their fields and different label options. For example:

```
Query.resolve_field_ids(ConnectionQuery.field_ids)
```

Parameters **ids** (*list*) – list of log field IDs. Use LogField constants to simplify search.

Returns raw dict representation of log fields

Return type list(dict)

update_filter (*filt*)
Update the query with a new filter.

Parameters **filt** (smc_monitoring.models.filters.QueryFilter) – change query to use new filter

update_format (*format*)
Update the format for this query.

Parameters **format** (*smc_monitoring.models.formats*) – new format to use for this query

12.1.2 Models

The models package consists of the building blocks that make up a query.

Each module represents different class models that simplify adding things like filters, specifying values and formats.

12.1.2.1 Filters

Filters are used by queries to refine how results are returned.

QueryFilter is the top level 'interface' for all filter types. The `filter` attribute of a QueryFilter provides access to the compiled query string used to build the filter. Each QueryFilter also has an `update_filter` method that can be used to swap new filters in and out of an existing query.

Filters can be added to queries using the add_XXX methods of the query, or by building the filters and adding to the query using query.update_filter(). Filters can be swapped in and out of a query.

Examples:

Build a query to return all records of alert severity high or critical:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.ALERTSEVERITY), [ConstantValue(Alerts.HIGH, Alerts.CRITICAL)])
```

If you prefer building your filters individually, it is not required to call the add_XX_filter methods of the query. You can also insert filters by building the filter and calling the `update_filter` method on the query:

```
query = LogQuery(fetch_size=50)
query.update_filter(
    InFilter(FieldValue(LogField.SERVICE), [ServiceValue('UDP/53', 'TCP/80')])
```

You can also replace existing query filters with new filters to re-use the base level query parameters such as fetch_size, format style, time/date ranges, etc.

Replace the existing query filter with a different filter:

```
new_filter = InFilter(FieldValue(LogField.SERVICE), [ServiceValue('UDP/53', 'TCP/80
↪')])
query.update_filter(new_filter)
```

Note: it is also possible to update a filter by calling query.add_XX_filter methods multiple times. Each time will replace an existing filter if it exists.

For example, calling add_XX_filter methods multiple times to refine filter results:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(    # First filter query - look for alert severity high and␣
↪critical
    FieldValue(LogField.ALERTSEVERITY), [ConstantValue(Alerts.HIGH, Alerts.CRITICAL)])

query.add_and_filter([    # Change filter to AND filter for further granularity
    InFilter(FieldValue(LogField.ALERTSEVERITY), [ConstantValue(Alerts.HIGH, Alerts.
↪CRITICAL)]),
    InFilter(FieldValue(LogField.SRC), [IPValue('192.168.4.84')])])
```

class smc_monitoring.models.filters.**AndFilter**(**filters*)

Bases: smc_monitoring.models.filters.QueryFilter

An AND filter combines other filter types and requires that each filter matches. An AND filter is a collection of QueryFilter's, typically IN or NOT filters that are AND'd together.

Example of fetching 50 records for sources matching '192.168.4.84' and a service of 'TCP/80':

```
query = LogQuery(fetch_size=50)
query.add_and_filter([
    InFilter(FieldValue(LogField.SRC), [IPValue('192.168.4.84')]),
    InFilter(FieldValue(LogField.SERVICE), [ServiceValue('TCP/80')])])
```

Parameters **filters** (*list or tuple*) – Any filter type in smc.monitoring.filters.

class smc_monitoring.models.filters.**CILikeFilter**
Bases: smc_monitoring.models.filters.QueryFilter

A CILikeFilter is a case insensitive LIKE string match filter.

class smc_monitoring.models.filters.**CSLikeFilter**
Bases: smc_monitoring.models.filters.QueryFilter

A CSLikeFilter is a case sensitive LIKE string match filter.

class smc_monitoring.models.filters.**DefinedFilter**(*value=None*)
Bases: smc_monitoring.models.filters.QueryFilter

A Defined Filter applied to a query will only match if the value specified has a value in the audit record/s.

Show only records that have a defined Action (read as 'match if action has a value'):

```
query = LogQuery(fetch_size=50)
query.add_defined_filter(FieldValue(LogField.ACTION))
```

DefinedFilter's can be used in AND, OR or NOT filter queries as well. Fetch the most recent 50 records for source 192.168.4.84 that have an application defined:

```
query = LogQuery(fetch_size=50)
query.add_and_filter([
    DefinedFilter(FieldValue(LogField.IPSAPPID)),
    InFilter(FieldValue(LogField.SRC), [IPValue('192.168.4.84')])])
```

Parameters **values** (Value) – single value type to require on filter

class smc_monitoring.models.filters.**InFilter**(*left*, *right*)
Bases: smc_monitoring.models.filters.QueryFilter

InFilter's are made up of two parts, a left and a right. An InFilter is considered a match if evaluation of the left part is equivalent to one of the elements of the right part. The left part of an InFilter is made up of a target of type smc.monitoring.values.Value. The right part is made up of a list of the same type.

Search the Source field for IP addresses 192.168.4.84 or 10.0.0.252:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.SRC), [IPValue('192.168.4.84', '10.0.0.252')])
```

Reverse the logic and search for IP address 192.168.4.84 in source and dest log fields:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    IPValue('192.168.4.84'), [FieldValue(LogField.SRC, LogField.DST)])
```

InFilter's are one of the most common filters and are often added to AND, OR or NOT filters for more specific matching.

Parameters

- **left** (Values: any value type in *smc_monitoring.models.values*) – single value for leftmost portion of filter
- **right** (list(Values): any value type in *smc_monitoring.models.values*) – list of values for rightmost portion of filter

class smc_monitoring.models.filters.**NotFilter**(**filters*)
Bases: smc_monitoring.models.filters.QueryFilter

A NOT filter provides the ability to suppress auditing based on a specific filter. A NOT filter is typically added to an AND filter to remove unwanted entries from the response.

Use only a NOT filter to a query and to ignore DNS traffic:

```
query = LogQuery(fetch_size=50)
query.add_not_filter(
    [InFilter(FieldValue(LogField.SERVICE), [ServiceValue('UDP/53')])])
```

The above example by itself is not overly useful, however you can use NOT filters with AND filters to achieve a logic like "Find source IP 192.168.4.68 and not service UDP/53 or TCP/80":

```
query = LogQuery(fetch_size=50)
not_dns = NotFilter(
    [InFilter(FieldValue(LogField.SERVICE), [ServiceValue('UDP/53', 'TCP/80')])])
by_ip = InFilter(
    FieldValue(LogField.SRC), [IPValue('172.18.1.20')])

query.add_and_filter([not_dns, by_ip])
```

Parameters **filters** (*list or tuple*) – Any filter type in smc.monitoring.filters.

class smc_monitoring.models.filters.**OrFilter**(**filters*)
Bases: smc_monitoring.models.filters.QueryFilter

An OR filter matches if any of the combined filters match. An OR filter is a collection of QueryFilter's, typically IN or NOT filters that are OR'd together.

Example of fetching 50 records for sources matching '192.168.4.84' or a service of 'TCP/80':

```
query = LogQuery(fetch_size=50)
query.add_or_filter([
    InFilter(FieldValue(LogField.SRC), [IPValue('192.168.4.84')]),
    InFilter(FieldValue(LogField.SERVICE), [ServiceValue('TCP/80')])])
```

Parameters **filters** (*list or tuple*) – Any filter type in smc.monitoring.filters.

class smc_monitoring.models.filters.**TranslatedFilter**
Bases: smc_monitoring.models.filters.QueryFilter

Translated filters use the SMC internal name alias and builds expressions to make more complex queries.

Example of using built in filter methods:

```
query = LogQuery(fetch_size=50)
query.format.timezone('CST')
query.format.field_format('name')
```

(continues on next page)

(continued from previous page)

```
translated_filter = query.add_translated_filter()
translated_filter.within_ipv4_network('$Dst', ['192.168.4.0/24'])
translated_filter.within_ipv4_range('$Src', ['1.1.1.1-192.168.1.254'])
translated_filter.exact_ipv4_match('$Src', ['172.18.1.152', '192.168.4.84'])
```

exact_ipv4_match (*field*, *values*)
An exact IPv4 address match on relevant address fields.

Parameters

- **field** (*str*) – name of field to filter on. Taken from 'Show Filter Expression' within SMC.
- **values** (*list*) – value/s to add. If more than a single value is provided, the query is modified to use UNION vs. ==
- **complex** (*bool*) – A complex filter is one which requires AND'ing or OR'ing values. Set to return the filter before committing.

within_ipv4_network (*field*, *values*)
This filter adds specified networks to a filter to check for inclusion.

Parameters

- **field** (*str*) – name of field to filter on. Taken from 'Show Filter Expression' within SMC.
- **values** (*list*) – network definitions, in cidr format, i.e: 1.1.1.0/24.

within_ipv4_range (*field*, *values*)
Add an IP range network filter for relevant address fields. Range (between) filters allow only one range be provided.

Parameters

- **field** (*str*) – name of field to filter on. Taken from 'Show Filter Expression' within SMC.
- **values** (*list*) – IP range values. Values would be a list of IP's separated by a '-', i.e. ['1.1.1.1-1.1.1.254']

12.1.2.2 Values

Values are used to provide searchable input for filters. Each value format is specific to the data type added to the filter. For example, an IPValue specifies IP's or network values that can be added to a filter from *smc_monitoring.models.filters*.

Each constructor can be initialized in the following ways:

Single value:

```
IPValue('1.1.1.1')
```

Multiple values:

```
IPValue('1.1.1.1', '2.2.2.2')
```

As a list of values:

```
i = ['1.1.1.1', '3.3.3.3']
IPValue(*i)
```

The `value` attribute of each *`Value`* stores the query string as a list that is absorbed by the filter.

class smc_monitoring.models.values.**ConstantValue**(**constants*)
Bases: *smc_monitoring.models.values.Value*

Constant values can be used for log field values. For example, specifying a filter by Action can be simplified by specifying the constant for the action value. Constant values are not used for log field names (use FieldValue instead).

Searching for all actions of discard and block:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.ACTION), [ConstantValue(Actions.DISCARD, Actions.BLOCK)])
```

Parameters **constants** (*list or str*) – constant values

class smc_monitoring.models.values.**ElementValue**(**elements*)
Bases: *smc_monitoring.models.values.Value*

Element Values are used when creating a filter for an element already defined in the SMC. The element can be referenced by it's type.

Search for a host element 'kali' in the 'source' log field:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.SRC), [ElementValue(Host('kali'))])
```

Parameters **elements** (*list or str*) – element definitions

---

Note: Using elements expands the search to potentially include a broader range of data. For example, a host can have multiple IP addresses, both ipv4 and ipv6.

---

class smc_monitoring.models.values.**FieldValue**(**fields*)
Bases: *smc_monitoring.models.values.Value*

FieldValue specifies a log field filter by either constant ID or name. The field name field is the internal name representation for the SMC. To find a given field name, within SMC go to Log Viewer and drag a field into the filter window, right click and select "Show Filter Expression".

Using field value as filter for InFilter type:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.SRC), [IPValue('192.168.4.84')])
```

Parameters **fields** (*list or str*) – fields definitions by name or int ID

Note: If using constant values, consult `smc.monitoring.constants.LogField` for valid attributes.

---

class smc_monitoring.models.values.**IPValue**(**addresses*)
Bases: *smc_monitoring.models.values.Value*

IP Values specify IP addresses used for searching.

Search for IP address in source and dest fields:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    IPValue('192.168.4.84'), [FieldValue(LogField.SRC, LogField.DST)])
```

Parameters **addresses** (*list or str*) – address definitions

class smc_monitoring.models.values.**ServiceValue**(**services*)
Bases: *smc_monitoring.models.values.Value*

Service Values allow searches on the service field. When specifying the service value, specify as <protocol/port>. For example, 'TCP/80', 'UDP/53'. For ICMP, specify as ICMP/Type/Code (Code is optional).

Search for any services with TCP port 80 and UDP port 53:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.SERVICE), [ServiceValue('TCP/80', 'UDP/53')])
```

Parameters **services** (*list or str*) – service definitions

class smc_monitoring.models.values.**StringValue**(**values*)
Bases: *smc_monitoring.models.values.Value*

String value match. Note that string matching can only be done on log fields that are of type string (no type conversions are done on non-string types). String matches are also exact.

Find all audits accessing URL play.googleapis.com:

```
query = LogQuery(fetch_size=50)
query.add_in_filter(
    FieldValue(LogField.HTTPREQUESTHOST),[StringValue('play.googleapis.com')])
```

Parameters **value** (*list*) – string to match

class smc_monitoring.models.values.**Value**(*values*)
Bases: object

Value is the topmost parent for all value types.

Variables ***value*** – stores value formatted into dict

12.1.2.3 Formats

Field formats represent a way to control the format of the returned data. By modifying a field format, you can control field level settings such as wther to resolve IP's via DNS, how to display field names and values and which fields to return in the query.

Each log format will return a different view type The most common and default for all queries is the *TextFormat* using a 'pretty' field format which is what you will see from the column data and values if using the SMC Log Viewer.

Return only a specific set of fields by id's:

```
query = LogQuery(fetch_size=5)
query.format.field_ids([
    LogField.TIMESTAMP, LogField.NODEID, LogField.SRC,
    LogField.DST, LogField.PROTOCOL, LogField.ACTION])
```

Return only a specific set of fields by name:

```
query = LogQuery(fetch_size=5)
query.format.field_names(['Src', 'Dst'])
```

Note: If both field_ids and field_names are provided, they will be merged.

class smc_monitoring.models.formats.**CombinedFormat**(***kw*)
Bases: object

CombinedFormat provides a way to specify different field resolvers based on field name or ID. Keyword arguments provided will define a unique key that represents the format object and value is the format object itself.

For example, using a combined filter to resolve the TIMESTMAP field in text format, but source and destination fields in detailed format:

```
text = TextFormat()
text.field_ids([LogField.TIMESTAMP])

detailed = DetailedFormat()
detailed.field_ids([LogField.SRC, LogField.DST])

combined = CombinedFormat(tformat=text, dformat=detailed)

query = LogQuery(fetch_size=1, format=combined)
```

After executing the query, the raw record results will be formatted as a list of dict's, which each record having a dict key equal to keyword argument input provided:

```
[{'dformat': {'Src Addr': '10.0.0.1', 'Dst Addr': '224.0.0.1'},
  'tformat': {'Creation Time': '2017-08-05 14:12:44'}},
 {'dformat': {'Src Addr': '10.0.0.1', 'Dst Addr': '224.0.0.1'},
  'tformat': {'Creation Time': '2017-08-05 14:12:44'}}]
  ...
```

The results can then be parsed and used to provide custom views as necessary.

Parameters **kw** – key word arguments should use an identifier key that will be present in the results, and a value which is a format object type in smc.monitoring.formats.

class smc_monitoring.models.formats.**DetailedFormat**(*field_format='pretty'*, ***kw*)
Bases: *smc_monitoring.models.formats.TextFormat*

Detailed format does not do a Log value conversion as the TextFormat would, however does provide a field map in the first payload with characteristics of the fields in the return data. This might be a useful format to obtain conversion ID's for specific fields or debugging.

class smc_monitoring.models.formats.**FormatFieldMixin**
Bases: object

Format field methods for modifying behavior of a query.

field_format(*name*)
Specify how the field name are printed in the response.

Parameters

- **id** (*str*) – as integer IDs from constants found in *smc_monitoring.models.constants.LogField*
- **name** (*str*) – as internal SMC names
- **pretty** (*str*) – pretty printed as you would see in the SMC UI

field_ids(*ids*)
Add filter to show only fields with given field ID's. Field ID's can be mapped to the LogField constants in *smc_monitoring.models.constants.LogField*

Note: Set the return display mode for the Log field name by using *field_format()*. The display value `name` will match the name of the LogField constant.

---

field_names(*names*)
Show only fields with given name. The `name` is the internal SMC name for the log field. The simplest way to obtain the name for a log field is from the SMC Log Viewer. Use the Log Viewer filter window to drag a column filter and select "Show Filter Expression".

..note:: The log field name is case sensitive and is typically using camelcase notation.

class smc_monitoring.models.formats.**RawFormat**(*field_format='pretty'*)
Bases: *smc_monitoring.models.formats.FormatFieldMixin*

Raw format is an abbreviated version of the detailed format. Fewer fields are provided and resolution of field values is not done.

class smc_monitoring.models.formats.**TextFormat**(*field_format='pretty'*, ***kw*)
Bases: *smc_monitoring.models.formats.FormatFieldMixin*

Text format with 'pretty' field formatting uses the same display to what you would see from the native SMC Log Viewer.

Keyword arguments can optionally be provided to set 'resolving' fields during instance creation, or they can be set on the instance afterwards by calling *set_resolving()*.

set_resolving(***kw*)
Certain log fields can be individually resolved. Use this method to set these fields. Valid keyword arguments:

Parameters

- **timezone** (*str*) – string value to set timezone for audits
- **time_show_zone** (*bool*) – show the time zone in the audit.
- **time_show_millis** (*bool*) – show timezone in milliseconds
- **keys** (*bool*) – resolve log field keys
- **ip_elements** (*bool*) – resolve IP's to SMC elements
- **ip_dns** (*bool*) – resolve IP addresses using DNS
- **ip_locations** (*bool*) – resolve locations

timezone(*tz*)
Set timezone on the audit records. Timezone can be in formats: 'US/Eastern', 'PST', 'Europe/Helsinki'

See SMC Log Viewer settings for more examples.

Parameters **tz** (*str*) – timezone, i.e. CST

12.1.2.4 Constants

Constants used within *smc_monitoring.models.values.Value* values to simplify referencing log viewer data.

class smc_monitoring.models.constants.**Actions**
Rule Actions

ALLOW = 1
Allowed

BLOCK = 13
Block

DISCARD = 0
Discard

DISCARD_PASSIVE = 4
Silent discard

PERMIT = 11
Permit the connection

REFUSE = 2
Reset

TERMINATE = 9
Terminate

TERMINATE_FAILED = 10
Failed terminating connection

TERMINATE_PASSIVE = 8
Silent terminate

TERMINATE_RESET = 12
Reset the connection

class smc_monitoring.models.constants.**Alerts**
Alert actions

CRITICAL = 10
Critical alert

HIGH = 5
High alert

INFO = 1
Info alert

LOW = 3
Low alert

class smc_monitoring.models.constants.**DataType**
Query by type of logs. This identifies which log types you are interested in filtering by, i.e. Audit, FW Logs, Third_Party, etc. Equivalent to the Query dropdown in SMC Log Viewer

class smc_monitoring.models.constants.**LogField**

Log field constants can be referenced when creating filters such as Field Values. i.e. FieldValue(LogField.SRC). Each constant name is identical to the value when using the field format type of 'name' (with exception that the constant names are in upper case).

ACCELAPSED = 104
Elapsed time of connection in seconds

ACCRXBYTES = 106
Number of bytes received during connection

ACCRXPACKETS = 139
Number of packets received during connection

ACCTXBYTES = 105
Number of bytes sent during connection

ACCTXPACKETS = 138
Number of packets sent during connection

ACK = 29
Acknowledged Alert

ACTION = 14
Connection action

ALERT = 25
Type of alert

ALERTCOUNT = 603
Alert count

ALERTERTRACE = 600
Alerter trace (events) information (datatype:4)

ALERTSEVERITY = 602
Severity of situation

ALERTSTATUS = 604
Alert status

ALLOWEDDATATAG = 482
Allowed data type tag

APPLICATION = 800
Application

APPLICATIONCOMBINATIONFLAGS = 54
Anomaly information of certain combination of network application and client application.

APPLICATIONDETAIL = 801
Application Detail

APPLICATIONUSAGE = 52
The type of the application that caused sending this event.

ASPAMEMAILMESSAGEID = 155
Email message-ID

ASPAMEMAILSCORE = 153
Email score value

ASPAMEMAILSUBJECT = 152
Email subject

ASPAMRECEIVEREMAIL = 151
Receiver email address

ASPAMSENDEREMAIL = 150
Sender email address

ASPAMSENDERMTA = 154
Sender Message Transfer Agent IP address

AUTHENTICATIONCOUNTER = 850
Authentication counters

AUTHMETHOD = 133
Authentication Method element

AUTHNAME = 108
User name of authorized user

AUTHRULEID = 107
The rule number of the rule that led to the log creation

BALANCINGPROBING = 397
BALANCING_PROBING

BALANCINGSELECTION = 392
BALANCING_SELECTION

BLACKLISTENTRYDESTINATIONIP = 120
Blacklist entry destination IP address

BLACKLISTENTRYDESTINATIONIPMASK = 121
Blacklist entry destination IP address mask

BLACKLISTENTRYDESTINATIONIPPREFIXLEN = 173
Blacklist entry destination IP address prefix length

BLACKLISTENTRYDESTINATIONPORT = 125
Blacklist entry destination port

BLACKLISTENTRYDESTINATIONPORTRANGE = 126
Blacklist entry destination port range end

BLACKLISTENTRYDURATION = 127
Blacklist entry duration

BLACKLISTENTRYID = 117
None

BLACKLISTENTRYPROTOCOL = 122
Blacklist entry IP protocol

BLACKLISTENTRYSOURCEIP = 118
Blacklist entry source IP address

BLACKLISTENTRYSOURCEIPMASK = 119
Blacklist entry source IP address mask

BLACKLISTENTRYSOURCEIPPREFIXLEN = 172
Blacklist entry source IP address prefix length

BLACKLISTENTRYSOURCEPORT = 123
Blacklist entry source port

BLACKLISTENTRYSOURCEPORTRANGE = 124
Blacklist entry source port range end

BLACKLISTER = 128
Blacklister

CIPHERALG = 536
Cipher algorithm

CLIENTIPADDRESS = 403
Address of client causing event

COMPID = 3
The identifier of the creator of the log entry.

CONNDIRECTION = 310
Connection direction

CONNECTEDMACADDR = 447
Connected MAC addresses

CONNECTIVITY = 306
Connectivity

CONNSTATUS = 309
Connection status

CONNTYPE = 308
Connection type

CONTAINEDDATATAG = 483
Contained data type tag

CONTROLCOMMANDID = 28
None

DATATAG = 481
Data type tag

DATATAGS = 485
Data tags concerning the record

DATATYPE = 34
Data type

DHCPLEASEEXPIRES = 528
DHCP_LEASE_EXPIRES

DHCPLEASEGW = 529
DHCP_LEASE_GW

DHCPLEASEIP = 530
DHCP_LEASE_IP

DHCPLEASENETMASK = 531
DHCP_LEASE_NETMASK

DHCPLEASEPREFIXLEN = 498
DHCP_LEASE_PREFIXLEN

DHCPLEASERECEIVED = 532
DHCP_LEASE_RECEIVED

DHCPLEASES = 527
DHCP_LEASES

DPD = 543
Dead Peer Detection

DPORT = 10
Connection destination protocol port

DSCPMARK = 130
DSCP Mark

DST = 8
Connection destination IP address

DSTADDRS = 20008
Destination addresses

DSTIF = 13
Destination interface of firewall

DSTIPRANGE = 526
Destination IP Range

DSTVLAN = 113
Destination VLAN

DSTZONE = 47
Connection destination interface zone

ELEMENTDOMAIN = 415
Administrative Domain of Associated Element

ENDPOINT = 504
Local VPN end point

ENTERPRISEOID = 493
Enterprise OID

EVENT = 6
Logged event

EVENTADDRESS = 705
Notification destination

EVENTINFO = 701
Description for event

EVENTLOGID = 702
Data Identifier of the alert

EVENTTIME = 700
Time stamp of the alert

EVENTTYPE = 703
Type of event

EVENTUSER = 704
User who executed the action

EXPIRATIONTIME = 534
VPN SA expiration time

FACILITY = 22
Firewall subsystem

FILETYPECOMPAT = 56
The type of the file that caused sending this event.

FLAG = 114
None

FPCACHED = 57
Fingerprint match came from fingerprinting cache.

FW100INTERFACE = 431
FW100 Interface

FW100TRAFFICCOUNTERS = 430
Fw100 Traffic counters

FWACCEPTEDBYTES = 326
FW_ACCEPTED_BYTES

FWACCEPTEDPACKETS = 327
FW_ACCEPTED_PACKETS

FWACCOUNTEDBYTES = 336
FW_ACCOUNTED_BYTES

FWACCOUNTEDPACKETS = 337
FW_ACCOUNTED_PACKETS

FWADSLRXBYTES = 417
FW_ADSL_RX_BYTES

FWADSLTXBYTES = 416
FW_ADSL_TX_BYTES

FWDECRYPTEDBYTES = 332
FW_DECRYPTED_BYTES

FWDECRYPTEDPACKETS = 333
FW_DECRYPTED_PACKETS

FWDROPPEDBYTES = 328
FW_DROPPED_BYTES

FWDROPPEDPACKETS = 329
FW_DROPPED_PACKETS

FWENCRYPTEDBYTES = 330
FW_ENCRYPTED_BYTES

FWENCRYPTEDPACKETS = 331
FW_ENCRYPTED_PACKETS

FWFORWARDEDBYTES = 419
FW_FORWARDED_BYTES

FWFORWARDEDPACKETS = 418
FW_FORWARDED_PACKETS

FWINTERFACEKEY = 340
FW_INTERFACE_KEY

FWNATTEDBYTES = 334
FW_NATTED_BYTES

FWNATTEDPACKETS = 335
FW_NATTED_PACKETS

FWRECEIVEDBYTES = 322
FW_RECEIVED_BYTES

FWRECEIVEDPACKETS = 323
FW_RECEIVED_PACKETS

FWSENTBYTES = 324
FW_SENT_BYTES

FWSENTPACKETS = 325
FW_SENT_PACKETS

FWTRAFFIC = 342
FW Traffic

FWTRAFFICACCOUNTEDBYTES = 352
Accounted Bytes

FWTRAFFICACCOUNTEDPACKETS = 346
Accounted Packets

FWTRAFFICALLOWEDBYTES = 349
Allowed Bytes

FWTRAFFICALLOWEDPACKETS = 343
Allowed Packets

FWTRAFFICDISCARDEDBYTES = 350
Discarded Bytes

FWTRAFFICDISCARDEDPACKETS = 344
Discarded Packets

FWTRAFFICENCRYPTEDBYTES = 354
Encrypted Bytes

FWTRAFFICENCRYPTEDPACKETS = 348
Encrypted Packets

FWTRAFFICLOGGEDBYTES = 351
Logged Bytes

FWTRAFFICLOGGEDPACKETS = 345
Logged Packets

FWTRAFFICNATTEDBYTES = 353
Natted Bytes

FWTRAFFICNATTEDPACKETS = 347
Natted Packets

GENERICTRAPTYPE = 494
Generic Trap Type

HASHALG = 538
Hash Algorithm

HITS = 48
HITS

HTTPREQUESTHOST = 1586
HTTP request host

ICMPCODE = 101
ICMP code attribute

ICMPID = 102
ICMP identifier

ICMPTYPE = 100
ICMP type attribute

IKEDHGROUP = 901
Diffie-Hellman Group

IKELOCALID = 540
Local IKE ID

IKEREMOTEID = 541
Remote IKE ID

IKEV1MODE = 542
IKEv1 negotiation mode

INCIDENTCASE = 411
Incident Case

INFOMSG = 19
Information Message

INTERFACE = 35
Interface

IPCOMPRESSION = 546
IP Compression

IPSAPPID = 134
Network application detected in the connection

IPSECSSPI = 103
Inbound IPsec SPI value (hexadecimal)

LOGID = 2
Data Identifier

LOGIFTOPDESTINATIONIPADDRS = 446
Amount of traffic flowing to the most used destination IP addresses per logical interface

LOGIFTOPSOURCEIPADDRS = 445
Amount of traffic originating from the most used source IP addresses per logical interface

LOGIFTOPTCPDESTINATIONPORTS = 443
Amount of traffic on the most used TCP destination ports per logical interface

LOGIFTOPUDPDESTINATIONPORTS = 444
Amount of traffic on the most used UDP destination ports per logical interface

LOGSEVERITY = 805
Severity

LONGMSG = 601
Long field description of alert

MACALG = 537
MAC Algorithm

MESSAGEID = 804
Message Id

NATBALANCEID = 393
NAT_BALANCE_ID

NATDPORT = 18
Translated packet destination port

NATDST = 16
Translated packet destination IP address

NATMAPID = 394
NAT_MAP_ID

NATRULEID = 21
The rule number of the rule that led to the log creation

NATSPORT = 17
Translated packet source protocol port

NATSRC = 15
Translated packet source IP address

NATT = 544
NAT Traversal

NEGOTIATIONROLE = 539
SA Negotiation Role

NODECAPACITY = 321
Capacity

NODECONFIGURATION = 304
Current configuration

NODECONFIGURATIONTIMESTAMP = 305
Configuration upload time

NODEDYNUP = 303
Update package level

NODEHWSTATUS = 315
Node hardware status

NODEID = 4
Firewall or server node that passes this information

NODELOAD = 320
Node load

NODESTATUS = 300
Node status

NODEVERSION = 301
Node version

NONCONTAINEDDATATAG = 484
Non-contained data type tag

NUMALERTRESPONSES = 365
Number of alert responses performed by this engine

NUMBLACKLISTRESPONSES = 369
Number of blacklist responses performed by this engine

NUMBYTESRECEIVED = 12201
Number of bytes received, used for VPN

NUMBYTESSENT = 12200
Number of bytes sent, used for VPN

NUMDISCARDRESPONSES = 368
Number of discard responses performed by this engine

NUMLOGEVENTS = 363
Number of log events

NUMLOGRESPONSES = 364
Number of log responses performed by this engine

NUMPACKETSRECEIVED = 549
Number of packets received

NUMPACKETSSENT = 548
Number of packets sent

NUMRECORDRESPONSES = 366
Number of record responses performed by this engine

NUMRESETRESPONSES = 367
Number of reset responses performed by this engine

OBJECTDN = 410
User and Group Information

OBJECTID = 406
Special field for filtering Audit entries using the defined resources. Not present in the audit entries as such.

OBJECTKEY = 409
Element Id

OBJECTNAME = 407
Elements being manipulated

OBJECTTYPE = 408
Element Type

ORIGINNAME = 400
Name of component producing event

OUTBOUNDSPI = 533
Outbound IPsec SPI value (hexadecimal)

PASSEDBYTES = 388
PASSED_BYTES

PEERCOMPONENTID = 307
Peer component id

PEERENDPOINT = 506
Peer VPN end point

PEERSECURITYGATEWAY = 505
Peer VPN gateway

PFSDHGROUP = 547
PFS Diffie-Hellman Group

PHASE1FAIL = 511
IKE_PHASE1_FAIL

PHASE1SUCC = 510
IKE_PHASE1_SUCC

PHASE2FAIL = 513
IKE_PHASE2_FAIL

PHASE2SUCC = 512
IKE_PHASE2_SUCC

POTENTIALLYDUPLICATERESPONSE = 170
Potentially duplicate correlation response

PROBEFAIL = 500
PROBE_FAIL

PROBEOK = 399
PROBE_OK

PROTOCOL = 11
IP protocol

QOSCLASS = 129
QoS Class

QOSPRIORITY = 131
QoS Priority

RADIUSACCOUNTINGTYPE = 851
Radius Accounting Type

RECEIVEDLOGEVENTS = 361
RECEIVED_LOG_EVENTS

RECEPTIONTIME = 24
Reception Time on the log Server

RESOURCE = 806
Resource

RESULT = 405
Result state

RETSRCIF = 49
Return source interface of the connection

ROUTEBGPPATH = 167
Active BGP path

ROUTEDISTANCE = 162
Relative distance for route validation

ROUTEGATEWAY = 164
IP address of the gateway for the route

ROUTEMETRIC = 163
Protocol specific metric value

ROUTENETMASK = 161
Netmask address of the network

ROUTENETWORK = 160
Network address of the network

ROUTEOSPFLSATYPE = 166
Type of OSPF LSA's

ROUTETYPE = 165
Type of route

RTT = 109
Round trip time of connection establishing

RULECOUNTERS = 412
RULE_COUNTERS

RULEHITS = 413
RULE_HITS

RULEID = 20
Rule tag value of acceptance rule

RWPHTTPREFERRER = 832
HTTP Referrer

RWPHTTPUSERAGENT = 830
HTTP User Agent

RWPSERVICENAME = 831
SSL VPN Portal Service Name

SAAUTHALG = 520
SA_AUTH_ALG

SABUNDLE = 514
SA_BUNDLE

SACIPHERALG = 518
SA_CIPHER_ALG

SACLASS = 535
SA Type

SACOMPRESSIONALG = 519
SA_COMPRESSION_ALG

SAEXPIREHARDLIMIT = 524
SA_EXPIRE_HARDLIMIT

SAEXPIRESOFTLIMIT = 523
SA_EXPIRE_SOFTLIMIT

SAINCOMING = 517
SA_INCOMING

SAKBHARDLIMIT = 522
SA_KB_HARDLIMIT

SAKBSOFTLIMIT = 521
SA_KB_SOFTLIMIT

SARESPONDER = 516
SA_RESPONDER

SATYPE = 515
SA_TYPE

SECURITYGATEWAY = 502
VPN gateway

SELECTEDCACHE = 396
SELECTED_CACHE

SELECTEDRTT = 395
SELECTED_RTT

SENDER = 5
None

SENDERDOMAIN = 38
Administrative Domain of Event Sender

SENDERTYPE = 31
Sender type

SENSORALLOWEDINSPECTEDTCPCONNECTIONS = 437

SENSORALLOWEDINSPECTEDUDPCONNECTIONS = 438

SENSORALLOWEDUNINSPECTEDTCPCONNECTIONS = 439

SENSORALLOWEDUNINSPECTEDUDPCONNECTIONS = 440

SENSORDISCARDEDTCPCONNECTIONS = 441

SENSORDISCARDEDUDPCONNECTIONS = 442

SENSORINSPECTEDBYTES = 357
Bytes inspected by sensor

SENSORINSPECTEDPACKETS = 358
Packets inspected by sensor

SENSORINTERFACEKEY = 370
Sensor interface key

SENSORLOSTBYTES = 359
Bytes lost in sensor

SENSORLOSTPACKETS = 360
Packets lost in sensor

SENSORPROCESSEDBYTES = 355
Bytes processed by sensor

SENSORPROCESSEDPACKETS = 356
Packets processed by sensor

SENSORRECEIVEDBYTES = 338
Bytes received by sensor

SENSORRECEIVEDPACKETS = 339
Packets received by sensor

SENSORTRAFFIC = 372
Sensor traffic

SENSORTRAFFICCLOSEDTCPCONNECTIONS = 383
Closed TCP Connections

SENSORTRAFFICINSPECTEDPACKETS = 376
Inspected Packets

SENSORTRAFFICLOSTPACKETS = 375
Lost Packets

SENSORTRAFFICNEWTCPCONNECTIONS = 381
New TCP Connections

SENSORTRAFFICNUMBEROFALERTS = 380
Number of Alerts

SENSORTRAFFICOKCONNECTIONS = 378
OK Connections

SENSORTRAFFICPROCESSEDBYTES = 374
Processed Bytes

SENSORTRAFFICPROCESSEDPACKETS = 373
Processed Packets

SENSORTRAFFICSTATSOFPACKETS = 377
Stats Of Packets

SENSORTRAFFICSUSPICIOUSCONNECTIONS = 379
Suspicious Connections

SENSORTRAFFICTCPHANDSHAKES = 382
TCP Handshakes

SENSORTRAFFICTCPTIMEOUTS = 384
TCP Timeouts

SENTLOGEVENTS = 362
SENT_LOG_EVENTS

SERVICE = 27
Special field for filtering logs using the defined services. Not present in the log entries as such.

SERVICEKEY = 132
Service primary key, used in service resolving

SESSIONDOMAIN = 414
Administrative Domain of Login Session

SESSIONEVENT = 302
Session monitoring event code (1 = new, 2 = update, 3 = remove, 4 = all sessions sent)

SESSIONID = 802
Id of the User Session

SFPINGRESS = 900
SFP_INGRESS

SHAPINGCLASS = 386
SHAPING_CLASS

SHAPINGGUARANTEE = 389
SHAPING_GUARANTEE

SHAPINGLIMIT = 390
SHAPING_LIMIT

SHAPINGPRIORITY = 391
SHAPING_PRIORITY

SITCATEGORY = 37
The type of the situation that caused sending this event.

SITUATION = 1000
The identifier of the situation that caused sending this event.

SNMPRETSRCIF = 51
SNMP index of return source interface

SNMPSRCIF = 50
SNMP index of source interface

SNMPTRAPMAP = 490
SNMP Trap

SNMPTRAPOID = 491
SNMP Trap OID

SNMPTRAPVALUE = 492
SNMP Trap Value

SPORT = 9
Connection source protocol port

SRC = 7
Connection source IP address

SRCADDRESS = 398
SRC_ADDRESS

SRCADDRS = 20007
Source addresses

SRCIF = 12
Source interface of firewall

SRCIPRANGE = 525
Source IP Range

SRCVLAN = 112
Source VLAN

SRCZONE = 46
Connection source interface zone

SRVHELPERID = 110
Protocol agent identification

SSLVPNSESSIONMONID = 811
Id of the User Session

SSLVPNSESSIONMONRECEIVED = 809
Node's local time when the SSL VPN session was created

SSLVPNSESSIONMONTIMEOUT = 810
Node's local time when the SSL VPN session will time-out

SSLVPNSESSIONTYPETYPE = 808
SSL VPN session client type

STATE = 116
Connection state in connection monitoring

STATUSTYPE = 311
Status type

STORAGESERVERID = 30
Storage Server

SYSLOGTYPE = 111
Syslog message type

TAGINFO = 480
Type tags

TCPDUMPSTATUS = 318
TCPDump Monitoring Status

TCPENCAPSULATION = 545
TCP Encapsulation

TIMEOUT = 115
Connection timeout in connection monitoring

TIMESTAMP = 1
Time of creating the event record.

TLSALERTDESCRIPTION = 45
TLS/SSL Alert Message Description

TLSALERTLEVEL = 44
TLS/SSL Alert Message Alert Level

TLSCERTIFICATEVERIFYERRORCODE = 39
TLS/SSL Certificate verify error code

TLSCIPHERSUITE = 42
TLS/SSL cipher suite

TLSCOMPRESSIONMETHOD = 43
TLS/SSL compression method

TLSDECRYPTED = 137
The connection was decrypted and re-encrypted in the engine to perform deep inspection or application identification.

TLSDETECTED = 136
The connection uses SSL/TLS protocol.

TLSDOMAIN = 40
Domain name field in SSL/TLS certificate

TLSMATCH = 135
TLS Match detected in the connection. Note that a single connection can have any number of distinct TLS Matches.

TLSPROTOCOLVERSION = 41
TLS/SSL protocol version

TOTALBYTES = 387
TOTAL_BYTES

TPACCEPTEDBYTES = 465
TP_ACCEPTED_BYTES

TPACCEPTEDPACKETS = 466
TP_ACCEPTED_PACKETS

TPDROPPEDBYTES = 467
TP_DROPPED_BYTES

TPDROPPEDPACKETS = 468
TP_DROPPED_PACKETS

TPMEMUSAGE = 470
Third party memory usage

TPNODELOAD = 469
Third party device load

TPRECEIVEDBYTES = 461
TP_RECEIVED_BYTES

TPRECEIVEDPACKETS = 462
TP_RECEIVED_PACKETS

TPSENTBYTES = 463
TP_SENT_BYTES

TPSENTPACKETS = 464
TP_SENT_PACKETS

TPTRAFFICCOUNTERS = 460
Third party traffic counters

TRAFFICCOUNTERS = 319
Traffic counters

TRAFFICSHAPING = 385
TRAFFIC_SHAPING

TRANSIENT = 26
None

TUNNELINGLEVEL = 95
Number of tunneling protocol layers encapsulating this protocol layer

TYPE = 23
Log event severity type

TYPEDESCRIPTION = 404
Description of the event

URLCATEGORYGROUP = 53
The type of the URL that caused sending this event.

URLCATEGORYRISK = 55
The risk of the URL that caused sending this event.

USERNAME = 3001
Username if present

USERORIGINATOR = 401
Administrator causing event

USERROLE = 402
Roles of Administrator causing event

VPNBYTESRECEIVED = 509
VPN_BYTES_RECEIVED

VPNBYTESSENT = 508
VPN_BYTES_SENT

VPNID = 501
Desination VPN

VPNSRCID = 499
Source VPN

VPNSTATISTICS = 507
VPN_STATISTICS

VPNSTATUS = 503
VPN_STATUS

VPNTYPE = 611
VPN_TYPE

VULNERABILITYREFERENCES = 20000
Generated from situation and original situation.

WIRELESSCHANNEL = 448
Wireless Access Point's channel

WIRELESSCONNECTIONS = 436
Number of wireless connections

WIRELESSMONITORING = 432
Wireless Monitoring

WIRELESSSECURITY = 435
Wireless Security mode

WIRELESSSSID = 433
Wireless SSID

WIRELESSSTATUS = 434
Wireless Status

ZIPEXPORTFILE = 420
Snapshot of element being manipulated

12.1.2.5 Formatters

Custom formats used to return data in different formats. These are used from the query itself when calling the fetch_as_format() method. For example, returning a LogQuery as a table:

```
query = LogQuery(fetch_size=200)
for log in query.fetch_batch(): # Default is TableFormat
    print(log)
```

As CSV:

```
query = LogQuery(fetch_size=200)
for log in query.fetch_batch(CSVFormat):
    print(log)
```

Each format also allows the ability to customize the fields that should be in the output. By default, each query type in *smc_monitoring.monitors* will have a class attribute `field_ids` which specify the default fields. These can be customized by modifying the query.format.field_ids([....]) parameter.

For example, modifying a routing query to return only destination interface and the route network:

```
query = RoutingQuery('sg_vm')
query.format.field_ids([LogField.DSTIF, LogField.ROUTENETWORK])
for log in query.fetch_batch():
    ...
```

The same field_id customization applies to all query types.

A simple way to view results is to use a RawDictFormat:

```
query = LogQuery(fetch_size=3)
query.format.field_names(['Src', 'Dst'])
for record in query.fetch_batch(RawDictFormat):
    ...
```

It is also possible to provide your own formatter. At a minimum you must provide a method called `formatted` in your class. The custom class should extend `_Header` to support custom field_ids within the query.

Note: Constants are defined in *smc_monitoring.models.constants*. Although there are many field values, not all field values will return results for every query. It is sometimes useful to log in to the SMC to verify available fields.

---

class smc_monitoring.models.formatters.**CSVFormat**(*query*)
Bases: smc_monitoring.models.formatters._Header

Return the results in CSV format. The first line will be a comma separated string with the field header. This is an iterable that will return results in batches of 200 (max) per iteration.

exception smc_monitoring.models.formatters.**InvalidFieldFormat**
Bases: exceptions.Exception

If using a complex format type such as combined, formatters are not supported. These specialized formats must be returned in raw dict format as they've been customized to return the data in a specific way.

class smc_monitoring.models.formatters.**RawDictFormat**(*query*)
Bases: object

Return the data as a list in raw dict format. The results are not filtered with exception of the returned fields based on field_id filters. This is a convenience format for consistency, although you can also call the *smc_monitoring.models.query.Query.fetch_raw* method to get the same data.

class smc_monitoring.models.formatters.**TableFormat**(*query*)
Bases: smc_monitoring.models.formatters._Header

Return the data in a table format. The field_id values will be used for the table header. Spacing will be calculated for each batch of results to align the table. The base spacing is determined by the header width, but adjusted wider if the data returned is wider. Anytime there is an adjustment to the width, a new table header will also be printed to visually realign. The query will return a max of 200 batch results per iteration.

Note: Table alignment will likely not be exact between batches as width is calculated per batch.

12.1.2.6 TimeRanges

Time formats are optionally used in a LogQuery to specify custom ranges for which to search 'stored' log events.

When adding a time format to a query, the `start_time` and `end_time` values need to be in milliseconds. The engine logs are stored in UTC time but in order to display the client side dates properly, you should set a timezone on the query.

There are helper methods to simplify retrieving for last_XXX period of time as well as custom range formats.

Set up a query with a time format:

```
query = LogQuery(fetch_size=50)
query.format.timezone('Europe/Helsinki')
query.time_range.last_five_minutes()
```

See also:

custom_range() for more examples on creating custom time range formats.

class smc_monitoring.models.calendar.**TimeFormat**(*start_ms=0*, *end_ms=0*)
Bases: object

Construct a time format to control the start and end times for a query. If unspecified, results will be limited by the fetch size quantity only. Helper methods are provided to simplify adding time based filters once the instance is constructed.

Parameters

- **start_ms** (*int*) – datetime object in milliseconds. Where to start the query in time. If your search should go backwards in time, specify the oldest time/date in start_time.
- **end_ms** (*int*) – datetime object in milliseconds. Where to end the query in time.

custom_range(*start_time*, *end_time=None*)
Provide a custom range for the search query. Start time and end time are expected to be naive `datetime` objects converted to milliseconds. When submitting the query, it is strongly recommended to set the timezone matching the local client making the query.

Example of finding all records on 9/2/2017 from 06:25:30 to 06:26:30 in the local time zone CST:

```
dt_start = datetime(2017, 9, 2, 6, 25, 30, 0)
dt_end = datetime(2017, 9, 2, 6, 26, 30, 0)

query = LogQuery()
query.format.timezone('CST')
query.time_range.custom_range(
    datetime_to_ms(dt_start),
    datetime_to_ms(dt_end))

for record in query.fetch_batch():
    print(record)
```

Last two minutes from current (py2):

```
now = datetime.now()
start_time = int((now - timedelta(minutes=2)).strftime('%s'))*1000
```

Specific start time (py2):

```
p2time = datetime.strptime("1.8.2017 08:26:42,76", "%d.%m.%Y %H:%M:%S,%f").
↪strftime('%s')
p2time = int(s)*1000
```

Specific start time (py3):

```
p3time = datetime.strptime("1.8.2017 08:40:42,76", "%d.%m.%Y %H:%M:%S,%f")
p3time.timestamp() * 1000
```

Parameters

- **start_time** (*int*) – search start time in milliseconds. Start time represents the oldest timestamp.
- **end_time** (*int*) – search end time in milliseconds. End time represents the newest timestamp.

end_time
Return the end time in datetime format. Will return 0 if end time is not specified.

Return type datetime

last_day()
Add time filter from current time back 1 day

last_fifteen_minutes()
Add time from current time back 15 minutes

last_five_minutes()
Add time from current time back 5 minutes

last_hour()
Add time from current time back 1 hour

last_thirty_minutes()
Add time from current time back 30 minutes

last_week()
Add time filter from current time back 7 days.

start_time
Return the start time in datetime format. Will return 0 if start time is not specified.

Return type datetime

smc_monitoring.models.calendar.**datetime_from_ms**(*ms*)
Convenience to return datetime from milliseconds

Returns datetime from ms

Return type datetime

smc_monitoring.models.calendar.**datetime_to_ms**(*dt*)
Convert an unaware datetime object to milliseconds. This datetime should be the time you would expect to see on the client side. The SMC will do the timestamp conversion based on the query timezone.

Returns value representing the datetime in milliseconds

Return type int

smc_monitoring.models.calendar.**subtract_from_now**(*td*)
Subtract timedelta from current time

12.1.3 Monitors

The monitors package provides modules that represent individual monitoring areas within the SMC monitoring API. Each monitor type extends *smc_monitoring.models.query.Query* to provide a consistent API for adding filters and executing queries.

12.1.3.1 Blacklist

Blacklist Query provides the ability to view current blacklist entries in the SMC by target. Target is defined as the cluster or engine. Retrieved results will have a reference to the entry and hence be possible to remove the entry.

```
query = BlacklistQuery('sg_vm')
query.format.timezone('CST')
```

Optionally add an "InFilter" to restrict search to a specific field:

```
query.add_in_filter(
    FieldValue(LogField.BLACKLISTENTRYSOURCEIP), [IPValue('2.2.2.2')])
```

An InFilter can also use a network based syntax:

```
query.add_in_filter(
    FieldValue(LogField.BLACKLISTENTRYSOURCEIP), [IPValue('2.2.2.0/24')])
```

Or combine filters using "AndFilter" or "OrFilter". Find an entry with source IP 2.2.2.2 OR 2.2.2.5:

```
ip1 = InFilter(FieldValue(LogField.BLACKLISTENTRYSOURCEIP), [IPValue('2.2.2.2')])
ip2 = InFilter(FieldValue(LogField.BLACKLISTENTRYSOURCEIP), [IPValue('2.2.2.5')])
query.add_or_filter([in_filter, or_filter])
```

Get the results of the query in the default TableFormat:

```
for entry in query.fetch_batch():
    print(entry)
```

Delete any blacklist entries with a source IP within a network range of 3.3.3.0/24:

```
query = BlacklistQuery('sg_vm')
query.add_in_filter(
    FieldValue(LogField.BLACKLISTENTRYSOURCEIP), [IPValue('3.3.3.0/24')])

for record in query.fetch_as_element():     # <-- must get as element to obtain␣
↪delete() method
    record.delete()
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.blacklist.**BlacklistEntry**(**kw*)
Bases: object

A blacklist entry represents an entry in the engines kernel table indicating that a source/destination/port/protocol mapping is currently being blocked by the engine. To remove a blacklist entry from an engine, retrieve all entries as element and remove the entry of interest by called `delete` on the element.

The simplest way to use search filters with a blacklist entry is to examine the BlacklistQuery `field_ids` and use these constant fields as InFilter definitions on the query.

blacklist_id
Blacklist entry ID. Useful if you want to locate the entry within the SMC UI.

Return type str

delete()
Delete the entry from the engine where the entry is applied.

Raises DeleteElementFailed

Returns None

dest_ports
Destination ports for this blacklist entry. If no ports are specified, 'ANY' is returned.

Return type str

destination
Destination network/netmask for this blacklist entry.

Return type str

duration
Duration for the blacklist entry.

Return type int

engine
The engine for this blacklist entry.

Return type str

href
The href for this blacklist entry. This is the reference to the entry for deleting the entry.

Return type str

protocol
Specified protocol for the blacklist entry. If none is specified, 'ANY' is returned.

Return type str

source
Source address/netmask for this blacklist entry.

Return type str

source_ports
Source ports for this blacklist entry. If no ports are specified (i.e. ALL ports), 'ANY' is returned.

Return type str

timestamp
Timestamp when this blacklist entry was added.

Return type str

class smc_monitoring.monitors.blacklist.**BlacklistQuery**(*target*, *timezone=None*, ***kw*)

Bases: *smc_monitoring.models.query.Query*

Query existing blacklist entries for a given cluster/engine. It is generally recommended to set your local timezone when making a query to convert the timestamp into a relevant format.

Parameters

- **target** (*str*) – NAME of the engine or cluster
- **timezone** (*str*) – timezone for timestamps.

Note: Timezone can be in the following formats: 'US/Eastern', 'PST', 'Europe/Helsinki'. More example time zone formats are available in the SMC Log Viewer -> Settings.

---

fetch_as_element (***kw*)

Fetch the blacklist and return as an instance of Element.

Returns generator returning element instances

Return type *BlacklistEntry*

12.1.3.2 Connections

A connection query returns all currently connected sessions on the given target.

Create a query to obtain all connections for a given engine:

```
query = ConnectionQuery('sg_vm')
```

Add a timezone to the query:

```
query.format.timezone('CST')
```

Add a filter to only get connections if the source address is 172.18.1.252:

```
query.add_in_filter(FieldValue(LogField.SRC), [IPValue('172.18.1.252')])
```

Only connections that match a specific service:

```
query.add_in_filter(FieldValue(LogField.SERVICE), [ServiceValue('TCP/443', 'UDP/53')])
```

Execute query and return raw results:

```
for records in query.fetch_raw():
    ...
```

Execute query and return as an *Connection* element:

```
for records in query.fetch_as_element():
    ...
```

Retrieving live streaming results:

```
for records in query.fetch_live():
    ...
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.connections.**Connection**(***data*)

Bases: object

Connection represents a state table entry. This is the result of making a *ConnectionQuery* and using *fetch_as_element()*.

dest_addr

Destination address for this entry

Return type str

dest_port

Destination port for the entry.

Return type int

engine

The engine/cluster for this state table entry

Returns engine or cluster for this entry

Return type str

protocol

Protocol for this entry

Returns protocol (UDP/TCP/ICMP, etc)

Return type str

service

Service for this entry

Returns service (HTTP/HTTPS, etc)

Return type str

source_addr

Source address for this entry

Return type str

source_port

Source port for the entry.

Return type int

state

State of the connection.

Returns state, i.e. UDP established, TCP established, etc.

Return type str

timestamp

Timestamp of this connection. It is recommended to set the timezone on the query to view this timestamp in the systems local time. For example:

```
query.format.timezone('CST')
```

Returns timestamp in string format

Return type str

class smc_monitoring.monitors.connections.**ConnectionQuery**(*target*, ***kw*)
Bases: *smc_monitoring.models.query.Query*

Show all current connections on the specified target.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters **target** (*str*) – name of target engine/cluster

fetch_as_element(***kw*)
Fetch the results and return as a Connection element. The original query is not modified.

Returns generator of elements

Return type *Connection*

12.1.3.3 Logs

LogQuery provides an interface to the SMC Log Viewer to retrieve data in real time or by batch.

There are a variety of settings you can configure on a query such as whether to execute a real time query versus a stored log fetch, time frame for the query, fetch size quantity, returned format style, specify which fields to return and adding filters to make a very specific query.

To make queries, first obtain a query object and optionally (recommended) specify a maximum number of records to fetch (for non-real time fetches). The default log query type is 'stored', and if a `fetch_size` is not provided, one batch of 200 records will be returned:

```
query = LogQuery(fetch_size=50)
```

If real time logs are preferred and set `fetch_type='current'` (default is fetch 'stored' logs):

```
query = LogQuery(fetch_type='current')
```

You can also use the shortcut `fetch_live` on the query:

```
query = LogQuery()
for result in query.fetch_live():
    ...
```

Note: If selecting `fetch_size='current'` log queries will be real-time and ignore the `fetch_size`, `time_range`, and `backwards` values if provided on the query.

You can also set a time_range on the query. There are convenience methods on a TimeFormat object to simplify adding a time range. When using time ranges, you should set the timezone on the query to the clients timezone:

```
query = LogQuery(fetch_size=50)
query.time_range.last_five_minutes()
query.format.timezone('CST')
```

You can also use custom time ranges to search between a specific period of time. This is done by providing a *smc_monitoring.models.calendar.TimeFormat* instance to the Query constructor, or by modifying the query `time_range` attribute. The TimeFormat object takes a 'naive' datetime object for start and end times. The start and end times must also be in milliseconds.

Example of finding all records on 9/2/2017 from 06:25:30 to 06:26:30 in the local time zone CST:

```
dt_start = datetime(2017, 9, 2, 6, 25, 30, 0)
dt_end = datetime(2017, 9, 2, 6, 26, 30, 0)

query = LogQuery()
query.format.timezone('CST')    # <--- Set the timezone on the query!
query.time_range.custom_range(
    datetime_to_ms(dt_start),
    datetime_to_ms(dt_end))
```

See also:

smc_monitoring.models.calendar.TimeFormat for more examples and information on using a TimeFormat in a query.

Adding filters to a query can be achieved by using add_XX_filter convenience methods or by calling `update_filter` with the filter object.

For example, customizing the fields returned using `query.format.field_ids`, and filtering for only HIGH alerts with a source address of 192.168.4.84:

```
query = LogQuery(fetch_size=10)
query.format.timezone('CST')

query.format.field_ids([LogField.TIMESTAMP, LogField.ACTION, LogField.SRC, LogField.
↪DST])

query.add_and_filter(
    [InFilter(FieldValue(LogField.ALERTSEVERITY), [ConstantValue(Alerts.HIGH)]),
     InFilter(FieldValue(LogField.SRC), [IPValue('192.168.4.84')])])
```

See also:

`smc.monitoring.filters` for information on how to use and combine filters for a query.

class smc_monitoring.monitors.logs.**LogQuery**(*fetch_type='stored'*, *fetch_size=None*, *backwards=True*, *format=None*, *time_range=None*, ***kw*)

Bases: *smc_monitoring.models.query.Query*

Make a Log Query to the SMC to fetch stored log data or monitor logs in real time.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters

- **fetch_type** (*str*) – 'stored' or 'current'
- **fetch_size** (*int*) – max number of logs to fetch
- **backwards** (*bool*) – by default records are returned from newest to oldest (backwards=True). To return in opposite direction, set backwards=False. Default: True
- **format** (format type from *smc_monitoring.models.formats* (default: TextFormat)) – A format object specifying format of return data
- **time_range** (TimeFormat) – time filter to add to query

fetch_batch(*formatter=<class 'smc_monitoring.models.formatters.TableFormat'>*)

Fetch a batch of logs and return using the specified formatter. Formatter is class type defined in *smc_monitoring.models.formatters*. This fetch type will be a single shot fetch (this method

forces `fetch_type='stored'`). If `fetch_size` is not already set on the query, the default fetch_size will be 200.

Parameters **formatter** – Formatter type for data representation. Any type in *smc_monitoring.models.formatters*.

Returns generator returning data in specified format

fetch_live(*formatter=<class 'smc_monitoring.models.formatters.TableFormat'>*)
View logs in real-time. If previous filters were already set on this query, they will be preserved on the original instance (this method forces `fetch_type='current'`).

Parameters **formatter** – Formatter type for data representation. Any type in *smc_monitoring.models.formatters*.

Returns generator of formatted results

fetch_raw()
Execute the query and return by batches. Optional keyword arguments are passed to Query.execute(). Whether this is real-time or stored logs is dependent on the value of `fetch_type`.

Returns generator of dict results

fetch_size
Return the fetch size for this query. If fetch size is set to 0, the query will be aborted after the first response message. If the fetch_size is None, it is considered undefined which indicates there is no fetch bound set on this query (i.e. fetch all).

..note:: It is recommended to provide a fetch_size to limit the results when doing a 'stored' query.

Returns configured fetch size for this query

Return type int

12.1.3.4 Routes

Query the current routing table entries.

Create a query to obtain all connections for a given engine:

```
query = RoutingQuery('sg_vm')
```

Add a timezone to the query:

```
query.format.timezone('CST')
```

Add a filter to only routes for destination network 192.168.4.0/24:

```
query.add_in_filter(FieldValue(LogField.ROUTENETWORK), [IPValue('192.168.4.0')])
```

Only routes that use a specific gateway:

```
query.add_in_filter(FieldValue(LogField.ROUTEGATEWAY), [IPValue('172.18.1.200')])
```

Execute query and return raw results:

```
for records in query.fetch_batch():
    ...
```

Execute query and return as an *RoutingView* element:

```
for records in query.fetch_as_element():
    ...
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.routes.**RoutingQuery**(*target*, ***kw*)
Bases: *smc_monitoring.models.query.Query*

Show all current dynamic and static routes on the specified target.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters **target** (*str*) – name of target engine/cluster

fetch_as_element(***kw*)
Fetch the results and return as a RoutingView element. The original query is not modified.

Returns generator of elements

Return type *RoutingView*

class smc_monitoring.monitors.routes.**RoutingView**(***data*)
Bases: object

A Routing View represents an entry in the current routing table. This is the result of making a *RoutingQuery* and using *fetch_as_element()*.

dest_if
Destination interface for this route

Return type str

dest_vlan
Destination VLAN for this route, if any.

Return type str

dest_zone
Destination zone for this route, if any.

Return type str

engine
The engine/cluster for this route

Return type str

route_gw
The route gateway for this route.

Return type str

route_metric
Metric for this route.

Returns route metric

Return type int

route_network
The route network for this route.

Return type str

route_type
The type of route.

Returns Static, Connection, Dynamic, etc.

Return type str

timestamp
Timestamp of this connection. It is recommended to set the timezone on the query to view this timestamp in the systems local time. For example:

```
query.format.timezone('CST')
```

:return timestamp in string format :rtype: str

12.1.3.5 SSLVPN

SSLVPN currently connected users.

Create a query to obtain all connections for a given engine:

```
query = SSLVPNQuery('sg_vm')
```

Add a timezone to the query:

```
query.format.timezone('CST')
```

Execute query and return raw results:

```
for records in query.fetch_batch():
    ...
```

Execute query and return as an *SSLVPNUser* element:

```
for records in query.fetch_as_element():
    ...
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.sslvpn.**SSLVPNQuery**(*target*, ***kw*)
Bases: *smc_monitoring.models.query.Query*

Show all current SSL VPN connections on the specified target.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters **target** (*str*) – name of target engine/cluster

fetch_as_element(***kw*)
Fetch the results and return as an SSLVPNUser element. The original query is not modified.

Returns generator of elements

Return type *SSLVPNUser*

class smc_monitoring.monitors.sslvpn.**SSLVPNUser**(***data*)
Bases: object

Connection represents a state table entry. This is the result of making a *SSLVPNQuery* and using *fetch_as_element()*.

engine
The engine/cluster for this state table entry

Returns engine or cluster for this entry

Return type str

session_expiration
Time the session expires. It is recommended that you add a timezone to the query to present this in human readable format:

```
query.format.timezone('CST')
```

Return type str

session_start
Time the session started. It is recommended that you add a timezone to the query to present this in human readable format:

```
query.format.timezone('CST')
```

Return type str

source_addr
Source IP address for the SSL VPN user

Return type str

username
Username for this SSL VPN user

Return type str

12.1.3.6 Users

Get active users on target cluster/engine.

Create a query to obtain all users for a given engine:

```
query = UserQuery('sg_vm')
```

Add a timezone to the query:

```
query.format.timezone('CST')
```

Execute query and return raw results:

```
for records in query.fetch_batch():
    ...
```

Execute query and return as a *User* element:

```
for records in query.fetch_as_element():
    ...
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.users.**User**(**data*)
Bases: object

User mapping currently in user cache on specified target. This is the result of making a *UserQuery* and using *fetch_as_element()*.

domain
SMC Domain that this user record belongs to

Returns name of SMC domain, 'Shared' is default

Return type str

engine
The engine/cluster for this state table entry

Returns engine or cluster for this entry

Return type str

expiration
Expiration time for this user entry. It is recommended to add a timezone to the query to display this field in the client local time.

Returns expiration time for this user authentication entry

Return type str

ipaddress
IP address for the entry

Return type str

timestamp
Timestamp of this connection. It is recommended to set the timezone on the query to view this timestamp in the systems local time. For example:

```
query.format.timezone('CST')
```

:return timestamp in string format :rtype: str

username
Username for entry

Returns username value as fully qualified domain name

Return type str

class smc_monitoring.monitors.users.**UserQuery**(*target*, **kw*)
Bases: *smc_monitoring.models.query.Query*

Show all authenticated users on the specified target.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters **target** (*str*) – name of target engine/cluster

fetch_as_element(**kw*)
Fetch the results and return as a User element. The original query is not modified.

Returns generator of elements

Return type *User*

12.1.3.7 VPNs

Get all active VPN SA's.

Create a query to obtain all connections for a given engine:

```
query = VPNSAQuery('sg_vm')
```

Add a timezone to the query:

```
query.format.timezone('CST')
```

Execute query and return raw results:

```
for records in query.fetch_batch():
    ...
```

Execute query and return as a *VPNSecurityAssoc* element:

```
for records in query.fetch_as_element():
    ...
```

Delete a VPN SA:

```
query = VPNSAQuery('sg_vm')
for sa in query.fetch_as_element():
    sa.delete()
```

See also:

smc_monitoring.models.filters for more information on creating filters

class smc_monitoring.monitors.vpns.**VPNSAQuery**(*target*, ***kw*)
Bases: *smc_monitoring.models.query.Query*

Show all current VPN SA's on the specified target.

Variables ***field_ids*** (*list*) – field IDs are the default fields for this entry type and are constants found in *smc_monitoring.models.constants.LogField*

Parameters **target** (*str*) – name of target engine/cluster

fetch_as_element(***kw*)
Fetch the results and return as a VPNSecurityAssoc element. The original query is not modified.

Returns generator of elements

Return type *VPNSecurityAssoc*

class smc_monitoring.monitors.vpns.**VPNSecurityAssoc**(***data*)
Bases: object

A VPN Security Association represents a currently connected VPN endpoint. This is the result of making a *VPNSAQuery* and using *fetch_as_element()*.

bytes_received
Number of bytes received.

Return type int

bytes_sent
Number of bytes sent.

Return type int

engine
The engine/cluster for this VPN

Return type str

expiration
Expiration time for this tunnel Security Association

Return type str

local_endpoint
Local endpoint (IP address) for this VPN tunnel.

Return type str

local_gateway
Local gateway for this VPN.

Return type str

local_networks
Local protected networks

Return type str

negotiation_role
Role for this tunnel entry.

Returns Negotiation role, i.e. Initiator, Responder, etc.

Return type str

peer_endpoint
Peer endpoint element and IP Address for this tunnel.

Return type str

peer_gateway
Peer gateway for this VPN.

Return type str

peer_networks
Remote protected networks

Return type str

protocol
WHich protocol is associated with this tunnel entry.

Returns IP protocol for tunnel, i.e. ESP/UDP

Return type str

sa_type
SA Type for this VPN tunnel. Each VPN tunnel will typically have at least two entries, one for IPSEC and another for IKE.

Return type str

timestamp
Timestamp of this connection. It is recommended to set the timezone on the query to view this timestamp in the systems local time. For example:

```
query.format.timezone('CST')
```

Return type str

12.1.3.8 Alerts

ActiveAlert Query provides the ability to view current alert entries from the alert log viewer. When creating the query, you must specify a target which speifies the SMC domain for which to retrieve the alerts.

A basic alert query using a local timezone example:

```
query = ActiveAlertQuery('Shared Domain')
query.format.timezone('CST')
```

You can also use standard filters to specify a more exact match, for example, showing alerts with a severity of CRITICAL:

```
query.add_in_filter(
    FieldValue(LogField.ALERTSEVERITY), [ConstantValue(Alerts.CRITICAL)])
```

class smc_monitoring.monitors.alerts.**ActiveAlertQuery**(*target='Shared Domain'*, *timezone=None*)
Bases: *smc_monitoring.models.query.Query*

Active Alert Query is an interface to the alert log viewer in SMC. This query type provides the ability to fetch and filter on active alerts.

You can create a new query specifying a valid timezone abbreviation:

```
query = ActiveAlertQuery('Shared Domain', timezone='CST')
```

Or alternatively no timezone:

```
query = ActiveAlertQuery('DomainFoo')
```

Parameters

- **target** (*str*) – domain for which to filter alerts. Default: 'Shared Domain'
- **timezone** (*str*) – timezone for timestamps, i.e. 'CST', etc

fetch_as_element(**kw*)
Fetch the results and return as a User element. The original query is not modified.

Returns generator returning element instances

Return type *Alert*

class smc_monitoring.monitors.alerts.**Alert**(**data*)
Bases: object

Alert definition returned from specified domain. This is the result of making a *ActiveAlertQuery* and using *fetch_as_element()*.

action
Action performed for the alert

Return type str

destination
Destination IP for the alert

Return type str

destination_port
Destination port for alert

Return type int

engine
The engine/cluster for this state table entry

Returns engine or cluster for this entry

Return type str

protocol
Protocol for alert

Return type str

service
Service associated with alert

Return type str

severity
Severity for this alert

Return type str

situation
Situation defined for this alert

Return type str

source
Source IP for the alert

Return type str

source_port
Source port for alert

Return type int

timestamp
Timestamp of this connection. It is recommended to set the timezone on the query to view this timestamp in the systems local time. For example:

```
query.format.timezone('CST')
```

:return timestamp in string format :rtype: str

vulnerability_refs
Comma seperated string listing any vulnerability references for the alert, if any.

Return type str

CHAPTER 13

API Reference

13.1 Session

Session module for tracking existing connection state to SMC

class smc.api.session.**Session**(*manager=None*)

Session represents the clients session to the SMC. A session is obtained by calling login(). If sessions need to be long lived as might be the case when running under a web platform, a session is automatically refreshed when it expires. Best practice is to call logout() after to clear the session from the SMC. A session will be automatically closed once the python interpreter closes.

Each session will also have a single connection pool associated with it. This results in a single persistent connection to the SMC that will be re-used as needed.

api_version

Current API Version

Return type str

domain

Logged in SMC domain

Return type str

entry_points

Entry points that are bound to this session. Entry points are exposed by the SMC API and provide links to top level resources

Return type Resource

is_active

Is this session active. Active means there is a stored session ID for the SMC using the current account. This does not specify whether the session ID has been timed out on the server but does indicate the account has not called logout.

Return type bool

is_ssl
Is this an SSL connection

Return type bool

login(*url=None*, *api_key=None*, *login=None*, *pwd=None*, *api_version=None*, *timeout=None*, *verify=True*, *alt_filepath=None*, *domain=None*, ***kwargs*)
Login to SMC API and retrieve a valid session. Sessions use a pool connection manager to provide dynamic scalability during times of increased load. Each session is managed by a global session manager making it possible to have more than one session per interpreter.

An example login and logout session:

```
from smc import session
session.login(url='http://1.1.1.1:8082', api_key='SomeSMCG3ener@t3dPwd')
.....do stuff.....
session.logout()
```

Parameters

- **url** (*str*) – ip of SMC management server
- **api_key** (*str*) – API key created for api client in SMC
- **login** (*str*) – Administrator user in SMC that has privilege to SMC API.
- **pwd** (*str*) – Password for user login.
- **(optional)** (*api_version*) – specify api version
- **timeout** (*int*) – (optional): specify a timeout for initial connect; (default 10)
- **verify** (*str|boolean*) – verify SSL connections using cert (default: verify=True) You can pass verify the path to a CA_BUNDLE file or directory with certificates of trusted CAs
- **alt_filepath** (*str*) – If using .smcrc, alternate path+filename
- **domain** (*str*) – domain to log in to. If domains are not configured, this field will be ignored and api client logged in to 'Shared Domain'.
- **retry_on_busy** (*bool*) – pass as kwarg with boolean if you want to add retries if the SMC returns HTTP 503 error during operation. You can also optionally customize this behavior and call *set_retry_on_busy()*

Raises ***ConfigLoadError*** – loading cfg from ~.smcrc fails

For SSL connections, you can disable validation of the SMC SSL certificate by setting verify=False, however this is not a recommended practice.

If you want to use the SSL certificate generated and used by the SMC API server for validation, set verify='path_to_my_dot_pem'. It is also recommended that your certificate has subjectAltName defined per RFC 2818

If SSL warnings are thrown in debug output, see: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings

Logout should be called to remove the session immediately from the SMC server.

Note: As of SMC 6.4 it is possible to give a standard Administrative user access to the SMC API. It is still possible to use an API Client by providing the api_key in the login call.

---

logout()
Logout session from SMC

Returns None

manager
Return the session manager for this session

Return type SessionManager

name
Return the administrator name for this session. Can be None if the session has not yet been established.

Note: The administrator name was introduced in SMC version 6.4. Previous versions will show the unique session identifier for this session.

Return type str

refresh()
Refresh session on 401. This is called automatically if your existing session times out and resends the operation/s which returned the error.

Raises ***SMCConnectionError*** – Problem re-authenticating using existing api credentials

session_id
The session ID in header type format. Can be inserted into a connection if necessary using:

```
{'Cookie': session.session_id}
```

Return type str

set_retry_on_busy(*total=5*, *backoff_factor=0.1*, *status_forcelist=None*, ***kwargs*)
Mount a custom retry object on the current session that allows service level retries when the SMC might reply with a Service Unavailable (503) message. This can be possible in larger environments with higher database activity. You can all this on the existing session, or provide as a dict to the login constructor.

Parameters

- **total** (*int*) – total retries
- **backoff_factor** (*float*) – when to retry
- **status_forcelist** (*list*) – list of HTTP error codes to retry on
- **method_whitelist** (*list*) – list of methods to apply retries for, GET, POST and PUT by default

Returns None

switch_domain(*domain*)
Switch from one domain to another. You can call session.login() with a domain key value to log directly into the domain of choice or alternatively switch from domain to domain. The user must have permissions to the domain or unauthorized will be returned. In addition, when switching domains, you will be logged out of the current domain to close the connection pool associated with the previous session. This prevents potentially excessive open connections to SMC

```
session.login() # Log in to 'Shared Domain'
...
session.switch_domain('MyDomain')
```

Raises ***SMCConnectionError*** – Error logging in to specified domain. This typically means the domain either doesn't exist or the user does not have privileges to that domain.

timeout
Session timeout in seconds

Return type int

url
The fully qualified SMC URL in use, includes the port number

Return type str

13.2 Element

class smc.base.model.**ElementBase**(**meta*)
Element base provides a meta data container and an instance cache as well as methods to retrieve aspects of an element. Meta is passed in to Element and SubElement types to provide links to resources. When a top level query is made to the SMC API, meta is returned for the element (unless a direct link query is made). The meta format include 'href','type','name'. For example:

```
"href":"http://1.1.1.1:8082/6.4/elements/host/707","name":"foobar","type":"host"
```

Methods of the element classes are designed to expose any links or attributes of the specific element to simplify manipulation. If a method, etc is accessed that requires the elements data, the element is fetched and the elements cache (stored in *data* attribute) is inflated. The ETag is also retained in the element and is used when updating or deleting the element to ensure we are operating on the latest version.

Meta can be passed to constructor through as key value pairs kwargs, href=.... (only partial meta), or meta={.....} (as dict)

If meta is not provided, the meta attribute will be None

delete()
Delete the element

Raises ***DeleteElementFailed*** – possible dependencies, record locked, etc

Returns None

update(*exception*, ***kwargs*)
Update the existing element and clear the instance cache. Removing the cache will ensure subsequent calls requiring element attributes will force a new fetch to obtain the latest copy.

Calling update() with no args will assume the element has already been modified directly and the data cache will be used to update. You can also override the following attributes: href, etag and json. If json is sent, it is expected to the be a complete payload to satisfy the update.

For kwargs, if attribute values are a list, you can pass 'append_lists=True' to add to an existing list, otherwise overwrite (default: overwrite)

See also:

To see different ways to utilize this method for updating, see: *Update*.

Parameters

- **exception** – pass a custom exception to throw if failure
- **kwargs** – optional kwargs to update request data to server.

Raises

- ***ModificationFailed*** – raised if element is tagged as System element
- ***UpdateElementFailed*** – failed to update element with reason

Returns href of the element modified

Return type str

class smc.base.model.**Element**(*name*, ***meta*)

Bases: *smc.base.model.ElementBase*

Base element with common methods shared by inheriting classes. If stashing attributes on this class, be sure to prefix with an underscore to avoid having the attributes serialized when calling update.

objects(self): Interface to element collections. All classes inheriting from *Element* can access collections through this class property:

```
for host in Host.objects.all():
        ...
```

Fetch a single entry:

```
host = Host.objects.filter('myhost')
...
```

For more information on collections, see: *smc.base.collection.CollectionManager*

add_category(*category*)

Category Tags are used to characterize an element by a type identifier. They can then be searched and returned as a group of elements. If the category tag specified does not exist, it will be created. This change will take effect immediately.

Parameters **tags** (*list(str)*) – list of category tag names to add to this element

Raises ***ElementNotFound*** – Category tag element name not found

Returns None

See also:

smc.elements.other.Category

categories

Search categories assigned to this element

```
>>> from smc.elements.network import Host
>>> Host('kali').categories
[Category(name=foo), Category(name=foocategory)]
```

Return type list(*Category*)

comment

Comment for element

duplicate(*name*)

New in version 0.5.8: Requires SMC version >= 6.3.2

Duplicate this element. This is a shortcut method that will make a direct copy of the element under the new name and type.

Parameters **name** (*str*) – name for the duplicated element

Raises *ActionCommandFailed* – failed to duplicate the element

Returns the newly created element

Return type *Element*

export (*filename='element.zip'*)
Export this element.

Usage:

```
engine = Engine('myfirewall')
extask = engine.export(filename='fooexport.zip')
while not extask.done():
    extask.wait(3)
print("Finished download task: %s" % extask.message())
print("File downloaded to: %s" % extask.filename)
```

Parameters **filename** (*str*) – filename to store exported element

Raises *TaskRunFailed* – invalid permissions, invalid directory, or this element is a system element and cannot be exported.

Returns DownloadTask

Note: It is not possible to export system elements

classmethod get (*name*, *raise_exc=True*)
Get the element by name. Does an exact match by element type.

Parameters

- **name** (*str*) – name of element
- **raise_exc** (*bool*) – optionally disable exception.

Raises *ElementNotFound* – if element does not exist

Return type *Element*

classmethod get_or_create (*filter_key=None*, *with_status=False*, ***kwargs*)
Convenience method to retrieve an Element or create if it does not exist. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to *get()*. Any keyword arguments passed except the optional filter_key will be used in a create() call. If filter_key is provided, this should define an attribute and value to use for an exact match on the element. Valid attributes are ones required on the elements `create` method or can be viewed by the elements class docs. If no filter_key is provided, the name field will be used to find the element.

```
>>> Network.get_or_create(
        filter_key={'ipv4_network': '123.123.123.0/24'},
        name='mynetwork',
        ipv4_network='123.123.123.0/24')
Network(name=mynetwork)
```

The kwargs should be used to satisfy the elements `create` classmethod parameters to create in the event it cannot be found.

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a tuple is returned with (Element, created), where the second tuple item indicates if the element has been created or not.

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

history

New in version 0.5.7: Requires SMC version >= 6.3.2

Obtain the history of this element. This will not chronicle every modification made over time, but instead a current snapshot with historical information such as when the element was created, by whom, when it was last modified and it's current state.

Raises ***ResourceNotFound*** – If not running SMC version >= 6.3.2

Return type *History*

name

Name of element

referenced_by

Show all references for this element. A reference means that this element is being used, for example, in a policy rule, as a member of a group, etc.

Returns list referenced elements

Return type list(*Element*)

rename (*name*)

Rename this element.

Parameters **name** (*str*) – new name of element

Raises ***UpdateElementFailed*** – update failed with reason

Returns None

classmethod update_or_create (*filter_key=None*, *with_status=False*, ***kwargs*)

Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to *get()*. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
```

(continues on next page)

(continued from previous page)

```
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

class smc.base.model.**SubElement**(***meta*)

Bases: *smc.base.model.ElementBase*

SubElement is the base class for elements that do not have direct entry points in the SMC and instead are obtained through a reference. They are not 'loaded' directly as are classes that inherit from *Element*.

class smc.base.model.**UserElement**(*name*, ***meta*)

Bases: *smc.base.model.ElementBase*

User element mixin for LDAP of Internal Domains. Provides comparison and encoding/decoding of the DN used in URIs.

unique_id

Fully qualified unique DN for this entry

Return type str

class smc.core.resource.**History**

History description of this element. This will provide basic information about the element such as when it was created, last modified along with the accounts making the modifications.

Variables

- **is_locked** (*bool*) – is this record currently locked
- **is_osbsolete** (*bool*) – is this record obsoleted
- **is_trashed** (*bool*) – is the record in the trash bin

created_by

The account that created this element. Returned as an Element.

Return type *Element*

last_modified

When the element was last modified as a datetime object

Return type datetime

modified_by

The account that last modified this element.

Return type *Element*

when_created
When the element was created as a datetime object

Return type datetime

13.3 Administration

13.3.1 Access Rights

Access Rights provide the ability to create administrative accounts and assign or create specific access control lists and roles to these accounts.

13.3.1.1 AccessControlList

class smc.administration.access_rights.**AccessControlList**(*name*, ***meta*)
Bases: *smc.base.model.Element*

An ACL is assigned to an AdminUser to grant limited access permissions to either Engines, Policies or Domains. The access control list will have 'granted elements' that represent the elements that apply to this permission. The SMC provides default ACL's that can be used or new ones can be created. Find all available ACL's:

```
>>> AccessControlList.objects.all()
```

add_permission(*elements*)
Add permission/s to this ACL. By default this change is committed after the method is called.

Parameters **elements** (*list(str,Element)*) – Elements to grant access to. Can be engines, policies, or other ACLs

Raises ***UpdateElementFailed*** – Failed updating permissions

Returns None

classmethod create(*name*, *granted_element=None*)
Create a new ACL

Parameters

- **name** (*str*) – Name of ACL
- **granted_elements** (*list(str,Element)*) – Elements to grant access to. Can be engines, policies or other acl's.

Raises ***CreateElementFailed*** – failed creating ACL

Returns instance with meta

Return type *AccessControlList*

permissions
Elements associated to this permission. Granted elements can be Engines, Policies or other Access Control Lists.

Returns Element class deriving from *smc.base.model.Element*

remove_permission(*elements*)
Remove permission/s to this ACL. Change is committed at end of method call.

Parameters **elements** (*list(str,Element)*) – list of element/s to remove

Raises *UpdateElementFailed* – Failed modifying permissions

Returns None

13.3.1.2 Administrators

User module to hold accounts related to users (admin or local) in the SMC

You can create an Admin User, enable superuser, enable/disable the account, assign local access to engines, and change the account password for SMC or engine access.

It is possible to fully provision an Admin User with specific permissions and roles and initial password.

Create the admin:

```
admin = AdminUser.create(name='auditor', superuser=False)
```

Note: If the Admin User should have unrestricted access, set `superuser=True` and skip the below sections related to adding permissions and roles.

Permissions relate to elements that the user will have access to (Policies, Engines or AccessControlLists) and the domain where the privileges apply (default is 'Shared Domain').

Create a permission using the default domain of Shared, granting access to a specific engine and firewall policy:

```
permission = Permission.create(
    elements=[Engine('vm'), FirewallPolicy('VM Policy')],
    role=Role('Viewer'))
```

Create a second permission granting access to all firewalls in the domain 'mydomain':

```
domain_perm = Permission.create(
    elements=[AccessControlList('ALL Firewalls')],
    role=Role('Owner'),
    domain=AdminDomain('mydomain'))
```

Add the permissions to the Admin User:

```
admin.add_permission([permission, domain_perm])
```

Set an initial password for the Admin User:

```
admin.change_password('Newpassword1')
```

Note: Roles are used to define what granular controls will be available to the assigned user, such as read/read write/all. AccessControlLists encapsulate elements into a single container for re-use.

See also:

smc.administration.role.Role and *smc.administration.access_rights.AccessControlList* for more information.

class smc.elements.user.**AdminUser**(*name*, ***meta*)

Bases: *smc.elements.user.UserMixin*, *smc.base.model.Element*

Represents an Adminitrator account on the SMC Use the constructor to create the user.

Create an Admin:

```
>>> AdminUser.create(name='dlepage', superuser=True)
AdminUser(name=dlepage)
```

If modifications are required after you can access the admin and make changes:

```
admin = AdminUser('dlepage')
admin.change_password('mynewpassword1')
admin.enable_disable()
```

Attributes available:

Variables

- **allow_sudo** (*bool*) – is this account allowed to sudo on an engine.
- **local_admin** (*bool*) – is the admin a local admin
- **superuser** (*bool*) – is this account a superuser for SMC

change_engine_password(*password*)

Change Engine password for engines on allowed list.

Parameters **password** (*str*) – password for engine level

Raises ***ModificationFailed*** – failed setting password on engine

Returns None

classmethod create(*name*, *local_admin=False*, *allow_sudo=False*, *superuser=False*, *enabled=True*, *engine_target=None*, *can_use_api=True*, *console_superuser=False*, *allowed_to_login_in_shared=True*, *comment=None*)

Create an admin user account.

New in version 0.6.2: Added can_use_api, console_superuser, and allowed_to_login_in_shared. Requires SMC >= SMC 6.4

Parameters

- **name** (*str*) – name of account
- **local_admin** (*bool*) – is a local admin only
- **allow_sudo** (*bool*) – allow sudo on engines
- **can_use_api** (*bool*) – can log in to SMC API
- **console_superuser** (*bool*) – can this user sudo via SSH/console
- **allowed_to_login_in_shared** (*bool*) – can this user log in to the shared domain
- **superuser** (*bool*) – is a super administrator
- **enabled** (*bool*) – is account enabled
- **engine_target** (*list*) – engine to allow remote access to

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *AdminUser*

enabled
Read only enabled status

Return type bool

class smc.elements.user.**ApiClient**(*name*, ***meta*)
Bases: *smc.elements.user.UserMixin*, *smc.base.model.Element*

Represents an API Client

classmethod create(*name*, *enabled=True*, *superuser=True*)
Create a new API Client. Once client is created, you can create a new password by:

```
>>> client = ApiClient.create('myclient')
>>> print(client)
ApiClient(name=myclient)
>>> client.change_password('mynewpassword')
```

Parameters

- **name** (*str*) – name of client
- **enabled** (*bool*) – enable client
- **superuser** (*bool*) – is superuser account

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *ApiClient*

class smc.elements.user.**UserMixin**
Bases: object

User Mixin class providing common operations for Admin Users and API Clients.

add_permission(*permission*)
Add a permission to this Admin User. A role defines permissions that can be enabled or disabled. Elements define the target for permission operations and can be either Access Control Lists, Engines or Policy elements. Domain specifies where the access is granted. The Shared Domain is default unless specific domain provided. Change is committed at end of method call.

Parameters **permission** (*list(Permission)*) – permission/s to add to admin user

Raises ***UpdateElementFailed*** – failed updating admin user

Returns None

change_password(*password*)
Change user password. Change is committed immediately.

Parameters **password** (*str*) – new password

Returns None

enable_disable()
Toggle enable and disable of administrator account. Change is committed immediately.

Raises ***UpdateElementFailed*** – failed with reason

Returns None

generate_password()
Generate a random password for this user.

Returns random password

Return type str

permissions
Return each permission role mapping for this Admin User. A permission role will have 3 fields:

- Domain
- Role (Viewer, Operator, etc)
- Elements (Engines, Policies, or ACLs)

Returns permissions as list

Return type list(*Permission*)

13.3.1.3 Permission

class smc.administration.access_rights.**Permission**(*granted_elements=None*, *role_ref=None*, *granted_domain_ref=None*)
Permissions are added to admin users that do not have super user access rights. An Admin User can also have multiple permissions. There are three primary fields associated with a permission:

- Domain to grant access
- Elements to grant access to (Engines, Policies or AccessControlLists)
- Role

A permission might be used to grant read-only access to specific policies or firewalls (read-only vs read write). It can also be specific to the Admin Domain.

See also:

smc.elements.user

classmethod create(*elements*, *role*, *domain=None*)
Create a permission.

Parameters

- **granted_elements** (*list*(*str*,Element)) – Elements for this permission. Can be engines, policies or ACLs
- **role** (*str*,Role) – role for this permission
- **domain** (*str*,Element) – domain to apply (default: Shared Domain)

Return type *Permission*

domain
Domain this permission applies to. Shared Domain if unspecified.

Return type *AdminDomain*

granted_elements
List of elements this permission has rights to. Elements will be of type Engine, Policy or ACLs

Return type list(*Element*)

role
Specific Role assigned to this permission. A role is what allows read/write access to specific operations on the granted elements

Return type *Role*

13.3.1.4 Roles

Administrator Role elements specify a restricted set of permissions that include the right to create, edit, and delete elements.

Each administrator can have several different Administrator Roles applied to different sets of elements. There are some default Administrator Roles, but if you want to customize the permissions in any way, you must create custom Administrator Role elements.

Create a new role is done by using the create classmethod. By default the role will not have any permissions set:

```
>>> from smc.administration.role import Role
>>> role = Role.create(name='mynewrole')
```

A role has many attributes (mostly boolean) that can be enabled, therefore the simplest way to create a new role is to duplicate an existing role.

```
>>> list(Role.objects.all())
[Role(name=myeditor), Role(name=Logs Viewer), Role(name=Reports Manager),␣
↪Role(name=Owner),
 Role(name=Viewer), Role(name=Operator), Role(name=Monitor), Role(name=Editor),
 Role(name=Superuser)]
...
```

Duplicate an existing role to simplify making modifications on permissions:

```
>>> role = Role('Editor')
>>> role.duplicate('customeditor')
Role(name=customeditor)
```

To enable or disable role permissions, use the enable/disable option after retrieving the Role resource.

Available and current permission settings can be found by calling permissions attribute:

```
>>> role = Role('newrole')
>>> role.permissions
[{'alert_mgmt': False}, {'send_advanced_commands': False}, {'license_mgmt': False}, {
↪'element_edit': False},
 {'view_edit_report': False}, {'view_system_alerts': False}, {'view_logs': False}, {
↪'vpn_mgmt': False},
 {'log_pruning_mgmt': False}, {'updates_and_upgrades_mgmt': False}, {'auth_server_
↪user_mgmt': False},
 {'view_audit': False}, {'element_delete': False}, {'element_create': False}, {
↪'upload_policy': False},
 {'send_commands': False}, {'backup_mgmt': False}, {'element_view_content': True}, {
↪'log_mgmt': False},
 {'bookmark_manage': True}, {'admin_mgmt': False}, {'name': 'newrole'}, {'overview_
↪manage': True},
 {'internal_user_mgmt': False}, {'refresh_policy': False}]
```

Then enable specific roles by specifying the keys to enable:

```
>>> role.enable(['element_create', 'upload_policy'])
```

Also disable specific roles:

```
>>> role.disable(['element_create', 'upload_policy'])
```

Once modification is complete, call update on the role:

```
>>> role.update()
'http://172.18.1.151:8082/6.4/elements/role/10'
```

class smc.administration.role.**Role**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Role class represents granular access control rights that can be applied to specific elements (Engines, Policies or Access Control Lists).

classmethod create(*name*, *comment=None*)

Create a new role. The role will not have any permissions by default so it will be required to call enable on the role after creation.

Parameters

- **name** (*str*) – name of role
- **comment** (*str*) – comment for role

Raises ***CreateElementFailed*** – failed to create role

Return type *Role*

disable(*values*)

Disable specific permissions on this role. Use *permissions* to view valid permission settings and current value/s. Change is committed immediately.

Parameters **values** (*list*) – list of values by allowed types

Returns None

enable(*values*)

Enable specific permissions on this role. Use *permissions* to view valid permission settings and current value/s. Change is committed immediately.

Parameters **values** (*list*) – list of values by allowed types

Returns None

permissions

Return valid permissions and setting for this role. Permissions are returned as a list of dict items, {permission: state}. State for the permission is either True or False. Use *enable()* and *disable()* to toggle role settings.

Returns list of permission settings

Return type list(dict)

13.3.2 Certificates

13.3.2.1 TLSCommon

TLS Common module provides mixin methods that are common to certificate handling in SMC. Importing certificates and private keys can be done by providing a file where the certificates/keys are stored, or providing in string format.

class smc.administration.certificates.tls_common.**ImportExportCertificate**

Mixin to provide certificate import and export methods to relevant classes.

export_certificate (*filename=None*)
Export the certificate. Returned certificate will be in string format. If filename is provided, the certificate will also be saved to the file specified.

Raises ***CertificateExportError*** – error exporting certificate

Return type str or None

import_certificate (*certificate*)
Import a valid certificate. Certificate can be either a file path or a string of the certificate. If string certificate, it must include the —–BEGIN CERTIFICATE—– string.

Parameters **certificate_file** (*str*) – fully qualified path to certificate file

Raises

- ***CertificateImportError*** – failure to import cert with reason
- **IOError** – file not found, permissions, etc.

Returns None

class smc.administration.certificates.tls_common.**ImportExportIntermediate**
Mixin to provide import and export capabilities for intermediate certificates

export_intermediate_certificate (*filename=None*)
Export the intermediate certificate. Returned certificate will be in string format. If filename is provided, the certificate will also be saved to the file specified.

Raises ***CertificateExportError*** – error exporting certificate, can occur if no intermediate certificate is available.

Return type str or None

import_intermediate_certificate (*certificate*)
Import a valid certificate. Certificate can be either a file path or a string of the certificate. If string certificate, it must include the —–BEGIN CERTIFICATE—– string.

Parameters **certificate** (*str*) – fully qualified path or string

Raises

- ***CertificateImportError*** – failure to import cert with reason
- **IOError** – file not found, permissions, etc.

Returns None

class smc.administration.certificates.tls_common.**ImportPrivateKey**
Mixin to provide import capabilities to relevant classes that require private keys.

import_private_key (*private_key*)
Import a private key. The private key can be a path to a file or the key in string format. If in string format, the key must start with —–BEGIN. Key types supported are PRIVATE RSA KEY and PRIVATE KEY.

Parameters **private_key** (*str*) – fully qualified path to private key file

Raises

- ***CertificateImportError*** – failure to import cert with reason
- **IOError** – file not found, permissions, etc.

Returns None

13.3.2.2 TLSServerCredential

TLS module provides interactions related to importing TLS Server Credentials for inbound SSL decryption, as well as client protection certificates used for outbound decryption.

To properly decrypt inbound TLS connections, you must provide the Stonesoft FW with a valid certificate and private key. Within SMC these certificate types are known as TLS Server Credentials.

Once you have imported these certificates, you must then assign them to the relevant engines that will perform the decryption services. Lastly you will need a rule that enables HTTPS with decryption.

First start by importing the TLS Server Credential class:

```
>>> from smc.administration.certificates.tls import TLSServerCredential
```

If you want to create a TLS Server Credential in steps, the process is as follows:

```
tls = TLSServerCredential.create(name)     # Create the certificate element
tls.import_certificate(certificate) # Import the certificate
tls.import_private_key(private_key) # Import the private key
tls.import_intermediate_certificate(intermediate) # Import intermediate certificate␣
↪(optional)
```

Otherwise, use helper methods that allow you to do this in a single step.

For example, creating the TLS credential from certificate files:

```
>>> tls = TLSServerCredential.import_signed(
              name='server.test.local',
              certificate='/pathto/server.crt',
              private_key='/pathto/server.key',
              intermediate=None)  # <-- You can also include intermediate certificates
>>> tls
TLSServerCredential(name=server.test.local)
```

Note: Certificate, private key and intermediate certificates can also be specified in raw string format and must start with the BEGIN CERTIFICATE, etc common syntax.

You can also import certificates from a certificate chain file. When doing so, the certificates are expected to be in the order: server certificate, intermediate/s, root certificate. You can optionally also add the private key to the chain file or provide it separately:

```
tls = TLSServerCredential.import_from_chain(
    name='fromchain', certificate_file='/path/cert.chain',
    private_key='/path/priv.key')
```

Note: If multiple intermediate certificates are added, only the first one is imported into the TLS Server Credential. In addition, the root certificate is ignored and should be imported using `TLSCertificateAuthority.create()`.

It is also possible to create self signed certificates using the SMC CA:

```
>>> tls = TLSServerCredential.create_self_signed(
    name='server.test.local', common_name='CN=server.test.local')
>>> tls
TLSServerCredential(name=server.test.local)
```

If you would rather use the SMC to generate the CSR and have the request signed by an external CA you can call *TLSServerCredential.create_csr()* and export the request:

```
>>> tls = TLSServerCredential.create_csr(name='public.test.local', common_name=
↪'CN=public.test.local')
>>> tls.certificate_export()
'-----BEGIN CERTIFICATE REQUEST-----
MIIEXTCCAkcCAQAwHDEaMBgGA1UEAwwRcHVibGljLnRlc3QubG9jYWwwggIiMA0G
CSqGSIb3DQEBAQUAA4ICDwAwggIKAoICAQC68xcXrWQ5E25nkTfmgmPQiWVPwf
....
....
-----END CERTIFICATE REQUEST-----'
```

Optionally export the request to a local file:

```
>>> tls = TLSServerCredential.create_csr(
    name='public2.test.local', common_name='CN=public2.test.local')
>>> tls.certificate_export(filename='public2.test.local.csr')
```

If you use an external CA for signing your certficiates, you can also import that as a TLS Certificate Authority. The link between the certificates and root CA will be made automatically:

```
TLSCertificateAuthority.create(
    name='myrootca',
    certificate='/path/to/cert/or/string')
```

Once you have the TLS Server Credentials within SMC, you can then assign them to the relevant engines:

```
>>> from smc.core.engine import Engine
>>> from smc.administration.certificates import TLSServerCredential
>>> engine = Engine('myfirewall')
>>> engine.tls_inspection.add_tls_credential([TLSServerCredential('public.test.local
↪'), TLSServerCredential('server.test.local')])
>>> engine.tls_inspection.server_credentials
[TLSServerCredential(name=public.test.local), TLSServerCredential(name=server.test.
↪local)]
```

Note: It is possible to import and export certificates from the SMC, but it is not possible to export private keys.

class smc.administration.certificates.tls.**TLSServerCredential**(*name*, ***meta*)

Bases: *smc.administration.certificates.tls_common.ImportExportIntermediate*, *smc.administration.certificates.tls_common.ImportPrivateKey*, *smc.administration.certificates.tls_common.ImportExportCertificate*, *smc.base.model.Element*

If you want to inspect TLS traffic for which an internal server is the destination, you must create a TLS Credentials element to store the private key and certificate of the server.

The private key and certificate allow the firewall to decrypt TLS traffic for which the internal server is the destination so that it can be inspected.

After a TLSServerCredential has been created, you must apply this to the engine performing decryption and create the requisite policy rule that uses SSL decryption.

Variables **certificate_state** (*str*) – State of the certificate. Available states are 'request' and 'certificate'. If the state is 'request', this represents a CSR and needs to be signed.

classmethod create(*name*)
Create an empty certificate. This will only create the element in the SMC and will then require that you import the server certificate, intermediate (optional) and private key.

See also:

import_signed() and *import_from_chain()*.

Raises ***CreateElementFailed*** – failed creating element

Return type *TLSServerCredential*

classmethod create_csr(*name*, *common_name*, *public_key_algorithm='rsa'*, *signature_algorithm='rsa_sha_512'*, *key_length=4096*)
Create a certificate signing request.

Parameters

- **name** (*str*) – name of TLS Server Credential
- **rcommon_name** (*str*) – common name for certificate. An example would be: "CN=CommonName,O=Organization,OU=Unit,C=FR,ST=PACA,L=Nice". At minimum, a "CN" is required.
- **public_key_algorithm** (*str*) – public key type to use. Valid values rsa, dsa, ecdsa.
- **signature_algorithm** (*str*) – signature algorithm. Valid values dsa_sha_1, dsa_sha_224, dsa_sha_256, rsa_md5, rsa_sha_1, rsa_sha_256, rsa_sha_384, rsa_sha_512, ecdsa_sha_1, ecdsa_sha_256, ecdsa_sha_384, ecdsa_sha_512. (Default: rsa_sha_512)
- **key_length** (*int*) – length of key. Key length depends on the key type. For example, RSA keys can be 1024, 2048, 3072, 4096. See SMC documentation for more details.

Raises ***CreateElementFailed*** – failed to create CSR

Return type *TLSServerCredential*

classmethod create_self_signed(*name*, *common_name*, *public_key_algorithm='rsa'*, *signature_algorithm='rsa_sha_512'*, *key_length=4096*)
Create a self signed certificate. This is a convenience method that first calls *create_csr()*, then calls *self_sign()* on the returned TLSServerCredential object.

Parameters

- **name** (*str*) – name of TLS Server Credential
- **rcommon_name** (*str*) – common name for certificate. An example would be: "CN=CommonName,O=Organization,OU=Unit,C=FR,ST=PACA,L=Nice". At minimum, a "CN" is required.
- **public_key_algorithm** (*str*) – public key type to use. Valid values rsa, dsa, ecdsa.
- **signature_algorithm** (*str*) – signature algorithm. Valid values dsa_sha_1, dsa_sha_224, dsa_sha_256, rsa_md5, rsa_sha_1, rsa_sha_256, rsa_sha_384, rsa_sha_512, ecdsa_sha_1, ecdsa_sha_256, ecdsa_sha_384, ecdsa_sha_512. (Default: rsa_sha_512)
- **key_length** (*int*) – length of key. Key length depends on the key type. For example, RSA keys can be 1024, 2048, 3072, 4096. See SMC documentation for more details.

Raises

- ***CreateElementFailed*** – failed to create CSR
- ***ActionCommandFailed*** – Failure to self sign the certificate

Return type *TLSServerCredential*

classmethod import_from_chain(*name*, *certificate_file*, *private_key=None*)

Import the server certificate, intermediate and optionally private key from a certificate chain file. The expected format of the chain file follows RFC 4346. In short, the server certificate should come first, followed by any intermediate certificates, optionally followed by the root trusted authority. The private key can be anywhere in this order. See https://tools.ietf.org/html/rfc4346#section-7.4.2.

Note: There is no validation done on the certificates, therefore the order is assumed to be true. In addition, the root certificate will not be imported and should be separately imported as a trusted root CA using `create`

If the certificate chain file has only two entries, it is assumed to be the server certificate and root certificate (no intermediates). In which case only the certificate is imported. If the chain file has 3 or more entries (all certificates), it will import the first as the server certificate, 2nd as the intermediate and ignore the root cert.

You can optionally provide a seperate location for a private key file if this is not within the chain file contents.

Warning: A private key is required to create a valid TLS Server Credential.

Parameters

- **name** (*str*) – name of TLS Server Credential
- **certificate_file** (*str*) – fully qualified path to chain file or file object
- **private_key** (*str*) – fully qualified path to chain file or file object

Raises

- **IOError** – error occurred reading or finding specified file
- **ValueError** – Format issues with chain file or empty

Return type *TLSServerCredential*

classmethod import_signed(*name*, *certificate*, *private_key*, *intermediate=None*)

Import a signed certificate and private key to SMC, and optionally an intermediate certificate. The certificate and the associated private key must be compatible with OpenSSL and be in PEM format. The certificate and private key can be imported as a raw string, file path or file object. If importing as a string, be sure the string has carriage returns after each line and the final *END CERTIFICATE* line.

Import a certificate and private key:

```
>>> tls = TLSServerCredential.import_signed(
        name='server2.test.local',
        certificate='mydir/server.crt',
        private_key='mydir/server.key')
>>> tls
TLSServerCredential(name=server2.test.local)
```

Parameters

- **name** (*str*) – name of TLSServerCredential
- **certificate** (*str*) – fully qualified to the certificate file, string or file object

- **private_key** (*str*) – fully qualified to the private key file, string or file object
- **intermediate** (*str*) – fully qualified to the intermediate file, string or file object

Raises

- ***CertificateImportError*** – failure during import
- ***CreateElementFailed*** – failed to create credential
- **IOError** – failure to find certificate files specified

Return type *TLSServerCredential*

self_sign()
Self sign the certificate in 'request' state.

Raises ***ActionCommandFailed*** – failed to sign with reason

valid_from
New in version 0.6.0: Requires SMC version >= 6.3.4

The valid from datetime for this TLS Server Credential.

Return type datetime.datetime

valid_to
New in version 0.6.0: Requires SMC version >= 6.3.4

The expiration (valid to) datetime for this TLS Server Credential.

Return type datetime.datetime

13.3.2.3 TLSProfile

class smc.administration.certificates.tls.**TLSProfile**(*name*, ***meta*)
Bases: *smc.base.model.Element*

New in version 0.6.2: Requires SMC >= 6.4

Represents a TLS Profile. Contains common parameters for establishing TLS based connections. TLS Profiles are used in various configuration areas such as SSL VPN portal and Active Directory (when using TLS) connections.

classmethod create(*name*, *tls_version*, *use_only_subject_alt_name=False*, *accept_wildcard=False*, *check_revocation=True*, *tls_cryptography_suites=None*, *crl_delay=0*, *ocsp_delay=0*, *ignore_network_issues=False*, *tls_trusted_ca_ref=None*, *comment=None*)
Create a TLS Profile. By default the SMC will have a default NIST TLS Profile but it is also possible to create a custom profile to provide special TLS handling.

Parameters

- **name** (*str*) – name of TLS Profile
- **tls_verison** (*str*) – supported tls verison, valid options are TLSv1.1, TLSv1.2, TLSv1.3
- **use_only_subject_alt_name** (*bool*) – Use Only Subject Alt Name when the TLS identity is a DNS name
- **accept_wildcard** (*bool*) – Does server identity check accept wildcards
- **check_revocation** (*bool*) – Is certificate revocation checked

- **tls_cryptography_suites** (*str*, TLSCryptographySuite) – allowed cryptography suites for this profile. Uses NIST profile if not specified
- **crl_delay** (*int*) – Delay time (hours) for fetching CRL
- **ocsp_delay** (*int*) – Ignore OCSP failure for (hours)
- **ignore_network_issues** (*bool*) – Ignore revocation check failures due to network issues
- **tls_trusted_ca_ref** (*list*) – Trusted Certificate Authorities, empty list means trust any
- **comment** (*str*) – optional comment

Raises

- ***CreateElementFailed*** – failed to create element with reason
- ***ElementNotFound*** – specified element reference was not found

Return type *TLSProfile*

13.3.2.4 TLSIdentity

class smc.administration.certificates.tls.**TLSIdentity**(*tls_field*, *tls_value*)

Bases: smc.base.structs.NestedDict

New in version 0.6.2: Requires SMC >= 6.4

A TLS Identity represents a field and value pair that will be used to validate a TLS certificate. This can be used in various areas where TLS is used such as VPN.

Valid tls field types are:

> DNSName IPAddress CommonName DistinguishedName SHA-1 SHA-256 SHA-512 MD5 Email user_principal_name

13.3.2.5 TLSCryptographySuite

class smc.administration.certificates.tls.**TLSCryptographySuite**(*name*, ***meta*)

Bases: *smc.base.model.Element*

This represents a TLS Cryptography Suite Set used in various configurations that require a TLS Profile such as SSL VPN Tunneling, Reverse Web Proxy, ActiveDirectory TLS, etc.

static ciphers(*from_suite=None*)

This is a helper method that will return all of the cipher strings used in a specified TLSCryptographySuite or returns the system default NIST profile list of ciphers. This can be used as a helper to identify the ciphers to specify/add when creating a new TLSCryptographySuite.

Return type dict

classmethod create(*name*, *comment=None*, ***ciphers*)

Create a new TLSCryptographySuite. The ciphers kwargs should be a dict with the cipher suite string as key and boolean value to indicate if this cipher should be enabled. To obtain the valid cipher suite string name, use the following method:

```
cipher_strings = TLSCryptographySuite.ciphers()
```

Then to create a custom cipher suite, provide the ciphers as a dict of kwargs. In this example, create a TLS Crypto Suite that only enables AES 256 bit ciphers:

```
only256 = dict(((cipher, True) for cipher in TLSCryptographySuite.ciphers()
    if 'aes_256' in cipher))

mytls = TLSCryptographySuite.create(name='mytls', **only256)
```

Parameters

- **name** (*str*) – name of this TLS Crypto suite
- **ciphers** (*dict*) – dict of ciphers with cipher string as key and bool as value, True enables the cipher

Raises ***CreateElementFailed*** – failed to create element with reason

Return type *TLSCryptographySuite*

13.3.2.6 ClientProtectionCA

class smc.administration.certificates.tls.**ClientProtectionCA**(*name*, ***meta*)

Bases: *smc.administration.certificates.tls_common.ImportPrivateKey*, *smc.administration.certificates.tls_common.ImportExportCertificate*, *smc.base.model.Element*

Client Protection Certificate Authority elements are used to inspect TLS traffic between an internal client and an external server for outbound decryption.

When an internal client makes a connection to an external server that uses TLS, the engine generates a substitute certificate that allows it to establish a secure connection with the internal client. The Client Protection Certificate Authority element contains the credentials the engine uses to sign the substitute certificate it generates.

Variables

- **certificate** (*str*) – base64 encoded certificate for this CA
- **crl_checking_enabled** (*bool*) – whether CRL checking is turned on
- **internal_ca** (*bool*) – is this an internal CA (default: false)
- **oscp_checking_enabled** (*bool*) – is OSCP validation enabled

---

Note: If the engine does not use a signing certificate that is already trusted by users web browsers when it signs the substitute certificates it generates, users receive warnings about invalid certificates. To avoid these warnings, you must either import a signing certificate that is already trusted, or configure users web browsers to trust the engine signing certificate.

---

classmethod create(*name*)

Create a client protection CA. Once the client protection CA is created, to activate you must then call import_certificate and import_private_key. Or optionally use the convenience classmethod *import_signed()*.

Raises ***CreateElementFailed*** – failed to create base Client CA

Return type *ClientProtectionCA*

classmethod create_self_signed(*name*, *prefix*, *password*, *public_key_algorithm='rsa'*, *life_time=365*, *key_length=2048*)

Create a self signed client protection CA. To prevent browser warnings during decryption, you must trust the signing certificate in the client browsers.

Parameters

- **name** (*str*) – Name of this ex: "SG Root CA" Used as Key. Real common name will be derivated at creation time with a uniqueId.
- **prefix** (*str*) – prefix used for derivating file names
- **password** (*str*) – password for private key
- **public_key_algorithm** – public key algorithm, either rsa, dsa or ecdsa
- **life_time** (*str*, *int*) – lifetime in days for CA
- **key_length** (*int*) – length in bits, either 1024 or 2048

Raises

- ***CreateElementFailed*** – creating element failed
- ***ActionCommandFailed*** – failed to self sign the certificate

Return type *ClientProtectionCA*

classmethod import_signed(*name*, *certificate*, *private_key*)

Import a signed certificate and private key as a client protection CA.

This is a shortcut method to the 3 step process:

- Create CA with name
- Import certificate
- Import private key

Create the CA:

```
ClientProtectionCA.import_signed(
    name='myclientca',
    certificate_file='/pathto/server.crt'
    private_key_file='/pathto/server.key')
```

Parameters

- **name** (*str*) – name of client protection CA
- **certificate_file** (*str*) – fully qualified path or string of certificate
- **private_key_file** (*str*) – fully qualified path or string of private key

Raises

- ***CertificateImportError*** – failure during import
- **IOError** – failure to find certificate files specified

Return type *ClientProtectionCA*

13.3.3 Domains

class smc.administration.system.**AdminDomain**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Administrative domain element. Domains are used to provide object based segmentation within SMC. If domains are in use, you can log in directly to a domain to modify contents within that domain.

Find all available domains:

```
>>> list(AdminDomain.objects.all())
[AdminDomain(name=Shared Domain)]
```

Note: Admin Domains require and SMC license.

---

classmethod create(*name*, *comment=None*)
Create a new Admin Domain element for SMC objects.

Example:

```
>>> AdminDomain.create(name='mydomain', comment='mycomment')
>>> AdminDomain(name=mydomain)
```

Parameters

- **name** (*str*) – name of domain
- **comment** (*str*) – optional comment

Raises *CreateElementFailed* – failed creating element with reason

Returns instance with meta

Return type *AdminDomain*

13.3.4 License

Module representing read-only licenses in SMC

class smc.administration.license.**License**(***data*)
Valid attributes (read-only) are:

Variables

- **binding** – master license binding serial number
- **binding_state** – state of license, unassigned, bound, etc
- **bindings** – which node is the license bound to
- **customer_name** – customer name, if any
- **enabled_feature_packs** – additional feature licenses
- **expiration_date** – when license expires
- **features** – features enabled on this license
- **granted_date** – when license date began
- **license_id** – license ID (unique for each license)

- **license_version** – max version for this license
- **maintenance_contract_expires_date** – date/time support ends
- **management_server_binding** – management server binding POS
- **proof_of_license** – proof of license key
- ***type*** – type of license (SECNODE, Mgmt, etc)

class smc.administration.license.**Licenses**(*licenses*)
List of all available licenses for this Management Server.

13.3.5 Scheduled Tasks

New in version 0.5.7: Requires SMC version >= 6.3.2

Scheduled tasks are administrative processes that can run either immediately after being defined, or scheduled to run on a regular basis. Scheduled tasks in the SMC are defined under Administration->Tasks->Definition.

Some tasks are read-only, meaning they are system elements and cannot be modified or copied and can therefore only be scheduled (these task related classes will not have a *create* method). Other tasks can be created and custom settings can be defined. Check the documentation for each task to determine the capabilities.

All tasks inherit the ScheduledTaskMixin which provides a *start* method and access to a TaskSchedule instance through the *task_schedule* property. The associated TaskSchedule defines whether to run the task ongoing and details specifying when the task should be run and how often.

An example follows that shows how to use a refresh policy task. Other tasks use the same API syntax.

Finding existing tasks for a specific task type:

```
for task in RefreshPolicyTask.objects.all():
    print(task, task.task_schedule)
```

Review an existing task and it's task schedule:

```
task = RefreshPolicyTask(name='mytask')
for schedule in task.task_schedule:
    print(schedule.activation_date, schedule.activated)
```

Create a refresh policy refresh task:

```
task = RefreshPolicyTask.create(
    name='mytask',
    engines=[Engine('engine1'), Engine('engine2')],
    comment='some comment')
```

A created task can always be run at any time without having to set a schedule for the task by calling *start* on the task:

```
task = RefreshPolicyTask('mytask')
task.start()
```

A task can also be scheduled for a future time. Adding a scheduled run to the task requires that we first obtain the task and add the schedule to it. This can be done when creating the task, or the retrieved after:

```
task = RefreshPolicyTask.create(
    name='mytask',
    engines=[Engine('engine1'), Engine('engine2')],
```

(continues on next page)

(continued from previous page)

```
    comment='refresh policy on specified engines')

task.add_schedule(
    name='refresh_policy_on_saturday',
    activation_date=1512325716000,  # 12/04/2017 00:00:00
    day_period='weekly',
    day_mask=128,
    comment='tun this task weekly')
```

You can also specify tasks that run on a regular interval, such as monthly:

```
task = RefreshPolicyTask(name='mytask')
task.add_schedule(
    name='run_monthly',
    activation_date=1512367200000, # Start 12/4/2017 at 00:00:00
    day_period='monthly')
```

Repeat a task for a period of time, then disable task on specified date:

```
task = DeleteLogTask.create(
    name='Delete SMC Server logs',
    servers='all',
    time_range='last_full_month',
    all_logs=True)

task.add_schedule(
    name='Run for 6 months',
    activation_date=1512367200000, # Start 12/04/2017
    day_period='monthly',
    repeat_until_date=1528088400000, # End 06/04/2018
    comment='purge log task')
```

Note: You can use the helper method `smc.base.util.datetime_to_ms()` for obtaining millisecond times for scheduled tasks.

---

class smc.administration.scheduled_tasks.**DeleteLogTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A delete log task defines a way to purge log data from the SMC. When defining the task, you specify which servers to delete from (typically management AND log server/s), and which log types to delete.

Note: Log tasks currently support pre-defined time ranges such as 'yesterday', 'last_week', etc. If creating custom time ranges for tasks, use the SMC UI.

---

classmethod create(*name*, *servers=None*, *time_range='yesterday'*, *all_logs=False*, *filter_for_delete=None*, *comment=None*, ***kwargs*)

Create a new delete log task. Provide True to all_logs to delete all log types. Otherwise provide kwargs to specify each log by type of interest.

Parameters

- **name** (*str*) – name for this task

- **servers** (*list(ManagementServer or LogServer)*) – servers to back up. Servers must be instances of management servers or log servers. If no value is provided, all servers are backed up.
- **time_range** (*str*) – specify a time range for the deletion. Valid options are 'yesterday', 'last_full_week_sun_sat', 'last_full_week_mon_sun', 'last_full_month' (default 'yesterday')
- **filter_for_delete** (FilterExpression) – optional filter for deleting. (default: FilterExpression('Match All')
- **all_logs** (*bool*) – if True, all log types will be deleted. If this is True, kwargs are ignored (default: False)
- **kwargs** – see *log_target_types()* for keyword arguments and default values.

Raises

- ***ElementNotFound*** – specified servers were not found
- ***CreateElementFailed*** – failure to create the task

Returns the task

Return type *DeleteLogTask*

class smc.administration.scheduled_tasks.**DeleteOldRunTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only task to delete the task history from already run tasks. This is generally a recommended task to run on a monthly basis to purge the old task data.

class smc.administration.scheduled_tasks.**DeleteOldSnapshotsTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only management server task to delete snapshots since the last scheduled run. For example, if this task is configured to run once per month, snapshots older than 1 month will be deleted.

class smc.administration.scheduled_tasks.**DisableUnusedAdminTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only task to disable any administrator account that has not been used within the time set in the Administrator password policy.

class smc.administration.scheduled_tasks.**FetchCertificateRevocationTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only management server task to download updated certificate revocation lists.

class smc.administration.scheduled_tasks.**RefreshMasterEnginePolicyTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

Refresh a Master Engine and virtual policy task.

Note: This task is only relevant for engines that are Master Engines. This does not apply to standard single FW or clustered FW's.

classmethod create(*name*, *master_engines*, *comment=None*)
Create a refresh task for master engines.

Parameters

- **name** (*str*) – name of task
- **master_engines** (*list (MasterEngine)*) – list of master engines for this task
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed to create the task

Returns the task

Return type *RefreshMasterEnginePolicyTask*

class smc.administration.scheduled_tasks.**RefreshPolicyTask**(*name*, ***meta*)
Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A scheduled task associated with refreshing policy on engine/s. A refresh will push an existing policy that is already mapped to the engine/s. Use *UploadPolicyTask* to create a task that will assign a policy to an engine/s and upload.

Note: Any engine can force a policy refresh on the engine node directly by calling engine.refresh(), or from the engines assigned policy by calling policy.refresh(engine) also.

classmethod create(*name*, *engines*, *comment=None*, *validate_policy=True*, ***kwargs*)
Create a refresh policy task associated with specific engines. A policy refresh task does not require a policy be specified. The policy used in the refresh will be the policy already assigned to the engine.

Parameters

- **name** (*str*) – name of this task
- **engines** (*list (Engine)*) – list of Engines for the task
- **comment** (*str*) – optional comment
- **validate_policy** (*bool*) – validate the policy before upload. If set to true, validation kwargs can also be provided if customization is required, otherwise default validation settings are used.
- **kwargs** – see *policy_validation_settings()* for keyword arguments and default values.

Raises

- ***ElementNotFound*** – engine specified does not exist
- ***CreateElementFailed*** – failure to create the task

Returns the task

Return type *RefreshPolicyTask*

class smc.administration.scheduled_tasks.**RenewGatewayCertificatesTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only management server task that renews certificates on internal gateways which have automatic certificate renewal enabled.

class smc.administration.scheduled_tasks.**RenewInternalCATask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only management server task that renews certificate authorities used in system communications and send alerts about expiring certificate authorities.

class smc.administration.scheduled_tasks.**RenewInternalCertificatesTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only management server task that renews certificates used in systems communications and send alerts about expiring certificates.

class smc.administration.scheduled_tasks.**SGInfoTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

An SGInfo task is used for obtaining support data from the engine/s.

Note: An sginfo can be executed directly on an engine node by calling the node.sginfo() method directly.

Variables

- **include_core_files** (*bool*) – whether to include core files in output
- **include_slapcat_output** (*bool*) – include slapcat in output

Warning: For an sginfo to be readable, the engine must not have the 'encrypt_configuration' field enabled on the engine or the data will be unreadable.

classmethod create(*name*, *engines*, *include_core_files=False*, *include_slapcat_output=False*, *comment=None*)

Create an sginfo task.

Parameters

- **name** (*str*) – name of task
- **engines** (*list(Engine)*) – list of engines to apply the sginfo task
- **include_core_files** (*bool*) – include core files in the sginfo backup (default: False)
- **include_slapcat_output** (*bool*) – include output from a slapcat command in output (default: False)

Raises

- ***ElementNotFound*** – engine not found

- **CreateElementFailed** – create the task failed

Returns the task

Return type *SGInfoTask*

class smc.administration.scheduled_tasks.**ScheduledTaskMixin**
Bases: object

Actions common to all scheduled tasks.

add_schedule(*name*, *activation_date*, *day_period='one_time'*, *final_action='ALERT_FAILURE'*, *activated=True*, *minute_period='one_time'*, *day_mask=None*, *repeat_until_date=None*, *comment=None*)
Add a schedule to an existing task.

Parameters

- **name** (*str*) – name for this schedule
- **activation_date** (*int*) – when to start this task. Activation date should be a UTC time represented in milliseconds.
- **day_period** (*str*) – when this task should be run. Valid options: 'one_time', 'daily', 'weekly', 'monthly', 'yearly'. If 'daily' is selected, you can also provide a value for 'minute_period'. (default: 'one_time')
- **minute_period** (*str*) – only required if day_period is set to 'daily'. Valid options: 'each_quarter' (15 min), 'each_half' (30 minutes), or 'hourly', 'one_time' (default: 'one_time')
- **day_mask** (*int*) – If the task day_period=weekly, then specify the day or days for repeating. Day masks are: sun=1, mon=2, tue=4, wed=8, thu=16, fri=32, sat=64. To repeat for instance every Monday, Wednesday and Friday, the value must be 2 + 8 + 32 = 42
- **final_action** (*str*) – what type of action to perform after the scheduled task runs. Options are: 'ALERT_FAILURE', 'ALERT', or 'NO_ACTION' (default: ALERT_FAILURE)
- **activated** (*bool*) – whether to activate the schedule (default: True)
- **repeat_until_date** (*str*) – if this is anything but a one time task run, you can specify the date when this task should end. The format is the same as the *activation_date* param.
- **comment** (*str*) – optional comment

Raises **ActionCommandFailed** – failed adding schedule

Returns None

resources
Resources associated with this task. Depending on the task, this may be engines, policies, servers, etc.

Returns list of Elements

Return type list

start()
Start the scheduled task now. Task can then be tracked by using common Task methods.

Raises **ActionCommandFailed** – failed starting task

Returns return as a generic Task

Return type *Task*

task_schedule
Return any task schedules associated with this scheduled task.

Raises ***ActionCommandFailed*** – failure to retrieve task schedule

Returns list of task schedules

Return type *TaskSchedule*

class smc.administration.scheduled_tasks.**ServerBackupTask**(*name*, ***meta*)
Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A task that will back up the Management Server/s, Log Server/s and optionally the Log Server data.

Variables **log_data_must_be_saved** (*bool*) – whether to back up logs

classmethod create(*name*, *servers*, *backup_log_data=False*, *encrypt_password=None*, *comment=None*)
Create a new server backup task. This task provides the ability to backup individual or all management and log servers under SMC management.

Parameters

- **name** (*str*) – name of task
- **servers** (*list(ManagementServer or LogServer)*) – servers to back up. Servers must be instances of management servers or log servers. If no value is provided all servers are backed up.
- **backup_log_data** (*bool*) – Should the log files be backed up. This field is only relevant if a Log Server is backed up.
- **encrypt_password** (*str*) – Provide an encrypt password if you want this backup to be encrypted.
- **comment** (*str*) – optional comment

Raises

- ***ElementNotFound*** – specified servers were not found
- ***CreateElementFailed*** – failure to create the task

Returns the task

Return type *ServerBackupTask*

class smc.administration.scheduled_tasks.**SystemSnapsotTask**(*name*, ***meta*)
Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

A read-only task that will make a snapshot of all system elements after a updating a dynamic package on SMC.

class smc.administration.scheduled_tasks.**TaskSchedule**(***meta*)
Bases: *smc.base.model.SubElement*

A task schedule is associated with a given task type that defines when the scheduled task should run.

Variables

- **day_period** (*str*) – how often to run the task
- **final_action** (*str*) – what to do when the task is complete
- **minute_period** (*str*) – if day_period is set to hourly, when to run within the hour.

activate()
If a task is suspended, this will re-activate the task. Usually it's best to check for activated before running this:

```
task = RefreshPolicyTask('mytask')
for scheduler in task.task_schedule:
    if scheduler.activated:
        scheduler.suspend()
    else:
        scheduler.activate()
```

activated
Whether this schedule is active for this task.

Return type bool

activation_date
Return the UTC time when the task is set to first run. The activation date is returned as a python datetime object.

Returns datetime object in format '%Y-%m-%d %H:%M:%S.%f'

Return type datetime.datetime

suspend()
Suspend this scheduled task.

Raises *ActionCommandFailed* – failed to suspend, already suspended. Call activate on this task to reactivate.

Returns None

class smc.administration.scheduled_tasks.**UploadPolicyTask**(*name*, ***meta*)
Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

An upload policy task will assign a specified policy to an engine or group of engines and upload. If an engine specified has an existing policy assigned, the engine will be reassigned the specified policy. If the intent is to create a policy task to push an existing assigned policy, use *RefreshPolicyTask* instead.

Note: Policy upload on an engine can be done from the engine node itself by calling engine.upload('policy_name') or from a policy directly by policy.upload('engine_name').

---

classmethod create(*name*, *engines*, *policy*, *comment=None*, *validate_policy=False*, ***kwargs*)
Create an upload policy task associated with specific engines. A policy reassigns any policies that might be assigned to a specified engine.

Parameters

- **name** (*str*) – name of this task
- **engines** (*list (Engine)*) – list of Engines for the task
- **policy** (Policy) – Policy to assign to the engine/s
- **comment** (*str*) – optional comment
- **validate_policy** (*bool*) – validate the policy before upload. If set to true, validation kwargs can also be provided if customization is required, otherwise default validation settings are used.

- **kwargs** – see *policy_validation_settings()* for keyword arguments and default values.

Raises

- ***ElementNotFound*** – engine or policy specified does not exist
- ***CreateElementFailed*** – failure to create the task

Returns the task

Return type *UploadPolicyTask*

class smc.administration.scheduled_tasks.**ValidatePolicyTask**(*name*, ***meta*)

Bases: *smc.administration.scheduled_tasks.ScheduledTaskMixin*, *smc.base.model.Element*

Run a policy validation task. This does not perform a policy push. This may be useful if you want to validate any pending changes before a future policy push.

Variables ***policy*** (Element) – The policy associated with this task

classmethod create(*name*, *engines*, *policy=None*, *comment=None*, ***kwargs*)

Create a new validate policy task. If a policy is not specified, the engines existing policy will be validated. Override default validation settings as kwargs.

Parameters

- **name** (*str*) – name of task
- **engines** (*list(Engine)*) – list of engines to validate
- **policy** (Policy) – policy to validate. Uses the engines assigned policy if none specified.
- **kwargs** – see *policy_validation_settings()* for keyword arguments and default values.

Raises

- ***ElementNotFound*** – engine or policy specified does not exist
- ***CreateElementFailed*** – failure to create the task

Returns the task

Return type *ValidatePolicyTask*

smc.administration.scheduled_tasks.**log_target_types**(*all_logs=False*, ***kwargs*)

Log targets for log tasks. A log target defines the log types that will be affected by the operation. For example, when creating a DeleteLogTask, you can specify which log types are deleted.

Parameters

- **for_alert_event_log** (*bool*) – alert events traces (default: False)
- **for_alert_log** (*bool*) – alerts (default: False)
- **for_fw_log** (*bool*) – FW logs (default: False)
- **for_ips_log** (*bool*) – IPS logs (default: False)
- **for_ips_recording** (*bool*) – any IPS pcaps (default: False)
- **for_l2fw_log** (*bool*) – layer 2 FW logs (default: False)
- **for_third_party_log** (*bool*) – any 3rd party logs (default: False)

Returns dict of log targets

smc.administration.scheduled_tasks.**policy_validation_settings**(***kwargs*)

Set policy validation settings. This is used when policy based tasks are created and *validate_policy* is set to True. The following kwargs can be overridden in the create constructor.

Parameters

- **configuration_validation_for_alert_chain** (*bool*) – default False
- **duplicate_rule_check_settings** (*bool*) – default False
- **empty_rule_check_settings** (*bool*) – default True
- **emtpy_rule_check_settings_for_alert** (*bool*) – default False
- **general_check_settings** (*bool*) – default True
- **nat_modification_check_settings** (*bool*) – default True
- **non_supported_feature** (*bool*) – default True
- **routing_modification_check** (*bool*) – default False
- **unreachable_rule_check_settings** (*bool*) – default False
- **vpn_validation_check_settings** (*bool*) – default True

Returns dict of validation settings

13.3.6 Reports

New in version 0.6.0: Requires SMC version >= 6.3

Reports generated from the SMC. Provides an interface to running existing report designs and exporting their contents.

Example usage:

```
>>> from smc.administration.reports import ReportDesign, ReportTemplate, Report
```

List all available report templates:

```
>>> list(ReportTemplate.objects.all())
[ReportTemplate(name=Firewall Weekly Summary),
 ReportTemplate(name=Firewall Daily Summary from Specific Firewall),
 ReportTemplate(name=Firewall Multi-Link Usage)
 ...
```

Create a report design using an existing report template:

```
>>> template = ReportTemplate('Firewall Weekly Summary')
>>> template.create_design('myfirewallreport')
ReportDesign(name=myfirewallreport)
```

Generate a report based on an existing or created report design:

```
>>> list(ReportDesign.objects.all())
[ReportDesign(name=Application and Web Security), ReportDesign(name=myfirewallreport)]
...
>>> design = ReportDesign('Application and Web Security')
>>> poller = design.generate(wait_for_finish=True)
>>> while not poller.done():
```

(continues on next page)

(continued from previous page)

```
...   poller.wait(3)
...
>>> poller.task.resource
>>> Report(name=Application and Web Security #1515295820751)
...
>>> design.report_files
[Report(name=Application and Web Security #1515295820751), Report(name=Application␣
↪and Web Security #1515360776422)]
>>> report = Report('Application and Web Security #1515360776422')
>>> print(report.creation_time)
2018-01-07 15:32:56.422000
>>> report.export_pdf(filename='/foo/bar/a.pdf')
```

class smc.administration.reports.**Report**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Report represent a report that has been generated and that is currently stored on the SMC. These reports can be exported in multiple formats.

creation_time

When this report was generated. Using local time.

Return type datetime.datetime

export_pdf(*filename*)

Export the report in PDF format. Specify a path for which to save the file, including the trailing filename.

Parameters **filename** (*str*) – path including filename

Returns None

export_text(*filename=None*)

Export in text format. Optionally provide a filename to save to.

Parameters **filename** (*str*) – path including filename (optional)

Returns None

period_begin

Period when this report was specified to start.

Return type datetime.datetime

period_end

Period when this report was specified to end.

Return type datetime.datetime

class smc.administration.reports.**ReportDesign**(*name*, ***meta*)

Bases: *smc.base.model.Element*

A ReportDesign defines a report available in the SMC. This class provides access to generating these reports and exporting into a format supported by the SMC. Example of generating a report, and providing a callback once the report is complete which exports the report:

```
>>> def export_my_report(task):
...   if task.resource:
...     report = task.resource[0]
...     print("My report reference: %s" % report)
...     report.export_pdf('/Users/foo/myfile.pdf')
...
```

(continues on next page)

(continued from previous page)

```
>>>
>>> report = ReportDesign('Application and Web Security')
>>> poller = report.generate(wait_for_finish=True)
>>> poller.add_done_callback(export_my_report)
>>> while not poller.done():
...   poller.wait(3)
...
My report reference: Report(name=Application and Web Security #1515375369483)
```

generate(*start_time=0*, *end_time=0*, *senders=None*, *wait_for_finish=False*, *timeout=5*, ***kw*)

Generate the report and optionally wait for results. You can optionally add filters to the report by providing the senders argument as a list of type Element:

```
report = ReportDesign('Firewall Weekly Summary')
begin = datetime_to_ms(datetime.strptime("2018-02-03T00:00:00", "%Y-%m-%dT%H:
↪%M:%S"))
end = datetime_to_ms(datetime.strptime("2018-02-04T00:00:00", "%Y-%m-%dT%H:%M:
↪%S"))
report.generate(start_time=begin, end_time=end, senders=[Engine('vm')])
```

Parameters

- **period_begin** (*int*) – milliseconds time defining start time for report
- **period_end** (*int*) – milliseconds time defining end time for report
- **senders** (*list (Element)*) – filter targets to use when generating report
- **wait_for_finish** (*bool*) – enable polling for results
- **timeout** (*int*) – timeout between polling

Raises ***TaskRunFailed*** – refresh failed, possibly locked policy

Return type *TaskOperationPoller*

report_files

Retrieve all reports that are currently available on the SMC.

Return type list(*Report*)

class smc.administration.reports.**ReportTemplate**(*name*, ***meta*)

Bases: *smc.base.model.Element*

A report template represents an existing template in the SMC. Templates can be retrieved through the normal collections:

```
>>> list(ReportTemplate.objects.all())
[ReportTemplate(name=Firewall Weekly Summary),
 ReportTemplate(name=Firewall Daily Summary from Specific Firewall),
 ReportTemplate(name=Firewall Multi-Link Usage)
 ...
```

Once a report template of interest is identified, you can create a ReportDesign using that template:

```
>>> template = ReportTemplate('Firewall Weekly Summary')
>>> template.create_design('myfirewallreport')
ReportDesign(name=myfirewallreport)
```

create_design(*name*)
Create a report design based on an existing template.

Parameters **name** (*str*) – Name of new report design

Raises ***CreateElementFailed*** – failed to create template

Return type *ReportDesign*

13.3.7 System

Module that controls aspects of the System itself, such as updating dynamic packages, updating engines, applying global blacklists, etc.

To load the configuration for system, do:

```
>>> from smc.administration.system import System
>>> system = System()
>>> system.smc_version
'6.2.0 [10318]'
>>> system.last_activated_package
'881'
>>> for pkg in system.update_package():
...   print(pkg)
...
UpdatePackage(name=Update Package 889)
UpdatePackage(name=Update Package 888)
UpdatePackage(name=Update Package 887)
```

class smc.administration.system.**AdminDomain**(*name*, ***meta*)
Administrative domain element. Domains are used to provide object based segmentation within SMC. If domains are in use, you can log in directly to a domain to modify contents within that domain.

Find all available domains:

```
>>> list(AdminDomain.objects.all())
[AdminDomain(name=Shared Domain)]
```

Note: Admin Domains require and SMC license.

classmethod create(*name*, *comment=None*)
Create a new Admin Domain element for SMC objects.

Example:

```
>>> AdminDomain.create(name='mydomain', comment='mycomment')
>>> AdminDomain(name=mydomain)
```

Parameters

- **name** (*str*) – name of domain
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *AdminDomain*

class smc.administration.system.**System**
System level operations such as SMC version, time, update packages, and updating engines

active_alerts_ack_all()
Acknowledge all active alerts in the SMC. Only valid for SMC version >= 6.2.

Raises *ActionCommandFailed* – Failure during acknowledge with reason

Returns None

blacklist(*src*, *dst*, *duration=3600*, ***kw*)
Add blacklist to all defined engines. Use the cidr netmask at the end of src and dst, such as: 1.1.1.1/32, etc.

Parameters

- **src** – source of the entry
- **dst** – destination of blacklist entry

Raises *ActionCommandFailed* – blacklist apply failed with reason

Returns None

See also:

smc.core.engine.Engine.blacklist. Applying a blacklist at the system level will be a global blacklist entry versus an engine specific entry.

Note: If more advanced blacklist is required using source/destination ports and protocols (udp/tcp), use kw to provide these arguments. See *smc.elements.other.prepare_blacklist()* for more details.

---

empty_trash_bin()
Empty system level trash bin

Raises *ActionCommandFailed* – failed removing trash

Returns None

engine_upgrade()
List all engine upgrade packages available

To find specific upgrades available from the returned collection, use convenience methods:

```
system = System()
upgrades = system.engine_upgrade()
upgrades.get_contains('6.2')
upgrades.get_all_contains('6.2')
```

Parameters **engine_version** – Version of engine to retrieve

Raises *ActionCommandFailed* – failure to retrieve resource

Return type *SubElementCollection*(*EngineUpgrade*)

export_elements(*filename='export_elements.zip'*, *typeof='all'*)
Export elements from SMC.

Valid types are: all (All Elements)|nw (Network Elements)|ips (IPS Elements)| sv (Services)|rb (Security Policies)|al (Alerts)| vpn (VPN Elements)

Parameters

- **type** – type of element
- **filename** – Name of file for export

Raises *TaskRunFailed* – failure during export with reason

Return type *DownloadTask*

import_elements(*import_file*)
Import elements into SMC. Specify the fully qualified path to the import file.

Parameters **import_file** (*str*) – system level path to file

Raises ActionCommandFailed

Returns None

last_activated_package
Return the last activated package by id

Raises *ActionCommandFailed* – failure to retrieve resource

license_check_for_new()
Launch the check and download of licenses on the Management Server. This task can be long so call returns immediately.

Raises *ActionCommandFailed* – failure to retrieve resource

license_details()
This represents the license details for the SMC. This will include information with regards to the POL/POS, features, type, etc

Raises *ActionCommandFailed* – failure to retrieve resource

Returns dictionary of key/values

license_fetch(*proof_of_serial*)
Request a license download for the specified POS (proof of serial).

Parameters **proof_of_serial** (*str*) – proof of serial number of license to fetch

Raises *ActionCommandFailed* – failure to retrieve resource

license_install(*license_file*)
Install a new license.

Parameters **license_file** (*str*) – fully qualified path to the license jar file.

Raises ActionCommandFailed

Returns None

licenses
List of all engine related licenses This will provide details related to whether the license is bound, granted date, expiration date, etc.

```
>>> for license in system.licenses:
...     if license.bound_to.startswith('Management'):
...         print(license.proof_of_license)
abcd-efgh-ijkl-mnop
```

Raises *ActionCommandFailed* – failure to retrieve resource

Return type list(*Licenses*)

mgt_integration_configuration
Retrieve the management API configuration for 3rd party integration devices.

Raises *ActionCommandFailed* – failure to retrieve resource

references_by_element (*element_href*)
Return all references to element specified.

Parameters **element_href** (*str*) – element reference

Returns list of references where element is used

Return type list(dict)

smc_time
Return the SMC time as datetime object in UTC

:rtype datetime

smc_version
Return the SMC version

system_properties ()
List of all properties applied to the SMC

Raises *ActionCommandFailed* – failure to retrieve resource

update_package ()
Show all update packages on SMC.

To find specific updates available from the returned collection, use convenience methods:

```
system = System()
updates = system.update_package()
updates.get_contains('1027')
```

Raises *ActionCommandFailed* – failure to retrieve resource

Return type *SubElementCollection*(*UpdatePackage*)

visible_security_group_mapping (*filter=None*)
Return all security groups assigned to VSS container types. This is only available on SMC >= 6.5.

Parameters **filter** (*str*) – filter for searching by name

Raises

- *ActionCommandFailed* – element not found on this version of SMC
- *ResourceNotFound* – unsupported method on SMC < 6.5

Returns dict

visible_virtual_engine_mapping (*filter=None*)
Mappings for master engines and virtual engines

Parameters **filter** (*str*) – filter to search by engine name

Raises *ActionCommandFailed* – failure to retrieve resource

Returns list of dict items related to master engines and virtual engine mappings

13.3.8 Tasks

Tasks will be fired when executing specific actions such as a policy upload, refresh, or making backups.

This module provides that ability to access task specific attributes and optionally poll for status of an operation.

An example of using a task poller when uploading an engine policy (use *wait_for_finish=True*):

```
engine = Engine('myfirewall')
poller = engine.upload(policy=fwpolicy, wait_for_finish=True)
while not poller.done():
    poller.wait(5)
    print("Task Progress {}%".format(poller.task.progress))
print(poller.last_message())
```

class smc.administration.tasks.**DownloadTask**(*filename*, *task*, ***kw*)
Bases: *smc.administration.tasks.TaskOperationPoller*

A download task handles tasks that have files associated, for example exporting an element to a specified file.

class smc.administration.tasks.**Task**(*task*)
Bases: *smc.base.model.SubElement*

Task representation. This is generic and the format is used for any calls to SMC that return an asynchronous follower link to check the status of the task.

Parameters

- **last_message** (*str*) – Last message received on this task
- **in_progress** (*bool*) – Whether the task is in progress or finished
- **success** (*bool*) – Whether the task succeeded or not
- **follower** (*str*) – Fully qualified path to the follower link to track this task.

abort()
Abort existing task.

Raises *ActionCommandFailed* – aborting task failed with reason

Returns None

end_time
Task end time in UTC datetime format

Return type datetime

progress
Percentage of completion

Return type int

resource
The resource/s associated with this task

Return type list(*Element*)

result_url
Link to result (this task)

Return type str

start_time
Task start time in UTC datetime format

Return type datetime

update_status()
Gets the current status of this task and returns a new task object.

Raises ***TaskRunFailed*** – fail to update task status

smc.administration.tasks.**TaskHistory**()
Task history retrieves a list of tasks in an event queue.

Returns list of task events

Return type list(*TaskProgress*)

class smc.administration.tasks.**TaskOperationPoller**(*task*, *timeout=5*, *max_tries=36*, *wait_for_finish=False*)
Bases: object

Task Operation Poller provides a way to poll the SMC for the status of the task operation. This is returned by functions that return a task. Typically these will be operations like refreshing policy, uploading policy, etc.

add_done_callback(*callback*)
Add a callback to run after the task completes. The callable must take 1 argument which will be the completed Task.

Parameters **callback** – a callable that takes a single argument which will be the completed Task.

done()
Is the task done yet

Return type bool

last_message(*timeout=5*)
Wait a specified amount of time and return the last message from the task

Return type str

result(*timeout=None*)
Return the current Task after waiting for timeout

Return type *Task*

stop()
Stop the running task

task
Access to task

Return type *Task*

wait(*timeout=None*)
Blocking wait for task status.

class smc.administration.tasks.**TaskProgress**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Task Progress represents a task event queue. These tasks may be completed or still running. The task event queue events can be retrieved by calling *TaskHistory()*.

task
Return the task associated with this event

Return type *Task*

13.3.9 Updates

Functionality related to updating dynamic update packages and engine upgrades

class smc.administration.updates.**PackageMixin**
Manages downloads and activations of update packages and software upgrades

activate(*resource=None*, *timeout=3*, *wait_for_finish=False*)
Activate this package on the SMC

Parameters

- **resource** (*list*) – node href's to activate on. Resource is only required for software upgrades
- **timeout** (*int*) – timeout between queries

Raises ***TaskRunFailed*** – failure during activation (downloading, etc)

Return type *TaskOperationPoller*

download(*timeout=5*, *wait_for_finish=False*)
Download Package or Engine Update

Parameters **timeout** (*int*) – timeout between queries

Raises ***TaskRunFailed*** – failure during task status

Return type *TaskOperationPoller*

release_notes
HTTP location of the release notes

13.3.9.1 Engine Upgrade

class smc.administration.updates.**EngineUpgrade**(***meta*)
Bases: *smc.administration.updates.PackageMixin*, *smc.base.model.SubElement*

Engine Upgrade package management

For example, to check engine upgrades and find a specific one, then download for installation:

```
system = System()
upgrades = system.engine_upgrade()
package = upgrades.get_contains('6.2')

poller = package.download(wait_for_finish=True)
while not poller.done():
    print(poller.result(3))
print("Finished download: %s" % poller.result())
package.activate()
```

platform
Platform for this engine upgrade

release_date
Release date for this engine upgrade

version
Engine upgrade version

13.3.9.2 Dynamic Update

class smc.administration.updates.**UpdatePackage**(**meta*)
Bases: *smc.administration.updates.PackageMixin*, *smc.base.model.SubElement*

Container for managing update packages on SMC

Download and activate a package:

```
system = System()
packages = system.update_package()
dynup = packages.get_contains('1007')

poller = dynup.download(wait_for_finish=True)
while not poller.done():
    print(poller.result(3))
print("Finished download: %s" % poller.result())
package.activate()
```

activation_date
Date this update was activated, if any

Return type str

package_id
ID of the package. These will increment as new versions are released.

Return type str

release_date
Date of release

Return type str

state
State of this package as string. Valid states are available, imported, active. If the package is available, you can execute a download. If the package is imported, you can activate.

Return type str

13.4 Elements

Elements used for various configuration areas within SMC. Element types are made up of network, service groups and other.

13.4.1 Network

Module representing network elements used within the SMC

13.4.1.1 Alias

class smc.elements.network.**Alias**(*name*, **meta*)
Bases: *smc.base.model.Element*

Aliases are adaptive objects that represent a single element having different values based on the engine applied on. There are many default aliases in SMC and new ones can also be created.

Finding aliases can be achieved by using collections or loading directly if you know the alias name:

```
>>> from smc.elements.network import Alias
>>> list(Alias.objects.all())
[Alias(name=$$ Interface ID 46.net), Alias(name=$$ Interface ID 45.net), etc]
```

Resolve an alias to a specific engine:

```
>>> alias = Alias('$$ Interface ID 0.ip')
>>> alias.resolve('myfirewall')
[u'10.10.0.1']
```

Create an alias and assign values specific to an engine:

```
>>> alias = Alias.update_or_create(
    name='fooalias', engine=Layer3Firewall('vm'), translation_values=[Host('foo
↪')])
>>> alias
Alias(name=fooalias)
```

classmethod create(*name*, *comment=None*)
Create an alias.

Parameters

- **name** (*str*) – name of alias
- **comment** (*str*) – comment for this alias

Raises ***CreateElementFailed*** – create failed with reason

Return type *Alias*

resolve(*engine*)
Resolve this Alias to a specific value. Specify the engine by name to find it's value.

```
alias = Alias('$$ Interface ID 0.ip')
alias.resolve('smcpython-fw')
```

Parameters **engine** (*str*) – name of engine to resolve value

Raises ***ElementNotFound*** – if alias not found on engine

Returns alias resolving values

Return type list

resolved_value = None
resolved value for alias

classmethod update_or_create(*name*, *engine*, *translation_values=None*, *with_status=False*)
Update or create an Alias and it's mappings.

Parameters

- **name** (*str*) – name of alias
- **engine** (Engine) – engine to modify alias translation values
- **translation_values** (*list(str,Element)*) – translation values as elements. Can be None if you want to unset any existing values

- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***ElementNotFound*** – specified engine or translation values are not found in the SMC
- ***UpdateElementFailed*** – update failed with reason
- ***CreateElementFailed*** – create failed with reason

Return type *Element*

13.4.1.2 AddressRange

class smc.elements.network.**AddressRange**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Class representing a IpRange object used in access rules

Create an address range element:

```
IpRange.create('myrange', '1.1.1.1-1.1.1.5')
```

Available attributes:

Variables ***ip_range*** (*str*) – IP range for element. In format: '10.10.10.1-10.10.10.10'

classmethod create(*name*, *ip_range*, *comment=None*)

Create an AddressRange element

Parameters

- **name** (*str*) – Name of element
- **iprange** (*str*) – iprange of element
- **comment** (*str*) – comment (optional)

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *AddressRange*

13.4.1.3 DomainName

class smc.elements.network.**DomainName**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Represents a domain name used as FQDN in policy Use this object to reference a DNS resolvable FQDN or partial domain name to be used in policy.

Create a domain based network element:

```
DomainName.create('mydomain.net')
```

classmethod create(*name*, *comment=None*)

Create domain name element

Parameters **name** (*str*) – name of domain, i.e. lepages.net, www.lepages.net

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *DomainName*

13.4.1.4 Expression

class smc.elements.network.**Expression**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Expressions are used to build boolean like objects used in policy. For example, if you wanted to create an expression that negates a specific set of network elements to use in a "NOT" rule, an expression would be the element type.

For example, adding a rule that negates (network A or network B):

```
sub_expression = Expression.build_sub_expression(
                    name='mytestexporession',
                    ne_ref=['http://172.18.1.150:8082/6.0/elements/host/3999',
                            'http://172.18.1.150:8082/6.0/elements/host/4325'],
                    operator='union')

Expression.create(name='apiexpression',
                  ne_ref=[],
                  sub_expression=sub_expression)
```

Note: The sub-expression creates the json for the expression (network A or network B) and is then used as an parameter to create.

static build_sub_expression(*name*, *ne_ref=None*, *operator='union'*)

Static method to build and return the proper json for a sub-expression. A sub-expression would be the grouping of network elements used as a target match. For example, (network A or network B) would be considered a sub-expression. This can be used to compound sub-expressions before calling create.

Parameters

- **name** (*str*) – name of sub-expression
- **ne_ref** (*list*) – network elements references
- **operator** (*str*) – exclusion (negation), union, intersection (default: union)

Returns JSON of subexpression. Use in *create()* constructor

classmethod create(*name*, *ne_ref=None*, *operator='exclusion'*, *sub_expression=None*, *comment=None*)

Create the expression

Parameters

- **name** (*str*) – name of expression
- **ne_ref** (*list*) – network element references for expression
- **operator** (*str*) – 'exclusion' (negation), 'union', 'intersection' (default: exclusion)
- **sub_expression** (*dict*) – sub expression used
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *Expression*

13.4.1.5 Host

class smc.elements.network.**Host**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Class representing a Host object used in access rules

Create a host element with ipv4:

```
Host.create(name='myhost', address='1.1.1.1',
            secondary=['1.1.1.2'],
            comment='some comment for my host')
```

Create a host element with ipv6 and secondary ipv4 address:

```
Host.create(name='mixedhost',
            ipv6_address='2001:cdba::3257:9652',
            secondary=['1.1.1.1'])
```

Available attributes:

Variables

- **address** (*str*) – IPv4 address for this element
- **ipv6_address** (*str*) – IPv6 address for this host element
- **secondary** (*list*) – secondary IP addresses for this host

add_secondary(*address*, *append_lists=False*)

Add secondary IP addresses to this host element. If append_list is True, then add to existing list. Otherwise overwrite.

Parameters

- **address** (*list*) – ip addresses to add in IPv4 or IPv6 format
- **append_list** (*bool*) – add to existing or overwrite (default: append)

Returns None

classmethod create(*name*, *address=None*, *ipv6_address=None*, *secondary=None*, *comment=None*)

Create the host element

Parameters

- **name** (*str*) – Name of element
- **address** (*str*) – ipv4 address of host object (optional if ipv6)
- **ipv6_address** (*str*) – ipv6 address (optional if ipv4)
- **secondary** (*list*) – secondary ip addresses (optional)
- **comment** (*str*) – comment (optional)

Raises *CreateElementFailed* – element creation failed with reason

Returns instance with meta

Return type *Host*

Note: Either ipv4 or ipv6 address is required

13.4.1.6 IPList

class smc.elements.network.**IPList**(*name*, ***meta*)

Bases: *smc.base.model.Element*

IPList represent a custom list of IP addresses, networks or ip ranges (IPv4 or IPv6). These are used in source/destination fields of a rule for policy enforcement.

Note: IPList requires SMC API version >= 6.1

Create an empty IPList:

```
IPList.create(name='mylist')
```

Create an IPList with initial content:

```
IPList.create(name='mylist', iplist=['1.1.1.1','1.1.1.2', '1.2.3.4'])
```

Example of downloading the IPList in text format:

```
>>> iplist = list(IPList.objects.filter('mylist'))
>>> print(iplist)
[IPList(name=mylist)]
>>> iplist[0].download(filename='iplist.txt', as_type='txt')
```

Example of uploading an IPList as a zip file:

```
>>> iplist = list(IPList.objects.filter('mylist'))
>>> print(iplist)
[IPList(name=mylist)]
iplist[0].upload(filename='/path/to/iplist.zip')
```

Upload an IPList using json format:

```
>>> iplist = IPList('mylist')
>>> iplist.upload(json={'ip': ['4.4.4.4']}, as_type='json')
```

classmethod create(*name*, *iplist=None*, *comment=None*)

Create an IP List. It is also possible to add entries by supplying a list of IPs/networks, although this is optional. You can also use upload/download to add to the iplist.

Parameters

- **name** (*str*) – name of ip list
- **iplist** (*list*) – list of ipaddress
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *IPList*

download (*filename=None*, *as_type='zip'*)
Download the IPList. List format can be either zip, text or json. For large lists, it is recommended to use zip encoding. Filename is required for zip downloads.

Parameters

- **filename** (*str*) – Name of file to save to (required for zip)
- **as_type** (*str*) – type of format to download in: txt,json,zip (default: zip)

Raises **IOError** – problem writing to destination filename

Returns None

iplist
Return a list representation of this IPList. This is not a recommended function if the list is extremely large. In that case use the download function in zip format.

Raises ***FetchElementFailed*** – Reason for retrieval failure

Return type list

classmethod update_or_create (*append_lists=True*, *with_status=False*, ***kwargs*)
Update or create an IPList.

Parameters

- **append_lists** (*bool*) – append to existing IP List
- **kwargs** (*dict*) – provide at minimum the name attribute and optionally match the create constructor values

Raises ***FetchElementFailed*** – Reason for retrieval failure

upload (*filename=None*, *json=None*, *as_type='zip'*)
Upload an IPList to the SMC. The contents of the upload are not incremental to what is in the existing IPList. So if the intent is to add new entries, you should first retrieve the existing and append to the content, then upload. The only upload type that can be done without loading a file as the source is as_type='json'.

Parameters

- **filename** (*str*) – required for zip/txt uploads
- **json** (*str*) – required for json uploads
- **as_type** (*str*) – type of format to upload in: txt|json|zip (default)

Raises

- **IOError** – filename specified cannot be loaded
- ***CreateElementFailed*** – element creation failed with reason

Returns None

13.4.1.7 Network

class smc.elements.network.**Network** (*name*, ***meta*)
Bases: *smc.base.model.Element*

Class representing a Network object used in access rules Network format should be CIDR based. It is recommended that when creating the network element, you use a naming convention that includes the network cidr

in the name, such as 'network-1.1.1.0/24'. This will simplify searches later and workaround the restriction that searches with '/' and '-' only match on the name field and not an actual attribute value.

Create an ipv4 network element:

```
Network.create('mynetwork', '2.2.2.0/24')
```

Create an ipv6 network element:

```
Network.create(name='mixednetwork', ipv6_network='fc00::/7')
```

Available attributes:

Variables

- **ipv4_network** (*str*) – IPv4 network, in format: 10.10.10.0/24
- **ipv6_network** (*str*) – IPv6 network

classmethod create(*name*, *ipv4_network=None*, *ipv6_network=None*, *comment=None*)

Create the network element

Parameters

- **name** (*str*) – Name of element
- **ipv4_network** (*str*) – network cidr (optional if ipv6)
- **ipv6_network** (*str*) – network cidr (optional if ipv4)
- **comment** (*str*) – comment (optional)

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *Network*

Note: Either an ipv4_network or ipv6_network must be specified

---

13.4.1.8 Router

class smc.elements.network.**Router**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Class representing a Router object used in access rules

Create a router element with ipv4 address:

```
Router.create('myrouter', '1.2.3.4', comment='my router comment')
```

Create a router element with ipv6 address:

```
Router.create(name='mixedhost',
              ipv6_address='2001:cdba::3257:9652')
```

Available attributes:

Variables

- **address** (*str*) – IPv4 address for this router

- **ipv6_address** (*str*) – IPv6 address for this router
- **secondary** (*list*) – list of additional IP's for this router

classmethod create(*name*, *address=None*, *ipv6_address=None*, *secondary=None*, *comment=None*)

Create the router element

Parameters

- **name** (*str*) – Name of element
- **address** (*str*) – ip address of host object (optional if ipv6)
- **ipv6_address** (*str*) – ipv6 address (optional if ipv4)
- **secondary** (*list*) – secondary ip address (optional)
- **comment** (*str*) – comment (optional)

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *Router*

Note: either ipv4 or ipv6 address is required

13.4.1.9 URLListApplication

class smc.elements.network.**URLListApplication**(*name*, ***meta*)

Bases: *smc.base.model.Element*

URL List Application represents a list of URL's (typically by domain) that allow for easy grouping for performing whitelist and blacklisting

Creating a URL List:

```
URLListApplication.create(
    name='whitelist',
    url_entry=['www.google.com', 'www.cnn.com'])
```

Note: URLListApplication requires SMC API version >= 6.1

Available attributes:

Variables **url_entry** (*list*) – URL entries as strings

classmethod create(*name*, *url_entry*, *comment=None*)

Create the custom URL list

Parameters

- **name** (*str*) – name of url list
- **url_entry** (*list*) – list of url's
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *URLListApplication*

13.4.1.10 Zone

class smc.elements.network.**Zone**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Class representing a zone used on physical interfaces and used in access control policy rules, typically in source and destination fields. Zones can be applied on multiple interfaces which would allow logical grouping in policy.

Create a zone:

```
Zone.create('myzone')
```

classmethod create(*name*, *comment=None*)

Create the zone element

Parameters

- **zone** (*str*) – name of zone
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *Zone*

13.4.1.11 Traffic Handlers (Netlinks)

NetLink elements are used to represent alternative routes that lead to the same destination IP addresses.

NetLinks usually represent Internet connections, but can be used for other communications links as well.

You can use a single Router if a single route is enough for routing traffic to a network through an interface or an aggregated link. If you want to create separate routes for traffic to a network through two or more interfaces, you must use NetLinks.

To use traffic handlers, you must first create the netlink type required, then add this to the engine routing node.

Creating a static netlink element:

```
StaticNetlink.create(
    name='netlink',
    gateway=Router('routerfoo'),
    network=[Network('mynetwork')],
    domain_server_address=['8.8.8.8', '8.8.4.4'],
    probe_address=['1.1.1.254'],
    comment='foobar')
```

Add the netlink to the desired routing interface:

```
engine = Engine('vm')
rnode = engine.routing.get(0) #interface 0
rnode.add_traffic_handler(
    netlink=StaticNetlink('mynetlink'),
    netlink_gw=[Router('myrtr')])
```

See also:

smc.core.route.Routing.add_traffic_handler

Creating Multilink's require that you first have StaticNetlink or DynamicNetlink elements. Once you have this created, you can create a multilink in a two step process.

First create the multilink members specifying the created netlinks. A multilink member encapsulates the creation process and collects the required information for each netlink such as ip_range to use for source NAT (static netlink only) and the network role:

```
member = MultilinkMember.create(
    StaticNetlink('netlink1'), ip_range='1.1.1.1-1.1.1.2', netlink_role='active')

member1 = MultilinkMember.create(
    StaticNetlink('netlink2'), ip_range='2.1.1.1-2.1.1.2', netlink_role='standby')
```

Then create the multilink specifying the multilink members:

```
Multilink.create(name='internet', multilink_members=[member, member1])
```

See also:

Multilink

class smc.elements.netlink.**DynamicNetlink**(*name*, ***meta*)

Bases: *smc.base.model.Element*

A Dynamic Netlink is automatically created when an interface is using DHCP to obtain it's network address. It is also possible to manually create a dynamic netlink.

Variables

- **input_speed** (*int*) – input speed in Kbps, used for ratio-based load-balancing
- **output_speed** (*int*) – output speed in Kbps, used for ratio-based load-balancing
- **probe_address** (*list*) – list of IP addresses to use as probing addresses to validate connectivity
- **standby_mode_period** (*int*) – Specifies the probe period when standby mode is used (in seconds)
- **standby_mode_timeout** (*int*) – probe timeout in seconds
- **active_mode_period** (*int*) – Specifies the probe period when active mode is used (in seconds)
- **active_mode_timeout** (*int*) – probe timeout in seconds
- **learn_dns_automatically** (*bool*) – whether to obtain the DNS server address from the DHCP lease

classmethod create(*name*, *input_speed=None*, *learn_dns_automatically=True*, *output_speed=None*, *provider_name=None*, *probe_address=None*, *standby_mode_period=3600*, *standby_mode_timeout=30*, *active_mode_period=5*, *active_mode_timeout=1*, *comment=None*)

Create a Dynamic Netlink.

Parameters

- **name** (*str*) – name of netlink Element
- **input_speed** (*int*) – input speed in Kbps, used for ratio-based load-balancing

- **output_speed** (*int*) – output speed in Kbps, used for ratio-based load-balancing
- **learn_dns_automatically** (*bool*) – whether to obtain DNS automatically from the DHCP interface
- **provider_name** (*str*) – optional name to identify provider for this netlink
- **probe_address** (*list*) – list of IP addresses to use as probing addresses to validate connectivity
- **standby_mode_period** (*int*) – Specifies the probe period when standby mode is used (in seconds)
- **standby_mode_timeout** (*int*) – probe timeout in seconds
- **active_mode_period** (*int*) – Specifies the probe period when active mode is used (in seconds)
- **active_mode_timeout** (*int*) – probe timeout in seconds

Raises ***CreateElementFailed*** – failure to create netlink with reason

Return type *DynamicNetlink*

Note: To monitor the status of the network links, you must define at least one probe IP address.

class smc.elements.netlink.**Multilink**(*name*, ***meta*)

Bases: *smc.base.model.Element*

You can use Multi-Link to distribute outbound traffic between multiple network connections and to provide High Availability and load balancing for outbound traffic.

Creating a multilink requires several steps:

- Create the static netlink/s
- Create the multilink using the netlinks
- Add the multilink to an outbound NAT rule

Create the static netlink:

```
StaticNetlink.create(
    name='isp1',
    gateway=Router('nexthop'),      # 10.10.0.1
    network=[Network('comcast')],   # 10.10.0.0/16
    probe_address=['10.10.0.1'])
```

Create the multilink members based on the pre-created netlinks. A multilink member specifies the ip range to use for source NAT, the role (active/standby) and obtains the defined network from the StaticNetlink:

```
member = MultilinkMember.create(
    StaticNetlink('netlink1'), ip_range='1.1.1.1-1.1.1.2', netlink_role='active')

member1 = MultilinkMember.create(
    StaticNetlink('netlink2'), ip_range='2.1.1.1-2.1.1.2', netlink_role='standby')
```

Create the multilink using the multilink members:

```
Multilink.create(name='internet', multilink_members=[member, member1])
```

Lastly, add a NAT rule with dynamic source nat using the multilink:

```
policy = FirewallPolicy('outbound')
policy.fw_ipv4_nat_rules.create(
    name='mynat',
    sources=[Network('mynetwork')],
    destinations='any',
    services='any',
    dynamic_src_nat=Multilink('internet'))
```

Note: Multi-Link is supported on Single Firewalls, Firewall Clusters, and Virtual Firewalls

classmethod create(*name*, *multilink_members*, *multilink_method='rtt'*, *retries=2*, *timeout=3600*, *comment=None*)

Create a new multilink configuration. Multilink requires at least one netlink for operation, although 2 or more are recommeneded.

Parameters

- **name** (*str*) – name of multilink
- **multilink_members** (*list*) – the output of calling `multilink_member()` to retrieve the proper formatting for this sub element.
- **multilink_method** (*str*) – 'rtt' or 'ratio'. If ratio is used, each netlink must have a probe IP address configured and also have input and output speed configured (default: 'rtt')
- **retries** (*int*) – number of keep alive retries before a destination link is considered unavailable (default: 2)
- **timeout** (*int*) – timeout between retries (default: 3600 seconds)
- **comment** (*str*) – comment for multilink (optional)

Raises ***CreateElementFailed*** – failure to create multilink

Return type *Multilink*

classmethod create_with_netlinks(*name*, *netlinks*, ***kwargs*)

Create a multilink with a list of StaticNetlinks. To properly create the multilink using this method, pass a list of netlinks with the following dict structure:

```
netlinks = [{'netlink': StaticNetlink,
             'ip_range': 1.1.1.1-1.1.1.2,
             'netlink_role': 'active'}]
```

The *netlink_role* can be either *active* or *standby*. The remaining settings are resolved from the StaticNetlink. The IP range value must be an IP range within the StaticNetlink's specified network. Use kwargs to pass any additional arguments that are supported by the *create* constructor. A full example of creating a multilink using predefined netlinks:

```
multilink = Multilink.create_with_netlinks(
    name='mynewnetlink',
    netlinks=[{'netlink': StaticNetlink('netlink1'),
               'ip_range': '1.1.1.2-1.1.1.3',
               'netlink_role': 'active'},
              {'netlink': StaticNetlink('netlink2'),
               'ip_range': '2.1.1.2-2.1.1.3',
               'netlink_role': 'standby'}])
```

Parameters

- **netlink** (StaticNetlink,DynamicNetlink) – StaticNetlink element
- **ip_range** (*str*) – ip range for source NAT on this netlink
- **netlink_role** (*str*) – the role for this netlink, *active* or *standby*

Raises ***CreateElementFailed*** – failure to create multilink

Return type *Multilink*

members

Multilink members associated with this multilink. This provides a a reference to the existing netlinks and their member settings.

Return type *MultilinkMember*

classmethod update_or_create(*with_status=False*, ***kwargs*)

Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to `get()`. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

class smc.elements.netlink.**MultilinkMember**(*kwargs*)

Bases: object

A multilink member represents an netlink member used on a multilink configuration. Multilink uses netlinks to specify settings specific to a connection, network, whether it should be active or standby and optionally QoS. Use this class to create mutlilink members that are required for creating a Multilink element.

Variables

- ***network*** (Network) – network element reference specifying netlink subnet
- ***netlink*** (StaticNetlink,DynamicNetlink) – netlink element reference

classmethod create(*netlink*, *ip_range=None*, *netlink_role='active'*)
Create a multilink member. Multilink members are added to an Outbound Multilink configuration and define the ip range, static netlink to use, and the role. This element can be passed to the Multilink constructor to simplify creation of the outbound multilink.

Parameters

- **netlink** (StaticNetlink,DynamicNetlink) – static netlink element to use as member
- **ip_range** (*str*) – the IP range for source NAT for this member. The IP range should be part of the defined network range used by this netlink. Not required for dynamic netlink
- **netlink_role** (*str*) – role of this netlink, 'active' or 'standby'

Raises ***ElementNotFound*** – Specified netlink could not be found

Return type *MultilinkMember*

ip_range
Specifies the IP address range for dynamic source address translation (NAT) for the internal source IP addresses on the NetLink. Can also be set.

Return type str

netlink_role
Shows whether the Netlink is active or standby. Active - traffic is routed through the NetLink according to the method you specify in the Outbound Multi-Link element properties. Standby - traffic is only routed through the netlink if all primary (active) netlinks are unavailable.

Return type str

class smc.elements.netlink.**StaticNetlink**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A Static Netlink is applied to an interface to provide an alternate route to a destination. It is typically used when you have fixed IP interfaces versus using DHCP (use a Dynamic NetLink).

Variables

- **gateway** (Router,Engine) – gateway for this netlink. Should be the 'next hop' element associated with the netlink
- ***network*** (*list* (Network)) – list of networks associated with this netlink
- **input_speed** (*int*) – input speed in Kbps, used for ratio-based load-balancing
- **output_speed** (*int*) – output speed in Kbps, used for ratio-based load-balancing
- **probe_address** (*list*) – list of IP addresses to use as probing addresses to validate connectivity
- **standby_mode_period** (*int*) – Specifies the probe period when standby mode is used (in seconds)
- **standby_mode_timeout** (*int*) – probe timeout in seconds
- **active_mode_period** (*int*) – Specifies the probe period when active mode is used (in seconds)
- **active_mode_timeout** (*int*) – probe timeout in seconds

classmethod create(*name*, *gateway*, *network*, *input_speed=None*, *output_speed=None*, *domain_server_address=None*, *provider_name=None*, *probe_address=None*, *standby_mode_period=3600*, *standby_mode_timeout=30*, *active_mode_period=5*, *active_mode_timeout=1*, *comment=None*)

Create a new StaticNetlink to be used as a traffic handler.

Parameters

- **name** (*str*) – name of netlink Element
- **gateway_ref** (Router,Engine) – gateway to map this netlink to. This can be an element or str href.
- **ref** (*list(str,Element)*) – network/s associated with this netlink.
- **input_speed** (*int*) – input speed in Kbps, used for ratio-based load-balancing
- **output_speed** (*int*) – output speed in Kbps, used for ratio-based load-balancing
- **domain_server_address** (*list*) – dns addresses for netlink. Engine DNS can override this field
- **provider_name** (*str*) – optional name to identify provider for this netlink
- **probe_address** (*list*) – list of IP addresses to use as probing addresses to validate connectivity
- **standby_mode_period** (*int*) – Specifies the probe period when standby mode is used (in seconds)
- **standby_mode_timeout** (*int*) – probe timeout in seconds
- **active_mode_period** (*int*) – Specifies the probe period when active mode is used (in seconds)
- **active_mode_timeout** (*int*) – probe timeout in seconds

Raises

- ***ElementNotFound*** – if using type Element parameters that are not found.
- ***CreateElementFailed*** – failure to create netlink with reason

Return type *StaticNetlink*

Note: To monitor the status of the network links, you must define at least one probe IP address.

---

domain_server_address

Configured DNS servers for this netlink

Returns list of DNS servers; if elements are specifed, they will be returned as type Element

Return type *RankedDNSAddress*

classmethod update_or_create(*with_status=False*, ***kwargs*)

Update or create static netlink. DNS entry differences are not resolved, instead any entries provided will be the final state for this netlink. If the intent is to add/remove DNS entries you can use the *domain_server_address()* method to add or remove.

Raises ***CreateElementFailed*** – failed creating element

Returns element instance by type or 3-tuple if with_status set

13.4.2 Services

Module providing service configuration and creation.

Some services may be generic services while others might provide more in depth functionality using protocol agents. A protocol agent provides layer 7 configuration capabilities specific to the protocol it defines. If a given service inherits the ProtocolAgentMixin, this service type is eligible to have a protocol agent attached.

See also:

smc.elements.protocols

class smc.elements.service.**ProtocolAgentMixin**
ProtocolAgentMixin is used by services that allow a protocol agent.

protocol_agent
Protocol Agent for this service

Returns Return the protocol agent or None if this service does not reference a protocol agent

Return type *ProtocolAgent*

protocol_agent_values
Protocol agent values are protocol specific settings configurable on a service when a protocol agent is assigned to that service. This property will return an iterable that represents each protocol specific parameter and it's value.

Return type *BaseIterable*(*ProtocolAgentValues*)

update_protocol_agent(*protocol_agent*)
Update this service to use the specified protocol agent. After adding the protocol agent to the service you must call *update* on the element to commit.

Parameters **protocol_agent** (*str*,ProtocolAgent) – protocol agent element or href

Returns None

13.4.2.1 EthernetService

class smc.elements.service.**EthernetService**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Represents an ethernet based service in SMC Ethernet service only supports adding Ethernet II frame type.

The value1 field should be the ethernet2 ethertype hex code which will be converted to decimal format.

Create an ethernet rule representing the presence of an IEEE 802.1Q tag:

```
>>> EthernetService.create(name='8021q frame', value1='0x8100')
EthernetService(name=8021q frame)
```

Note: Ethernet Services are only available as of SMC version 6.1.2

classmethod create(*name*, *frame_type='eth2'*, *value1=None*, *comment=None*)
Create an ethernet service

Parameters

- **name** (*str*) – name of service
- **frame_type** (*str*) – ethernet frame type, eth2

- **value1** (*str*) – hex code representing ethertype field
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *EthernetService*

13.4.2.2 ICMPService

class smc.elements.service.**ICMPService**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Represents an ICMP Service in SMC Use the RFC icmp type and code fields to set values. ICMP type is required, icmp code is optional but will make the service more specific if type codes exist.

Create an ICMP service using type 3, code 7 (Dest. Unreachable):

```
>>> ICMPService.create(name='api-icmp', icmp_type=3, icmp_code=7)
ICMPService(name=api-icmp)
```

Available attributes:

Variables

- **icmp_type** (*int*) – icmp type field
- **icmp_code** (*int*) – icmp type code

classmethod create(*name*, *icmp_type*, *icmp_code=None*, *comment=None*)

Create the ICMP service element

Parameters

- **name** (*str*) – name of service
- **icmp_type** (*int*) – icmp type field
- **icmp_code** (*int*) – icmp type code

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *ICMPService*

13.4.2.3 ICMPIPv6Service

class smc.elements.service.**ICMPIPv6Service**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Represents an ICMPv6 Service type in SMC Set the icmp type field at minimum. At time of writing the icmp code fields were all 0.

Create an ICMPv6 service for Neighbor Advertisement Message:

```
>>> ICMPIPv6Service.create('api-Neighbor Advertisement Message', 139)
ICMPIPv6Service(name=api-Neighbor Advertisement Message)
```

Available attributes:

Variables **icmp_type** (*int*) – ipv6 icmp type field

classmethod create(*name*, *icmp_type*, *comment=None*)
Create the ICMPIPv6 service element

Parameters

- **name** (*str*) – name of service
- **icmp_type** (*int*) – ipv6 icmp type field

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *ICMPIPv6Service*

13.4.2.4 IPService

class smc.elements.service.**IPService**(*name*, ***meta*)
Bases: *smc.elements.protocols.ProtocolAgentMixin*, *smc.base.model.Element*

Represents an IP-Proto service in SMC IP Service is represented by a protocol number. This will display in the SMC under Services -> IP-Proto. It may also show up in Services -> With Protocol if the protocol is tied to a Protocol Agent.

Create an IP Service for protocol 93 (AX.25):

```
>>> IPService.create('ipservice', 93)
IPService(name=ipservice)
```

Available attributes:

Variables ***protocol_number*** (*str*) – IP protocol number for this service

classmethod create(*name*, *protocol_number*, *protocol_agent=None*, *comment=None*)
Create the IP Service

Parameters

- **name** (*str*) – name of ip-service
- **protocol_number** (*int*) – ip proto number for this service
- **protocol_agent** (*str*,ProtocolAgent) – optional protocol agent for this service
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *IPService*

protocol_number
Protocol number for this IP Service

Return type int

13.4.2.5 TCPService

class smc.elements.service.**TCPService**(*name*, ***meta*)
Bases: *smc.elements.protocols.ProtocolAgentMixin*, *smc.base.model.Element*

Represents a TCP based service in SMC TCP Service can use a range of ports or single port. If using single port, set only min_dst_port. If using range, set both min_dst_port and max_dst_port.

Create a TCP Service for port 5000:

```
>>> TCPService.create('tcpservice', 5000, comment='my service')
TCPService(name=tcpservice)
```

Available attributes:

Variables

- **min_dst_port** (*int*) – starting destination port for this service. If the service is a single port service, use only this field
- **max_dst_port** (*int*) – used in conjunction with min_dst_port for creating a port range service.

classmethod create(*name*, *min_dst_port*, *max_dst_port=None*, *min_src_port=None*, *max_src_port=None*, *protocol_agent=None*, *comment=None*)
Create the TCP service

Parameters

- **name** (*str*) – name of tcp service
- **min_dst_port** (*int*) – minimum destination port value
- **max_dst_port** (*int*) – maximum destination port value
- **min_src_port** (*int*) – minimum source port value
- **max_src_port** (*int*) – maximum source port value
- **protocol_agent** (*str,ProtocolAgent*) – optional protocol agent for this service
- **comment** (*str*) – optional comment for service

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *TCPService*

13.4.2.6 UDPService

class smc.elements.service.**UDPService**(*name*, ***meta*)
Bases: *smc.elements.protocols.ProtocolAgentMixin*, *smc.base.model.Element*

UDP Services can use a range of ports or single port. If using single port, set only min_dst_port. If using range, set both min_dst_port and max_dst_port.

Create a UDP Service for port range 5000-5005:

```
>>> UDPService.create('udpservice', 5000, 5005)
UDPService(name=udpservice)
```

Available attributes:

Variables

- **min_dst_port** (*int*) – starting destination port for this service. If the service is a single port service, use only this field
- **max_dst_port** (*int*) – used in conjunction with min_dst_port for creating a port range service

classmethod create(*name*, *min_dst_port*, *max_dst_port=None*, *min_src_port=None*, *max_src_port=None*, *protocol_agent=None*, *comment=None*)

Create the UDP Service

Parameters

- **name** (*str*) – name of udp service
- **min_dst_port** (*int*) – minimum destination port value
- **max_dst_port** (*int*) – maximum destination port value
- **min_src_port** (*int*) – minimum source port value
- **max_src_port** (*int*) – maximum source port value
- **protocol_agent** (*str*,*ProtocolAgent*) – optional protocol agent for this service
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failure creating element with reason

Returns instance with meta

Return type *UDPService*

13.4.2.7 URLCategory

class smc.elements.service.**URLCategory**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Represents a URL Category for policy. URL Categories are read only. To make whitelist or blacklists, use *smc.elements.network.IPList*.

13.4.2.8 With Protocol

New in version 0.6.2: Requires SMC version >= 6.4.3

Protocols define elements within the SMC that are specified to Protocol Agents. Protocol Agents can be attached to specific services by type. If a service inherits the ProtocolAgentMixin, the service type is eligible to add a protocol agent.

An example of attaching a protocol agent to a generic TCP Service, in this case used as a custom HTTP service:

```
>>> TCPService.create(name='testservice', min_dst_port=8080,
        protocol_agent=ProtocolAgent('HTTP'), comment='foo')
TCPService(name=testservice)
```

You can optionally also add a protocol agent to an existing service if the service is already created:

```
>>> from smc.elements.protocols import ProtocolAgent
>>> service.update_protocol_agent(ProtocolAgent('HTTP'))
>>> service.update()
```

(continues on next page)

(continued from previous page)

```
...
>>> service.protocol_agent
ProtocolAgent(name=HTTP)
```

To make modifications on an existing Protocol Agent assigned to a service, you can iterate the protocol agent values to see the available parameter settings then call update on the same collection.

For example, to set the above service to redirect to a Proxy Server (CIS redirect), you can use this logic.

First view the available protocol agent parameters:

```
>>> service = TCPService('testservice')
>>> service.protocol_agent
ProtocolAgent(name=HTTP)
...
>>> for parameter in service.protocol_agent_values:
...     parameter
...
BooleanValue(name=http_enforce_safe_search,description=Enforce SafeSearch,value=0)
ProxyServiceValue(name=redir_cis,description=Redirect to Proxy Server,proxy_
↪server=None)
StringValue(name=http_server_stream_by_user_agent,description=Optimized server stream␣
↪fingerprinting,value=Yes)
StringValue(name=http_url_logging,description=Logging of accessed URLs,value=Yes)
```

The parameters returned all inherit from a base class template *ProtocolParameterValue*. Each returned parameter is generated dynamically based on the type of input expected for the given parameter/field type.

Note: The *description* field of the parameter matches what you would see in the SMC UI under the Protocol Parameters tab of a service using a ProtocolAgent. The *name* field is the internal name that you would use to reference the setting when calling *protocol_agent_values.update*.

We want to add the CIS (ProxyServer) redirect, so update is done on the 'redir_cis' (name) field:

```
>>> from smc.elements.servers import ProxyServer
>>> service.protocol_agent_values.update(name='redir_cis', proxy_server=ProxyServer(
↪'generic5'))
True
```

The update was successful, and we can now validate the parameter repr shows an assigned proxy:

```
>>> for parameter in service.protocol_agent_values:
...   parameter
...
BooleanValue(name=http_enforce_safe_search,description=Enforce SafeSearch,value=0)
ProxyServiceValue(name=redir_cis,description=Redirect to Proxy Server,proxy_
↪server=ProxyServer(name=generic5))
StringValue(name=http_server_stream_by_user_agent,description=Optimized server stream␣
↪fingerprinting,value=Yes)
StringValue(name=http_url_logging,description=Logging of accessed URLs,value=Yes)
```

Lastly, to commit this change to SMC, you must still call *update* on the service element:

```
service.update()
```

You can unset a ProxyServer by setting the proxy_server field to None and updating:

```
service.update_protocol_agent(None)
service.update()
```

class smc.elements.protocols.**ProtocolAgent**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Protocol Agents ensure that related connections for a service are properly grouped and evaluated by the engine, as well as assisting the engine with content filtering or network address translation tasks.

class smc.elements.protocols.**ProtocolAgentMixin**
Bases: object

ProtocolAgentMixin is used by services that allow a protocol agent.

protocol_agent
Protocol Agent for this service

Returns Return the protocol agent or None if this service does not reference a protocol agent

Return type *ProtocolAgent*

protocol_agent_values
Protocol agent values are protocol specific settings configurable on a service when a protocol agent is assigned to that service. This property will return an iterable that represents each protocol specific parameter and it's value.

Return type *BaseIterable*(*ProtocolAgentValues*)

update_protocol_agent(*protocol_agent*)
Update this service to use the specified protocol agent. After adding the protocol agent to the service you must call *update* on the element to commit.

Parameters **protocol_agent** (*str*, ProtocolAgent) – protocol agent element or href

Returns None

class smc.elements.protocols.**ProtocolAgentValues**(*protocol_agent*, *values*)
Bases: *smc.base.structs.BaseIterable*

Protocol Agent Values define settings that can be set for specific protocols when a protocol agent is referenced in a service.

This is a collection of parameters that are relevant based on the protocol agent type. This is called from the service itself when a service has a protocol agent attached. An example of iterating a given service with an HTTP protocol agent attached:

```
>>> from smc.elements.service import TCPService
>>> service = TCPService('mynewservice')
>>> service.protocol_agent
ProtocolAgent(name=HTTP)
>>> for parameter in service.protocol_agent_values:
...   parameter
...
BooleanValue(name=http_enforce_safe_search,description=Enforce SafeSearch,value=0)
ProxyServiceValue(name=redir_cis,description=Redirect to Proxy Server,proxy_
↪server=None)
StringValue(name=http_server_stream_by_user_agent,description=Optimized server␣
↪stream fingerprinting,value=Yes)
StringValue(name=http_url_logging,description=Logging of accessed URLs,value=Yes)
```

Each protocol agent parameter has a name value and description. The name is an internal name representation but the description is the value you would see within the SMC UI for the given field.

Each parameter class is dynamically generated based on the class template ProtocolParameterValue. The class name indicates the type of parameter value that is expected for the given field.

Return type *ProtocolParameterValue*

get (*parameter_name*)
Get the parameter by it's name. This is a convenience for fetching. For example, fetch the proxy server (redir_cis) parameter from a HTTP or HTTPS protocol agent:

```
pv = newservice.protocol_agent_values.get('redir_cis')
```

Returns Return the parameter value if it exists, otherwise None

Return type *ProtocolParameterValue*

update (*name*, ***kwargs*)
Update protocol agent parameters based on the parameter name. Provide the relevant keyword pairs based on the parameter type. When update is called, a boolean is returned indicating whether the field was successfully updated or not. You should check the return value and call *update* on the service to commit to SMC.

Example of updating a TCP Service using the HTTPS Protocol Agent to set an HTTPS Inspection Exception:

```
>>> service = TCPService('httpsservice')
>>> service.protocol_agent
ProtocolAgent(name=HTTPS)
>>> for parameter in service.protocol_agent_values:
...   parameter
...
ProxyServiceValue(name=redir_cis,description=Redirect connections to Proxy␣
↪Server,proxy_server=None)
BooleanValue(name=http_enforce_safe_search,description=Enforce SafeSearch,
↪value=0)
StringValue(name=http_server_stream_by_user_agent,description=Optimized␣
↪server stream fingerprinting,value=Yes)
StringValue(name=http_url_logging,description=Logging of accessed URLs,
↪value=Yes)
TlsInspectionPolicyValue(name=tls_policy,description=HTTPS Inspection␣
↪Exceptions,tls_policy=None)
StringValue(name=tls_inspection,description=HTTPS decryption and inspection,
↪value=No)
...
>>> service.protocol_agent_values.update(name='tls_policy', tls_
↪policy=HTTPSInspectionExceptions('myexceptions'))
True
```

Parameters

- **name** (*str*) – The name of the parameter to update
- **kwargs** (*dict*) – The keyword args to perform the update

Raises

- ***ElementNotFound*** – Can be thrown when an element reference was passed but the element does not exist
- ***MissingDependency*** – A dependency was missing preventing the update. This can happen when adding a ProxyServer for a protocol that isn't enabled

class smc.elements.protocols.**ProtocolParameterValue**
Bases: object

A ProtocolParameterValue defines a protocol agent parameter setting when a protocol agent is assigned to a service. There are multiple protocol parameter types and each protocol agent will have specific parameters depending on functionality.

Read only attributes are:

Variables

- ***protocol_agent*** (ProtocolAgent) – The protocol agent for this parameter value
- ***protocol_agent_values*** (*dict*) – The protocol agent values for this setting
- ***description*** (*str*) – The read-only description of this setting, used in SMC UI
- ***type*** (*str*) – The value type that this parameter is expected, i.e. string, integer, etc

Mutable attributes are:

Variables ***value*** (*str*) – The mutable value for this particular setting

description
Description of this protocol parameter. The description is what will be displayed on the service properties under the Protocol Parameters tab when a Protocol Agent is assigned to a service

Return type str

name
Name of this protocol setting

Return type str

type
The type of this parameter. Can be string value, integer value, etc. The type is returned as a string representation.

Return type str

value
The value for this given protocol parameter. The return type is defined by the *type* of parameter

Returns value based on *type* of parameter. Will return None if this parameter does not support the *value* key for this parameter

class smc.elements.protocols.**ProxyServiceValue**
Bases: *smc.elements.protocols.ProtocolParameterValue*

This represents a protocol parameter specific to setting a redirect to proxy setting on a service with a protocol agent.

Mutable attributes are:

Variables ***proxy_server*** (*str*) – The mutable value for this particular setting. Represents the ProxyServer element

proxy_server
The ProxyServer element referenced in this protocol parameter, if any.

Returns The proxy server element or None if one is not assigned

Return type *ProxyServer*

class smc.elements.protocols.**TlsInspectionPolicyValue**
Bases: *smc.elements.protocols.ProtocolParameterValue*

This represents HTTPS Inspection Exceptions that would be a parameter for a HTTPS Protocol Agent service.

Mutable attributes are:

Variables ***tls_policy*** (*str*) – The mutable value for this particular setting. Represents the HTTPS Inspection Exceptions element

tls_policy
The HTTPSInspectionExceptions element referenced in this protocol agent parameter. Will be None if one is not assigned.

Returns The https inspection exceptions element or None if not assigned

Return type *HTTPSInspectionExceptions*

13.4.3 Groups

Groups that are used for element types, such as TCPServiceGroup, Group (generic), etc. All group types inherit from GroupMixin which allow for modifications of existing groups and their members.

class smc.elements.group.**GroupMixin**
Methods associated with handling modification of Group objects for existing elements

empty_members()
Empty members from group

Returns None

members
Return members in raw href format. If you want to obtain a resolved list of elements as instance of Element, call *~obtain_members*.

Return type list

obtain_members()
Obtain all group members from this group

Returns group members as elements

Return type list(*Element*)

update_members(*members*, *append_lists=False*, *remove_members=False*)
Update group members with member list. Set append=True to append to existing members, or append=False to overwrite.

Parameters

- **members** (*list[str, Element]*) – new members for group by href or Element
- **append_lists** (*bool*) – whether to append
- **remove_members** (*bool*) – remove members from the group

Returns bool was modified or not

classmethod update_or_create(*append_lists=True*, *with_status=False*, *remove_members=False*, ***kwargs*)
Update or create group entries. If the group exists, the members will be updated. Set append_lists=True to add new members to the list, or False to reset the list to the provided members. If setting remove_members, this will override append_lists if set.

Parameters

- **append_lists** (*bool*) – add to existing members, if any
- **remove_members** (*bool*) – remove specified members instead of appending or overwriting

Paran dict kwargs keyword arguments to satisfy the *create* constructor if the group needs to be created.

Raises ***CreateElementFailed*** – could not create element with reason

Returns element instance by type

Return type *Element*

13.4.3.1 ICMPServiceGroup

class smc.elements.group.**ICMPServiceGroup**(*name*, ***meta*)

Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

IP Service Group Used for storing IP Services or IP Service Groups

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)

Create the IP Service group element

Parameters

- **name** (*str*) – name of service group
- **element** (*list*) – IP services or IP service groups by href

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *ICMPServiceGroup*

13.4.3.2 IPServiceGroup

class smc.elements.group.**IPServiceGroup**(*name*, ***meta*)

Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

IP Service Group Used for storing IP Services or IP Service Groups

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)

Create the IP Service group element

Parameters

- **name** (*str*) – name of service group
- **element** (*list*) – IP services or IP service groups by href

Raises *CreateElementFailed* – element creation failed with reason

Returns instance with meta

Return type *IPServiceGroup*

13.4.3.3 Group

class smc.elements.group.**Group**(*name*, ***meta*)

Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

Class representing a Group object used in access rules Groups can hold other network element types as well as other groups.

Create a group element:

```
Group.create('mygroup') #no members
```

Group with members:

```
Group.create('mygroup', [Host('kali'), Network('mynetwork')])
```

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)

Create the group

Parameters

- **name** (*str*) – Name of element
- **members** (*str,Element*) – group members by element names
- **comment** (*str*) – optional comment

Raises *CreateElementFailed* – element creation failed with reason

Returns instance with meta

Return type *Group*

13.4.3.4 ServiceGroup

class smc.elements.group.**ServiceGroup**(*name*, ***meta*)

Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

Represents a service group in SMC. Used for grouping objects by service. Services can be "mixed" TCP/UDP/ICMP/ IPService, Protocol or other Service Groups. Element is an href to the location of the resource.

Create a TCP and UDP Service and add to ServiceGroup:

```
tcp1 = TCPService.create('api-tcp1', 5000)
udp1 = UDPService.create('api-udp1', 5001)
ServiceGroup.create('servicegroup', element=[tcp1, udp1])
```

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)
Create the TCP/UDP Service group element

Parameters

- **name** (*str*) – name of service group
- **members** (*list (str,Element)*) – elements to add by href or Element

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *ServiceGroup*

13.4.3.5 TCPServiceGroup

class smc.elements.group.**TCPServiceGroup**(*name*, ***meta*)
Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

Represents a TCP Service group

Create TCP Services and add to TCPServiceGroup:

```
tcp1 = TCPService.create('api-tcp1', 5000)
tcp2 = TCPService.create('api-tcp2', 5001)
ServiceGroup.create('servicegroup', element=[tcp1, tcp2])
```

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)
Create the TCP Service group

Parameters

- **name** (*str*) – name of tcp service group
- **element** (*list (str,Element)*) – tcp services by element or href

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *TCPServiceGroup*

13.4.3.6 UDPServiceGroup

class smc.elements.group.**UDPServiceGroup**(*name*, ***meta*)
Bases: *smc.elements.group.GroupMixin*, *smc.base.model.Element*

UDP Service Group Used for storing UDP Services or UDP Service Groups.

Create two UDP Services and add to UDP service group:

```
udp1 = UDPService.create('udp-svc1', 5000)
udp2 = UDPService.create('udp-svc2', 5001)
UDPServiceGroup.create('udpsvcgroup', element=[udp1, udp2])
```

Available attributes:

Variables **element** (*list*) – list of elements by href. Call *~obtain_members* to retrieved the resolved list of elements.

classmethod create(*name*, *members=None*, *comment=None*)
Create the UDP Service group

Parameters

- **name** (*str*) – name of service group
- **element** (*list*) – UDP services or service group by reference

Raises ***CreateElementFailed*** – element creation failed with reason

Returns instance with meta

Return type *UDPServiceGroup*

13.4.3.7 URLCategoryGroup

class smc.elements.group.**URLCategoryGroup**(*name*, ***meta*)
Bases: *smc.base.model.Element*

13.4.4 Servers

Module that represents server based configurations

class smc.elements.servers.**MultiContactAddress**(***meta*)
A MultiContactAddress is a location and contact address pair which can have multiple addresses. Server elements such as Management and Log Server can have configured locations with mutliple addresses per location.

Use this server reference to create, add or remove contact addresses from servers:

```
mgt_server = ManagementServer.objects.first()
mgt_server.contact_addresses.update_or_create(
    location='mylocation', addresses=['1.1.1.1', '1.1.1.2'])
```

Or remove by location:

```
mgt_server.contact_addresses.delete('mylocation')
```

delete(*location_name*)
Remove a given location by location name. This operation is performed only if the given location is valid, and if so, *update* is called automatically.

Parameters **location** (*str*) – location name or location ref

Raises ***UpdateElementFailed*** – failed to update element with reason

Return type bool

get(*location_name*)
Get a contact address by location name

Parameters **location_name** (*str*) – name of location

Returns return contact address element or None

Return type *ContactAddress*

update_or_create(*location*, *contact_addresses*, *with_status=False*, *overwrite_existing=False*, ***kw*)

Update or create a contact address and location pair. If the location does not exist it will be automatically created. If the server already has a location assigned with the same name, the contact address specified will be added if it doesn't already exist (Management and Log Server can have multiple address for a single location).

Parameters

- **contact_addresses** (*list (str)*) – list of contact addresses for the specified location
- **location** (*str*) – location to place the contact address in
- **overwrite_existing** (*bool*) – if you want to replace existing location to address mappings set this to True. Otherwise if the location exists, only new addresses are appended
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises ***UpdateElementFailed*** – failed to update element with reason

Return type *MultiContactAddress*

class smc.elements.servers.**ContactAddressMixin**

Mixin class to provide an interface to contact addresses on the management and log server. Contact addresses on servers can contain multiple IP's for a single location.

add_contact_address(*contact_address*, *location*)

Add a contact address to the Log Server:

```
server = LogServer('LogServer 172.18.1.25')
server.add_contact_address('44.44.44.4', 'ARmoteLocation')
```

Parameters

- **contact_address** (*str*) – IP address used as contact address
- **location** (*str*) – Name of location to use, will be created if it doesn't exist

Raises ***ModificationFailed*** – failed adding contact address

Returns None

contact_addresses

Provides a reference to contact addresses used by this server.

Obtain a reference to manipulate or iterate existing contact addresses:

```
>>> from smc.elements.servers import ManagementServer
>>> mgt_server = ManagementServer.objects.first()
>>> for contact_address in mgt_server.contact_addresses:
...   contact_address
...
ContactAddress(location=Default,addresses=[u'1.1.1.1'])
ContactAddress(location=foolocation,addresses=[u'12.12.12.12'])
```

Return type *MultiContactAddress*

remove_contact_address(*location*)
Remove contact address by name of location. You can obtain all contact addresses by calling *contact_addresses()*.

Parameters **location** (*str*) – str name of location, will be created if it doesn't exist

Raises ***ModificationFailed*** – failed removing contact address

Returns None

13.4.4.1 LogServer

class smc.elements.servers.**LogServer**(*name*, ***meta*)
Bases: *smc.elements.servers.ContactAddressMixin*, *smc.base.model.Element*

Log Server elements are used to receive log data from the security engines Most settings on Log Server generally do not need to be changed, however it may be useful to set a contact address location and IP mapping if the Log Server needs to be reachable from an engine across NAT

It's easiest to get the management server reference through a collection:

```
>>> LogServer.objects.first()
LogServer(name=LogServer 172.18.1.150)
```

13.4.4.2 ManagementServer

class smc.elements.servers.**ManagementServer**(*name*, ***meta*)
Bases: *smc.elements.servers.ContactAddressMixin*, *smc.base.model.Element*

Management Server configuration. Most configuration settings are better set through the SMC UI, such as HA, however this object can be used to do simple tasks such as add a contact addresses to the Management Server when a security engine needs to communicate over NAT.

It's easiest to get the management server reference through a collection:

```
>>> ManagementServer.objects.first()
ManagementServer(name=Management Server)
```

Variables

- ***name*** – name of management server
- **address** – address of Management Server

13.4.4.3 DNSServer

class smc.elements.servers.**DNSServer**(*name*, ***meta*)
There are some cases in which you must define an External DNS Server element.

- For dynamic DNS (DDNS) updates with a Multi-Link configuration.
- If you want to use a DNS server for resolving malware signature mirrors.
- If you want to use a DNS server for resolving domain names and URL filtering categorization services on Firewalls, IPS engines, and Layer 2 Firewalls.

You can also optionally use External DNS Server elements to specify the DNS servers to which the firewall forwards DNS requests when you configure DNS relay.

Variables

- **time_to_live** (*int*) – how long a DNS entry can be cached
- **update_interval** (*int*) – how often DNS entries can be updated

classmethod create(*name*, *address*, *time_to_live=20*, *update_interval=10*, *secondary=None*, *comment=None*)

Create a DNS Server element.

Parameters

- **name** (*str*) – Name of DNS Server
- **address** (*str*) – IP address for DNS Server element
- **time_to_live** (*int*) – Defines how long a DNS entry can be cached before querying the DNS server again (default: 20)
- **update_interval** (*int*) – Defines how often the DNS entries can be updated to the DNS server if the link status changes constantly (default: 10)
- **secondary** (*list*) – a secondary set of IP address for this element

Raises ***CreateElementFailed*** – Failed to create with reason

Return type *DNSServer*

13.4.4.4 HttpProxy

class smc.elements.servers.**HttpProxy**(*name*, ***meta*)

An HTTP Proxy based element. Used in various areas of the configuration such as engine properties to define proxies for File Reputation, etc.

classmethod create(*name*, *address*, *proxy_port=8080*, *username=None*, *password=None*, *secondary=None*, *comment=None*)

Create a new HTTP Proxy service. Proxy must define at least one primary address but can optionally also define a list of secondary addresses.

Parameters

- **name** (*str*) – Name of the proxy element
- **address** (*str*) – Primary address for proxy
- **proxy_port** (*int*) – proxy port (default: 8080)
- **username** (*str*) – optional username for authentication (default: None)
- **password** (*str*) – password for username if defined (default: None)
- **comment** (*str*) – optional comment
- **secondary** (*list*) – secondary list of proxy server addresses

Raises ***CreateElementFailed*** – Failed to create the proxy element

Return type *HttpProxy*

13.4.4.5 ProxyServer

class smc.elements.servers.**ProxyServer**(*name*, ***meta*)

Bases: *smc.elements.servers.ContactAddressMixin*, *smc.base.model.Element*

A ProxyServer element is used in the firewall policy to provide the ability to send HTTP, HTTPS, FTP or SMTP traffic to a next hop proxy. There are two types of next hop proxies, 'Generic' and 'Forcepoint AP Web".

Example of creating a configuration for a Forcepoint AP-Web proxy redirect:

```
server = ProxyServer.update_or_create(name='myproxy',
    address='1.1.1.1', proxy_service='forcepoint_ap-web_cloud',
    fp_proxy_key='mypassword', fp_proxy_key_id=3, fp_proxy_user_id=1234,
    inspected_service=[{'service_type': 'HTTP', 'port': '80'}])
```

Create a Generic Proxy forward service:

```
server = ProxyServer.update_or_create(name='generic', address='1.1.1.1,1.1.1.2',
    inspected_service=[{'service_type': 'HTTP', 'port': 80}, {'service_type':
↪'HTTPS', 'port': 8080}])
```

Inspected services take a list of keys *service_type* and *port*. Service type key values are 'HTTP', 'HTTPS', 'FTP' and 'SMTP'. Port value is the port for the respective protocol.

Parameters **http_proxy** (*str*) – type of proxy configuration, either generic or forcepoint_ap-web_cloud

classmethod create(*name*, *address*, *inspected_service*, *secondary=None*, *balancing_mode='ha'*, *proxy_service='generic'*, *location=None*, *comment=None*, *add_x_forwarded_for=False*, *trust_host_header=False*, ***kw*)

Create a Proxy Server element

Parameters

- **name** (*str*) – name of proxy server element
- **address** (*str*) – address of element. Can be a single FQDN or comma separated list of IP addresses
- **secondary** (*list*) – list of secondary IP addresses
- **balancing_mode** (*str*) – how to balance traffic, valid options are ha (first available server), src, dst, srcdst (default: ha)
- **proxy_service** (*str*) – which proxy service to use for next hop, options are generic or forcepoint_ap-web_cloud
- **location** (*str,Element*) – location for this proxy server
- **add_x_forwarded_for** (*bool*) – add X-Forwarded-For header when using the Generic Proxy forwarding method (default: False)
- **trust_host_header** (*bool*) – trust the host header when using the Generic Proxy forwarding method (default: False)
- **inspected_service** (*dict*) – inspection services dict. Valid keys are service_type and port. Service type valid values are HTTP, HTTPS, FTP or SMTP and are case sensitive
- **comment** (*str*) – optional comment
- **kw** – keyword arguments are used to collect settings when the proxy_service value is forcepoint_ap-web_cloud. Valid keys are *fp_proxy_key*, *fp_proxy_key_id*, *fp_proxy_user_id*. The fp_proxy_key is the password value. All other values are of type int

inspected_services

The specified services for inspection. An inspected service is a reference to a protocol that can be forwarded for inspection, such as HTTP, HTTPS, FTP and SMTP.

Return type list(InspectedService)

proxy_service
The proxy service for this proxy server configuration

Return type str

classmethod update_or_create(*with_status=False*, ***kwargs*)
Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to `get()`. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

13.4.5 Other

Other element types that treated more like generics, or that can be applied in different areas within the SMC. They will not independently be created as standalone objects and will be more generic container classes that define the required json when used by API functions or methods. For example, Blacklist can be applied to an engine directly or system wide. This class will define the format when calling blacklist functions.

class smc.elements.other.**Blacklist**
Blacklist provides a simple container to add multiple blacklist entries. Pass an instance of this to `smc.core.engine.blacklist_bulk` to upload to the engine.

add_entry(*src*, *dst*, *duration=3600*, *src_port1=None*, *src_port2=None*, *src_proto='predefined_tcp'*, *dst_port1=None*, *dst_port2=None*, *dst_proto='predefined_tcp'*)
Create a blacklist entry.

A blacklist can be added directly from the engine node, or from the system context. If submitting from the system context, it becomes a global blacklist. This will return the properly formatted json to submit.

Parameters

- **src** – source address, with cidr, i.e. 10.10.10.10/32 or 'any'
- **dst** – destination address with cidr, i.e. 1.1.1.1/32 or 'any'
- **duration** (*int*) – length of time to blacklist

Both the system and engine context blacklist allow kw to be passed to provide additional functionality such as adding source and destination ports or port ranges and specifying the protocol. The following parameters define the `kw` that can be passed.

The following example shows creating an engine context blacklist using additional kw:

```
engine.blacklist('1.1.1.1/32', '2.2.2.2/32', duration=3600,
    src_port1=1000, src_port2=1500, src_proto='predefined_udp',
    dst_port1=3, dst_port2=3000, dst_proto='predefined_udp')
```

Parameters

- **src_port1** (*int*) – start source port to limit blacklist
- **src_port2** (*int*) – end source port to limit blacklist
- **src_proto** (*str*) – source protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')
- **dst_port1** (*int*) – start dst port to limit blacklist
- **dst_port2** (*int*) – end dst port to limit blacklist
- **dst_proto** (*str*) – dst protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')

Note: if blocking a range of ports, use both src_port1 and src_port2, otherwise providing only src_port1 is adequate. The same applies to dst_port1 / dst_port2. In addition, if you provide src_portX but not dst_portX (or vice versa), the undefined port side definition will default to all ports.

---

class smc.elements.other.**Category**(*name*, ***meta*)

A Category is used by an element to group and categorize elements by some criteria. Once a category is created, it can be assigned to the element and used as a search filter when managing large numbers of elements. A category can be added to a category tag (or tags) to provide a higher level container/group for searching.

```
>>> from smc.elements.other import Category
>>> Category.create(name='footag', comment='test tag')
Category(name=footag)
```

Variables ***categories*** (*list (CategoryTag)*) – category tags for this category

add_category(*tags*)

Category Tags are used to characterize an element by a type identifier. They can then be searched and returned as a group of elements. If the category tag specified does not exist, it will be created. This change will take effect immediately.

Parameters **tags** (*list (str)*) – list of category tag names to add to this element

Raises ***ElementNotFound*** – Category tag element name not found

Returns None

See also:

smc.elements.other.Category

add_category_tag(*tags*, *append_lists=True*)
Add this category to a category tag (group). This provides drop down filters in the SMC UI by category tag.

Parameters

- **tags** (*list(str)*) – category tag by name
- **append_lists** (*bool*) – append to existing tags or overwrite default: append)

Returns None

add_element(*element*)
Element can be href or type *smc.base.model.Element*

```
>>> from smc.elements.other import Category
>>> category = Category('foo')
>>> category.add_element(Host('kali'))
```

Parameters **element** (*str,Element*) – element to add to tag

Raises ModificationFailed: failed adding element

Returns None

classmethod create(*name*, *comment=None*)
Add a category element

Parameters **name** – name of location

Returns instance with meta

Return type *Category*

remove_element(*element*)
Remove an element from this category tag. Find elements assigned by *search_elements()*. Element can be str href or type *smc.base.model.Element*.

```
>>> from smc.elements.other import Category
>>> from smc.elements.network import Host
>>> category.remove_element(Host('kali'))
```

Parameters **Element element** (*str,*) – element to remove

Raises ***ModificationFailed*** – cannot remove element

Returns None

search_elements()
Find all elements assigned to this category tag. You can also find category tags assigned directly to an element also:

```
>>> host = Host('kali')
>>> host.categories
[Category(name=myelements), Category(name=foocategory)]
```

Returns *smc.base.model.Element*

Return type list

class smc.elements.other.**CategoryTag**(*name*, ***meta*)
A Category Tag is a grouping of categories within SMC. Category Tags are used as filters (typically in the SMC UI) to change the view based on the tag.

Variables

- **child_categories** (*list(Category,CategoryTag)*) – child categories
- **parent_categories** (*list(Category,CategoryTag)*) – parent categories

classmethod create(*name*, *comment=None*)
Create a CategoryTag. A category tag represents a group of categories or a group of category tags (nested groups). These are used to provide filtering views within the SMC and organize elements by user defined criteria.

Parameters

- **name** (*str*) – name of category tag
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – problem creating tag

Returns instance with meta

Return type *CategoryTag*

remove_category(*categories*)
Remove a category from this Category Tag (group).

Parameters **categories** (*list(str,Element)*) – categories to remove

Returns None

class smc.elements.other.**ContactAddress**(*data=None*, ***kwargs*)
A contact address is used by elements to provide an alternative IP or FQDN mapping based on a location

addresses
List of addresses set as contact address

Return type list

name
Location name for this contact address

Return type str

class smc.elements.other.**FilterExpression**(*name*, ***meta*)
A filter expression defines either a system element filter or a user defined filter based on an expression. For example, a system level filter is 'Match All'. For classes that allow filters as input, a filter expression can be used.

class smc.elements.other.**HTTPSInspectionExceptions**(*name*, ***meta*)
The HTTPS Inspection Exceptions element is a list of domains that are excluded from decryption and inspection. HTTPS Inspection Exceptions are used in a custom HTTPS service to define a list of domains for which HTTPS traffic is not decrypted. The custom HTTPS service must be used in a rule, and only traffic that matches the rule is excluded from decryption and inspection.

Note: As of SMC 6.4.3, this is a read-only element

---

class smc.elements.other.**Location**(*name*, ***meta*)

Locations are used by elements to identify when they are behind a NAT connection. For example, if you have an engine that connects to the SMC across the internet using a public address, a location will be the tag applied to the Management Server (with contact address) and on the engine to identify how to connect. In this case, the location will map to a contact address using a public IP.

Note: Locations require SMC API version >= 6.1

classmethod create(*name*, *comment=None*)

Create a location element

Parameters **name** – name of location

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *Location*

used_on

Return all NAT'd elements using this location.

Note: Available only in SMC version 6.2

Returns elements used by this location

Return type list

class smc.elements.other.**LogicalInterface**(*name*, ***meta*)

Logical interface is used on either inline or capture interfaces. If an engine has both inline and capture interfaces (L2 Firewall or IPS role), then you must use a unique Logical Interface on the interface type.

Create a logical interface:

```
LogicalInterface.create('mylogical_interface')
```

classmethod create(*name*, *comment=None*)

Create the logical interface

Parameters

- **name** (*str*) – name of logical interface
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *LogicalInterface*

class smc.elements.other.**MacAddress**(*name*, ***meta*)

Mac Address network element that can be used in L2 and IPS policy source and destination fields.

Creating a MacAddress:

```
>>> MacAddress.create(name='mymac', mac_address='22:22:22:22:22:22')
MacAddress(name=mymac)
```

classmethod create(*name*, *mac_address*, *comment=None*)
Create the Mac Address

Parameters

- **name** (*str*) – name of mac address
- **mac_address** (*str*) – mac address notation
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *MacAddress*

class smc.elements.other.**SituationTag**(*name*, ***meta*)
A situation tag is used to categorize situations based on some sort of user defined criteria such as Botnet, Attacks, etc. These can help with categorization of specific threat event types.

smc.elements.other.**prepare_blacklist**(*src*, *dst*, *duration=3600*, *src_port1=None*, *src_port2=None*, *src_proto='predefined_tcp'*, *dst_port1=None*, *dst_port2=None*, *dst_proto='predefined_tcp'*)
Create a blacklist entry.

A blacklist can be added directly from the engine node, or from the system context. If submitting from the system context, it becomes a global blacklist. This will return the properly formatted json to submit.

Parameters

- **src** – source address, with cidr, i.e. 10.10.10.10/32 or 'any'
- **dst** – destination address with cidr, i.e. 1.1.1.1/32 or 'any'
- **duration** (*int*) – length of time to blacklist

Both the system and engine context blacklist allow kw to be passed to provide additional functionality such as adding source and destination ports or port ranges and specifying the protocol. The following parameters define the `kw` that can be passed.

The following example shows creating an engine context blacklist using additional kw:

```
engine.blacklist('1.1.1.1/32', '2.2.2.2/32', duration=3600,
    src_port1=1000, src_port2=1500, src_proto='predefined_udp',
    dst_port1=3, dst_port2=3000, dst_proto='predefined_udp')
```

Parameters

- **src_port1** (*int*) – start source port to limit blacklist
- **src_port2** (*int*) – end source port to limit blacklist
- **src_proto** (*str*) – source protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')
- **dst_port1** (*int*) – start dst port to limit blacklist
- **dst_port2** (*int*) – end dst port to limit blacklist
- **dst_proto** (*str*) – dst protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')

Note: if blocking a range of ports, use both src_port1 and src_port2, otherwise providing only src_port1 is adequate. The same applies to dst_port1 / dst_port2. In addition, if you provide src_portX but not dst_portX (or vice versa), the undefined port side definition will default to all ports.

13.4.5.1 Blacklist

class smc.elements.other.**Blacklist**

Blacklist provides a simple container to add multiple blacklist entries. Pass an instance of this to `smc.core.engine.blacklist_bulk` to upload to the engine.

add_entry(*src*, *dst*, *duration=3600*, *src_port1=None*, *src_port2=None*, *src_proto='predefined_tcp'*, *dst_port1=None*, *dst_port2=None*, *dst_proto='predefined_tcp'*)

Create a blacklist entry.

A blacklist can be added directly from the engine node, or from the system context. If submitting from the system context, it becomes a global blacklist. This will return the properly formatted json to submit.

Parameters

- **src** – source address, with cidr, i.e. 10.10.10.10/32 or 'any'
- **dst** – destination address with cidr, i.e. 1.1.1.1/32 or 'any'
- **duration** (*int*) – length of time to blacklist

Both the system and engine context blacklist allow kw to be passed to provide additional functionality such as adding source and destination ports or port ranges and specifying the protocol. The following parameters define the `kw` that can be passed.

The following example shows creating an engine context blacklist using additional kw:

```
engine.blacklist('1.1.1.1/32', '2.2.2.2/32', duration=3600,
    src_port1=1000, src_port2=1500, src_proto='predefined_udp',
    dst_port1=3, dst_port2=3000, dst_proto='predefined_udp')
```

Parameters

- **src_port1** (*int*) – start source port to limit blacklist
- **src_port2** (*int*) – end source port to limit blacklist
- **src_proto** (*str*) – source protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')
- **dst_port1** (*int*) – start dst port to limit blacklist
- **dst_port2** (*int*) – end dst port to limit blacklist
- **dst_proto** (*str*) – dst protocol. Either 'predefined_tcp' or 'predefined_udp'. (default: 'predefined_tcp')

Note: if blocking a range of ports, use both src_port1 and src_port2, otherwise providing only src_port1 is adequate. The same applies to dst_port1 / dst_port2. In addition, if you provide src_portX but not dst_portX (or vice versa), the undefined port side definition will default to all ports.

13.4.5.2 Category

class smc.elements.other.**Category**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A Category is used by an element to group and categorize elements by some criteria. Once a category is created, it can be assigned to the element and used as a search filter when managing large numbers of elements. A category can be added to a category tag (or tags) to provide a higher level container/group for searching.

```
>>> from smc.elements.other import Category
>>> Category.create(name='footag', comment='test tag')
Category(name=footag)
```

Variables ***categories*** (*list (CategoryTag)*) – category tags for this category

add_category(*tags*)
Category Tags are used to characterize an element by a type identifier. They can then be searched and returned as a group of elements. If the category tag specified does not exist, it will be created. This change will take effect immediately.

Parameters **tags** (*list (str)*) – list of category tag names to add to this element

Raises ***ElementNotFound*** – Category tag element name not found

Returns None

See also:

smc.elements.other.Category

add_category_tag(*tags*, *append_lists=True*)
Add this category to a category tag (group). This provides drop down filters in the SMC UI by category tag.

Parameters

- **tags** (*list (str)*) – category tag by name
- **append_lists** (*bool*) – append to existing tags or overwrite default: append)

Returns None

add_element(*element*)
Element can be href or type *smc.base.model.Element*

```
>>> from smc.elements.other import Category
>>> category = Category('foo')
>>> category.add_element(Host('kali'))
```

Parameters **element** (*str,Element*) – element to add to tag

Raises ModificationFailed: failed adding element

Returns None

classmethod create(*name*, *comment=None*)
Add a category element

Parameters **name** – name of location

Returns instance with meta

Return type *Category*

remove_element (*element*)
Remove an element from this category tag. Find elements assigned by *search_elements()*. Element can be str href or type *smc.base.model.Element*.

```
>>> from smc.elements.other import Category
>>> from smc.elements.network import Host
>>> category.remove_element(Host('kali'))
```

Parameters **Element element** (*str,*) – element to remove

Raises ***ModificationFailed*** – cannot remove element

Returns None

search_elements ()
Find all elements assigned to this category tag. You can also find category tags assigned directly to an element also:

```
>>> host = Host('kali')
>>> host.categories
[Category(name=myelements), Category(name=foocategory)]
```

Returns *smc.base.model.Element*

Return type list

13.4.5.3 CategoryTag

class smc.elements.other.**CategoryTag**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A Category Tag is a grouping of categories within SMC. Category Tags are used as filters (typically in the SMC UI) to change the view based on the tag.

Variables

- **child_categories** (*list(Category,CategoryTag)*) – child categories
- **parent_categories** (*list(Category,CategoryTag)*) – parent categories

classmethod create (*name*, *comment=None*)
Create a CategoryTag. A category tag represents a group of categories or a group of category tags (nested groups). These are used to provide filtering views within the SMC and organize elements by user defined criteria.

Parameters

- **name** (*str*) – name of category tag
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – problem creating tag

Returns instance with meta

Return type *CategoryTag*

remove_category (*categories*)
Remove a category from this Category Tag (group).

Parameters **categories** (*list(str,Element)*) – categories to remove

Returns None

13.4.5.4 FilterExpression

class smc.elements.other.**FilterExpression**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A filter expression defines either a system element filter or a user defined filter based on an expression. For example, a system level filter is 'Match All'. For classes that allow filters as input, a filter expression can be used.

13.4.5.5 Location

class smc.elements.other.**Location**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Locations are used by elements to identify when they are behind a NAT connection. For example, if you have an engine that connects to the SMC across the internet using a public address, a location will be the tag applied to the Management Server (with contact address) and on the engine to identify how to connect. In this case, the location will map to a contact address using a public IP.

Note: Locations require SMC API version >= 6.1

---

classmethod **create**(*name*, *comment=None*)
Create a location element

Parameters **name** – name of location

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *Location*

used_on
Return all NAT'd elements using this location.

Note: Available only in SMC version 6.2

Returns elements used by this location

Return type list

13.4.5.6 LogicalInterface

class smc.elements.other.**LogicalInterface**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Logical interface is used on either inline or capture interfaces. If an engine has both inline and capture interfaces (L2 Firewall or IPS role), then you must use a unique Logical Interface on the interface type.

Create a logical interface:

```
LogicalInterface.create('mylogical_interface')
```

classmethod create(*name*, *comment=None*)
Create the logical interface

Parameters

- **name** (*str*) – name of logical interface
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *LogicalInterface*

13.4.5.7 MacAddress

class smc.elements.other.**MacAddress**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Mac Address network element that can be used in L2 and IPS policy source and destination fields.

Creating a MacAddress:

```
>>> MacAddress.create(name='mymac', mac_address='22:22:22:22:22:22')
MacAddress(name=mymac)
```

classmethod create(*name*, *mac_address*, *comment=None*)
Create the Mac Address

Parameters

- **name** (*str*) – name of mac address
- **mac_address** (*str*) – mac address notation
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating element with reason

Returns instance with meta

Return type *MacAddress*

13.4.5.8 HTTPSInspectionExceptions

class smc.elements.other.**HTTPSInspectionExceptions**(*name*, ***meta*)
Bases: *smc.base.model.Element*

The HTTPS Inspection Exceptions element is a list of domains that are excluded from decryption and inspection. HTTPS Inspection Exceptions are used in a custom HTTPS service to define a list of domains for which HTTPS traffic is not decrypted. The custom HTTPS service must be used in a rule, and only traffic that matches the rule is excluded from decryption and inspection.

Note: As of SMC 6.4.3, this is a read-only element

13.4.6 Situations

Module that represents inspection and correlated situations.

New in version 0.6.3: Requires SMC version >= 6.5

Situations can be either inspection related or correlated. Both types can be searched to obtain collections.

Every situation has an associated 'context' which identifies properties of the situation and how matching or correlation is performed.

A situation context group is a top level structure that encapsulates similar individual inspection contexts. You can retrieve these as follows:

```
>>> from smc.elements.situations import SituationContextGroup
>>> for group in SituationContextGroup.objects.all():
...   group
...
SituationContextGroup(name=DoS Detection)
SituationContextGroup(name=FINGER)
SituationContextGroup(name=SMTP Deprecated)
SituationContextGroup(name=PPTP)
SituationContextGroup(name=IPv6)
SituationContextGroup(name=NETBIOS)
SituationContextGroup(name=SIP)
SituationContextGroup(name=SNMP)
```

You can optionally retrieve situation context groups directly, and iterate the inspection contexts (sub_elements), which might be additional situation context groups or inspection contexts:

```
>>> group = SituationContextGroup('DoS Detection')
>>> group.sub_elements
[InspectionSituationContext(name=TCP synflood detection (SYN-ACK timeout based␣
↪detection)),
 InspectionSituationContext(name=TCP synflood detection (SYN-timeout method)),
 InspectionSituationContext(name=Non-ratebased DoS attacks),
 InspectionSituationContext(name=TCP DoS events),
 InspectionSituationContext(name=UDP DoS events), InspectionSituationContext(name=UDP␣
↪DoS detected)]
```

If you are interested in inspection contexts directly (i.e. groups are 'flattened' out), you can retrieve these as follows:

```
>>> from smc.elements.situations import InspectionSituationContext
>>> for context in InspectionSituationContext.objects.all():
...   context
...
InspectionSituationContext(name=Context for DNS_POLICY_NOTIFY_FAIL)
InspectionSituationContext(name=Context for FTP AUTH success)
InspectionSituationContext(name=TCP PPTP Server Stream)
InspectionSituationContext(name=Context for SMTP_INCONSISTENT_REPLIES)
InspectionSituationContext(name=Context for TCP Option Too Short)
InspectionSituationContext(name=RIFF File Stream)
InspectionSituationContext(name=Context for IP Total Length Error)
...
```

You can optionally retrieve an inspection situation context directly. Most situation contexts are system level elements and will be read only, but you can fetch them to view configurations if necessary.

Every situation context will have at least one *situation parameter*, which is the parameter / value pair used to match the on inspection situations which are categorized by the situation context. For example, in the case of detecting a text

file stream, a single regular expression type situation parameter is used:

```
>>> context = InspectionSituationContext('Text File Stream')
>>> for parameter in context.situation_parameters:
...   parameter
...
SituationParameter(name=Regular Expression)
```

Inspection Situations are the individual events that are either predefined or system defined that identify specific events to inspect for. All inspection situations have an inspection context (see above), and can also be customized or be duplicated.

Creating an inspection situation is a two step process. You must first create the situation with a specified context, then add the necessary parameter values.

An example of creating a new situation that uses a regular expression pattern to match within a Text File Stream:

```
>>> from smc.elements.situations import InspectionSituation
>>> from smc.elements.situations import InspectionSituationContext
>>>
>>> situation = InspectionSituation.create(name='foosituation', situation_
↪context=InspectionSituationContext('Text File Stream'), severity='high')
>>> situation
InspectionSituation(name=foosituation)
>>> situation.create_regular_expression(r'(?x)\n.*ActiveXObject \x28 \x22 WScript\.
↪Shell(?[s_file_text_script -> sid()])\n')
>>>
```

class smc.elements.situations.**CorrelationSituation**(*name*, ***meta*)
Bases: *smc.elements.situations.Situation*

Correlation Situations are used by NGFW Engines and Log Servers to conduct further analysis of detected events. Correlation Situations do not handle traffic directly. Instead they analyze the events generated by matches to Situations found in traffic. Correlation Situations use Event Binding elements to define the log events that bind together different types of events in traffic.

class smc.elements.situations.**CorrelationSituationContext**(*name*, ***meta*)
Bases: *smc.elements.situations.SituationContext*

Correlation Contexts define the patterns for matching groups of related events in traffic. Examples of correlation contexts are Count, Compress, Group, Match and Sequence. See SMC documentation for more details on each context type and meaning.

class smc.elements.situations.**InspectionSituation**(*name*, ***meta*)
Bases: *smc.elements.situations.Situation*

It is an element that identifies and describes detected events in the traffic or in the operation of the system. Situations contain the Context information, i.e., a pattern that the system is to look for in the inspected traffic.

classmethod create(*name*, *situation_context*, *attacker=None*, *target=None*, *severity='information'*, *situation_type=None*, *description=None*, *comment=None*)
Create an inspection situation.

Parameters

- **name** (*str*) – name of the situation
- **situation_context** (InspectionSituationContext) – The situation context type used to define this situation. Identifies the proper parameter that identifies how the situation is defined (i.e. regex, etc).

- **attacker** (*str*) – Attacker information, used to identify last packet the triggers attack and is only used for blacklisting. Values can be packet_source, packet_destination, connection_source, or connection_destination
- **target** (*str*) – Target information, used to identify the last packet that triggers the attack and is only used for blacklisting. Values can be packet_source, packet_destination, connection_source, or connection_destination
- **severity** (*str*) – severity for this situation. Valid values are critical, high, low, information
- **description** (*str*) – optional description
- **comment** (*str*) – optional comment

create_regular_expression(*regexp*)
Create a regular expression for this inspection situation context. The inspection situation must be using an inspection context that supports regex.

Parameters **regexp** (*str*) – regular expression string

Raises ***CreateElementFailed*** – failed to modify the situation

vulnerability_references
If this inspection situation has associated CVE, OSVDB, BID, etc references, this will return those reference IDs

Return type list(str)

class smc.elements.situations.**InspectionSituationContext**(*name*, ***meta*)
Bases: *smc.elements.situations.SituationContext*

Represents groups of situation contexts that can be characterized by a common technique used for identifying the situation. Contexts also typically have in common the type of situation they apply to, i.e. *File Text Stream* would be an inspection context, and encapsulates inspection situations such as ActiveX in text file stream detection, etc.

class smc.elements.situations.**Situation**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Situation defines a common interface for inspection and correlated situations.

attacker
How the Attacker is determined when the Situation matches. This information is used for blacklisting and in log entries and may be None

Return type str or None

description
The description for this situation

Return type str

parameter_values
Parameter values for this inspection situation. This correlate to the the situation_context.

Return type list(*SituationParameterValue*)

severity
The severity of this inspection situation, critical, high, low, information

Return type int

target
How the Target is determined when the Situation matches. This information is used for blacklisting and in log entries and may be None

Return type str or None

class smc.elements.situations.**SituationContext**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A situation context can be used by an inspection situation or by a correlated situation. The context defines the situation parameters used to define a pattern match and how that match is made.

Variables

- ***name*** (*str*) – name of this situation context
- ***comment*** (*str*) – comment for the context

description
Description for this context

Return type str

situation_parameters
Situation parameters defining detection logic for the context. This will return a list of SituationParameter indicating how the detection is made, i.e. regular expression, integer value, etc.

Return type list(*SituationParameter*)

class smc.elements.situations.**SituationContextGroup**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A situation context group is simply a top level group for organizing individual situation contexts. This is a top level element that can be retrieved directly:

```
>>> from smc.elements.situations import SituationContextGroup
>>> for group in SituationContextGroup.objects.all():
...   group
...
SituationContextGroup(name=DoS Detection)
SituationContextGroup(name=FINGER)
SituationContextGroup(name=SMTP Deprecated)
SituationContextGroup(name=PPTP)
SituationContextGroup(name=IPv6)
SituationContextGroup(name=NETBIOS)
SituationContextGroup(name=SIP)
SituationContextGroup(name=SNMP)
...
```

Variables **InspectionContextGroup) sub_elements**(*list(InspectionContext,*) – the members of this inspection context group

class smc.elements.situations.**SituationParameter**(***meta*)
Bases: *smc.base.model.SubElement*

A situation parameter defines the parameter type used to define the inspection situation context. For example, Regular Expression would be a situation parameter.

display_name
The display name as shown in the SMC

Return type str

order
The order placement for this parameter. This is only relevant when there are multiple parameters in an inspection context definition.

Return type int

type
The type of this situation parameter in textual format. For example, integer, regexp, etc.

Return type str

class smc.elements.situations.**SituationParameterValue**(***meta*)
Bases: *smc.base.model.SubElement*

The situation parameter value is associated with a situation parameter and as the name implies, provides the value payload for the given parameter.

13.4.7 Profiles

Profiles are generic container settings that are used in other areas of the SMC configuration. Each profile should document it's usage and how it is referenced.

13.4.7.1 DNSRelayProfile

Profiles are templates used in other parts of the system to provide default functionality for specific feature sets. For example, to enable DNS Relay on an engine you must specify a DNSRelayProfile to use which defines the common settings (or sub-settings) for that feature.

A DNS Relay Profile allows multiple DNS related mappings that can be configured. Example usage:

```
>>> from smc.elements.profiles import DNSRelayProfile
>>> profile = DNSRelayProfile('mynewprofile')
```

Note: If the DNSRelayProfile does not exist, it will automatically be created when a DNS relay rule is added to the DNSRelayProfile instance.

Add a fixed domain answer rule:

```
>>> profile.fixed_domain_answer.add([('microsoft3.com', 'foo.com'), ('microsoft4.com',
↪)])
>>> profile.fixed_domain_answer.all()
[{u'domain_name': u'microsoft3.com', u'translated_domain_name': u'foo.com'}, {u
↪'domain_name': u'microsoft4.com'}]
```

Translate hostnames (not fqdn) to a specific IP address:

```
>>> profile.hostname_mapping.add([('hostname1,hostname2', '1.1.1.12')])
>>> profile.hostname_mapping.all()
[{u'hostnames': u'hostname1,hostname2', u'ipaddress': u'1.1.1.12'}]
```

Translate an IP address to another:

```
>>> profile.dns_answer_translation.add([('12.12.12.12', '172.18.1.20')])
>>> profile.dns_answer_translation.all()
[{u'translated_ipaddress': u'172.18.1.20', u'original_ipaddress': u'12.12.12.12'}]
```

Specify a DNS server to handle specific domains:

```
>>> profile.domain_specific_dns_server.add([('myfoo.com', '172.18.1.20')])
>>> profile.domain_specific_dns_server.all()
[{u'dns_server_addresses': u'172.18.1.20', u'domain_name': u'myfoo.com'}]
```

class smc.elements.profiles.**DNSRelayProfile**(*name*, ***meta*)
Bases: *smc.base.model.Element*

DNS Relay Settings specify a profile to handle how the engine will interpret DNS queries. Stonesoft can act as a DNS relay, rewrite DNS queries or redirect domains to the specified DNS servers.

dns_answer_translation
Add a DNS answer translation

Return type *DNSAnswerTranslation*

domain_specific_dns_server
Add domain to DNS server mapping

Return type *DomainSpecificDNSServer*

fixed_domain_answer
Add a fixed domain answer entry.

Return type *FixedDomainAnswer*

hostname_mapping
Add a hostname to IP mapping

Return type *HostnameMapping*

class smc.elements.profiles.**FixedDomainAnswer**(*profile*)
Bases: *smc.elements.profiles.DNSRule*

Direct requests for specific domains to IPv4 addresses, IPv6 addresses, fully qualified domain names (FQDNs), or empty DNS replies

add(*answers*)
Add a fixed domain answer. This should be a list of two-tuples, the first entry is the domain name, and the second is the translated domain value:

```
profile = DNSRelayProfile('dnsrules')
profile.fixed_domain_answer.add([
    ('microsoft.com', 'foo.com'), ('microsoft2.com',)])
```

Parameters **answers** (*tuple[str, str]*) – (domain_name, translated_domain_name)

Raises ***UpdateElementFailed*** – failure to add to SMC

Returns None

Note: translated_domain_name can be none, which will cause the NGFW to return NXDomain for the specified domain.

class smc.elements.profiles.**HostnameMapping**(*profile*)
Bases: *smc.elements.profiles.DNSRule*

Statically map host names, aliases for host names, and unqualified names (a host name without the domain suffix) to IPv4 or IPv6 addresses

add(*answers*)
Map specific hostname to specified IP address. Provide a list of two-tuples. The first entry is the hostname/s to translate (you can provide multiple comma separated values). The second entry should be the IP address to map the hostnames to:

```
profile = DNSRelayProfile('dnsrules')
profile.hostname_mapping.add([('hostname1,hostname2', '1.1.1.1')])
```

Parameters **answers** (*tuple[str, str]*) – (hostnames, ipaddress), hostnames can be a comma separated list.

Raises ***UpdateElementFailed*** – failure to add to SMC

Returns None

class smc.elements.profiles.**DomainSpecificDNSServer**(*profile*)
Bases: *smc.elements.profiles.DNSRule*

Forward DNS requests to different DNS servers based on the requested domain.

add(*answers*)
Relay specific domains to a specified DNS server. Provide a list of two-tuple with first entry the domain name to relay for. The second entry is the DNS server that should handle the query:

```
profile = DNSRelayProfile('dnsrules')
profile.domain_specific_dns_server.add([('myfoo.com', '172.18.1.20')])
```

Parameters **answers** (*tuple[str, str]*) – (domain_name, dns_server_addresses), dns server addresses can be a comma separated string

Raises ***UpdateElementFailed*** – failure to add to SMC

Returns None

class smc.elements.profiles.**DNSAnswerTranslation**(*profile*)
Bases: *smc.elements.profiles.DNSRule*

Map IPv4 addresses resolved by external DNS servers to IPv4 addresses in the internal network.

add(*answers*)
Takes an IPv4 address and translates to a specified IPv4 value. Provide a list of two-tuple with the first entry providing the original address and second entry specifying the translated address:

```
profile = DNSRelayProfile('dnsrules')
profile.dns_answer_translation.add([('12.12.12.12', '172.18.1.20')])
```

Parameters **answers** (*tuple[str, str]*) – (original_ipaddress, translated_ipaddress)

Raises ***UpdateElementFailed*** – failure to add to SMC

Returns None

class smc.elements.profiles.**DNSRule**(*profile*)
Bases: object

DNSRule is the parent class for all DNS relay rules.

all()
Return all entries

Return type list(dict)

13.4.7.2 SNMPAgent

class smc.elements.profiles.**SNMPAgent**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Minimal implementation of SNMPAgent

13.5 Engine

class smc.core.engine.**Engine**(*name*, ***meta*)
Bases: *smc.base.model.Element*

An engine is the top level representation of a firewall, IPS or virtualized software.

Engine can be referenced directly and will be loaded when attributes are accessed:

```
>>> from smc.core.engine import Engine
>>> engine = Engine('testfw')
>>> print(engine.href)
http://1.1.1.1:8082/6.1/elements/single_fw/39550
```

Generically search for engines of all types:

```
>>> list(Engine.objects.all())
[Layer3Firewall(name=i-06145fc6c59a04335 (us-east-2a)), FirewallCluster(name=sg_
↪vm),
Layer3VirtualEngine(name=ve-5), MasterEngine(name=master-eng)]
```

Or only search for specific engine types:

```
>>> from smc.core.engines import Layer3Firewall
>>> list(Layer3Firewall.objects.all())
[Layer3Firewall(name=i-06145fc6c59a04335 (us-east-2a))]
```

Engine types are defined in *smc.core.engines*.

add_interface(*interface*)
Add interface is a lower level option to adding interfaces directly to the engine. The interface is expected to be an instance of Layer3PhysicalInterface, Layer2PhysicalInterface, TunnelInterface, or ClusterInterface. The engines instance cache is flushed after this call is made to provide an updated cache after modification.

See also:

smc.core.engine.interface.update_or_create

Parameters **interface** (PhysicalInterface, TunnelInterface) – instance of pre-created interface

Returns None

add_route(*gateway*, *network*)
Add a route to engine. Specify gateway and network. If this is the default gateway, use a network address of 0.0.0.0/0.

Parameters

- **gateway** (*str*) – gateway of an existing interface
- **network** (*str*) – network address in cidr format

Raises ***EngineCommandFailed*** – invalid route, possibly no network

Returns None

adsl_interface
Get only adsl interfaces for this engine node.

Raises ***UnsupportedInterfaceType*** – adsl interfaces are only supported on layer 3 engines

Returns list of dict entries with href,name,type, or None

alias_resolving()
Alias definitions with resolved values as defined on this engine. Aliases can be used in rules to simplify multiple object creation

```
fw = Engine('myfirewall')
for alias in fw.alias_resolving():
    print(alias, alias.resolved_value)
...
(Alias(name=$$ Interface ID 0.ip), [u'10.10.0.1'])
(Alias(name=$$ Interface ID 0.net), [u'10.10.0.0/24'])
(Alias(name=$$ Interface ID 1.ip), [u'10.10.10.1'])
```

Returns generator of aliases

Return type *Alias*

antispoofing
Antispoofing interface information. By default is based on routing but can be modified.

```
for entry in engine.antispoofing.all():
    print(entry)
```

Returns top level antispoofing node

Return type *Antispoofing*

antivirus
AntiVirus engine settings. Note that for virtual engines the AV settings are configured on the Master Engine. Get current status:

```
engine.antivirus.status
```

Raises ***UnsupportedEngineFeature*** – Invalid engine type for AV

Return type *AntiVirus*

blacklist(*src*, *dst*, *duration=3600*, ***kw*)
Add blacklist entry to engine node by name. For blacklist to work, you must also create a rule with action "Apply Blacklist".

Parameters

- **src** – source address, with cidr, i.e. 10.10.10.10/32 or 'any'

- **dst** – destination address with cidr, i.e. 1.1.1.1/32 or 'any'
- **duration** (*int*) – how long to blacklist in seconds

Raises ***EngineCommandFailed*** – blacklist failed during apply

Returns None

Note: This method is only valid for SMC version < 6.4. Use *blacklist_bulk()* to add entries.

---

blacklist_bulk(*blacklist*)

Add blacklist entries to the engine node in bulk. For blacklist to work, you must also create a rule with action "Apply Blacklist". First create your blacklist entries using *smc.elements.other.Blacklist* then provide the blacklist to this method.

Parameters **Blacklist** (*blacklist*) – pre-configured blacklist entries

---

Note: This method requires SMC version >= 6.4

---

blacklist_flush()

Flush entire blacklist for engine

Raises ***EngineCommandFailed*** – flushing blacklist failed with reason

Returns None

blacklist_show(***kw*)

New in version 0.5.6: Requires pip install smc-python-monitoring

Blacklist show requires that you install the smc-python-monitoring package. To obtain blacklist entries from the engine you need to use this extension to plumb the websocket to the session. If you need more granular controls over the blacklist such as filtering by source and destination address, use the smc-python-monitoring package directly. Blacklist entries that are returned from this generator have a delete() method that can be called to simplify removing entries. A simple query would look like:

```
for bl_entry in engine.blacklist_show():
    print(bl_entry)
```

Parameters **kw** – keyword arguments passed to blacklist query. Common setting is to pass max_recv=20, which specifies how many "receive" batches will be retrieved from the SMC for the query. At most, 200 results can be returned in a single query. If max_recv=5, then 1000 results can be returned if they exist. If less than 1000 events are available, the call will be blocking until 5 receives has been reached.

Returns generator of results

Return type *smc_monitoring.monitors.blacklist.BlacklistEntry*

contact_addresses

Contact addresses are NAT addresses that are assigned to interfaces. These are used when a component needs to communicate with another component through a NAT'd connection. For example, if a firewall is known by a pubic address but the interface uses a private address, you would assign the public address as a contact address for that interface.

Note: Contact addresses are only supported with SMC >= 6.2.

Obtain all eligible interfaces for contact addressess:

```
>>> engine = Engine('dingo')
>>> for ca in engine.contact_addresses:
...   ca
...
ContactAddressNode(interface_id=11, interface_ip=10.10.10.20)
ContactAddressNode(interface_id=120, interface_ip=120.120.120.100)
ContactAddressNode(interface_id=0, interface_ip=1.1.1.1)
ContactAddressNode(interface_id=12, interface_ip=3.3.3.3)
ContactAddressNode(interface_id=12, interface_ip=17.17.17.17)
```

See also:

smc.core.contact_address

This is set to a private method because the logic doesn't make sense with respects to how this is configured under the SMC.

Return type *ContactAddressCollection*(*ContactAddressNode*)

default_nat

Configure default nat on the engine. Default NAT provides automatic NAT without the requirement to add specific NAT rules. This is a more common configuration for outbound traffic. Inbound traffic will still require specific NAT rules for redirection.

Return type *DefaultNAT*

dns

Current DNS entries for the engine. Add and remove DNS entries. This resource is iterable and yields instances of `smc.core.addon.DNSEntry`. Example of adding entries:

```
>>> from smc.elements.servers import DNSServer
>>> server = DNSServer.create(name='mydnsserver', address='10.0.0.1')
>>> engine.dns.add(['8.8.8.8', server])
>>> engine.update()
'http://172.18.1.151:8082/6.4/elements/single_fw/948'
>>> list(engine.dns)
[DNSEntry(rank=0,value=8.8.8.8,ne_ref=None),
 DNSEntry(rank=1,value=None,ne_ref=DNSServer(name=mydnsserver))]
```

Return type *RankedDNSAddress*

dns_relay

Enable, disable or get status for the DNS Relay Service on this engine. You must still separately configure the *smc.elements.profiles.DNSRelayProfile* that the engine references.

Raises ***UnsupportedEngineFeature*** – unsupported feature on this engine type.

Return type *DNSRelay*

dynamic_routing

Dynamic Routing entry point. Access BGP, OSPF configurations

Raises ***UnsupportedEngineFeature*** – Only supported on layer 3 engines

Return type DynamicRouting

file_reputation

File reputation status on engine. Note that for virtual engines the AV settings are configured on the Master Engine. Get current status:

```
engine.file_reputation.status
```

Raises ***UnsupportedEngineFeature*** – Invalid engine type for file rep

Return type *FileReputation*

generate_snapshot (*filename='snapshot.zip'*)
Generate and retrieve a policy snapshot from the engine This is blocking as file is downloaded

Parameters **filename** (*str*) – name of file to save file to, including directory path

Raises ***EngineCommandFailed*** – snapshot failed, possibly invalid filename specified

Returns None

installed_policy
Return the name of the policy installed on this engine. If no policy, None will be returned.

Return type str or None

interface
Get all interfaces, including non-physical interfaces such as tunnel or capture interfaces. These are returned as Interface objects and can be used to load specific interfaces to modify, etc.

```
for interfaces in engine.interface:
    ......
```

Return type *InterfaceCollection*

See *smc.core.interfaces.Interface* for more info

interface_options
Interface options specify settings related to setting primary/ backup management, outgoing, and primary/backup heartbeat interfaces. For example, set primary management interface (this unsets it from the currently assigned interface):

```
engine.interface_options.set_primary_mgt(10)
```

Obtain the primary management interface:

```
print(engine.interface_options.primary_mgt)
```

Return type *InterfaceOptions*

internal_gateway
Engine level VPN gateway information. This is a link from the engine to VPN level settings like VPN Client, Enabling/disabling an interface, adding VPN sites, etc. Example of adding a new VPN site to the engine's site list with associated networks:

```
>>> network = Network.get_or_create(name='mynetwork', ipv4_network='1.1.1.0/24
↪')
Network(name=mynetwork)
>>> engine.internal_gateway.vpn_site.create(name='mynewsite', site_
↪element=[network])
VPNSite(name=mynewsite)
```

Raises ***UnsupportedEngineFeature*** – internal gateway is only supported on layer 3 engine types.

Returns this engines internal gateway

Return type *InternalGateway*

l2fw_settings

Layer 2 Firewall Settings make it possible for a layer 3 firewall to run specified interfaces in layer 2 mode. This requires that a layer 2 interface policy is assigned to the engine and that inline_l2fw interfaces are created.

Raises ***UnsupportedEngineFeature*** – requires layer 3 engine

Return type *Layer2Settings*

location

The location for this engine. May be None if no specific location has been assigned.

Parameters **value** – location to assign engine. Can be name, str href, or Location element. If name, it will be automatically created if a Location with the same name doesn't exist.

Raises ***UpdateElementFailed*** – failure to update element

Returns Location element or None

log_server

Log server for this engine.

Returns The specified log server

Return type *LogServer*

loopback_interface

Retrieve any loopback interfaces for this engine. Loopback interfaces are only supported on layer 3 firewall types.

Retrieve all loopback addresses:

```
for loopback in engine.loopback_interface:
    print(loopback)
```

Raises ***UnsupportedInterfaceType*** – supported on layer 3 engine only

Return type *LoopbackCollection*

modem_interface

Get only modem interfaces for this engine node.

Raises UnsupportedInterfaceType: modem interfaces are only supported on layer 3 engines

Returns list of dict entries with href,name,type, or None

nodes

Return a list of child nodes of this engine. This can be used to iterate to obtain access to node level operations

```
>>> print(list(engine.nodes))
[Node(name=myfirewall node 1)]
>>> engine.nodes.get(0)
Node(name=myfirewall node 1)
```

Returns nodes for this engine

Return type *SubElementCollection*(*Node*)

pending_changes

Pending changes provides insight into changes on an engine that are pending approval or disapproval. Feature requires SMC >= v6.2.

Raises ***UnsupportedEngineFeature*** – SMC version >= 6.2 is required to support pending changes

Return type *PendingChanges*

permissions

Retrieve the permissions for this engine instance.

```
>>> from smc.core.engine import Engine
>>> engine = Engine('myfirewall')
>>> for x in engine.permissions:
...   print(x)
...
AccessControlList(name=ALL Elements)
AccessControlList(name=ALL Firewalls)
```

Raises ***UnsupportedEngineFeature*** – requires SMC version >= 6.1

Returns access control list permissions

Return type list(*AccessControlList*)

physical_interface

Returns a PhysicalInterface. This property can be used to add physical interfaces to the engine. For example:

```
engine.physical_interface.add_inline_interface(....)
engine.physical_interface.add_layer3_interface(....)
```

Raises ***UnsupportedInterfaceType*** – engine doesn't support this type

Return type *PhysicalInterfaceCollection*

policy_route

Configure policy based routes on the engine.

```
engine.policy_route.create(
    source='172.18.2.0/24', destination='192.168.3.0/24',
    gateway_ip='172.18.2.1')
```

Return type *PolicyRoute*

refresh(*timeout=3*, *wait_for_finish=False*, ***kw*)

Refresh existing policy on specified device. This is an asynchronous call that will return a 'follower' link that can be queried to determine the status of the task.

```
poller = engine.refresh()
while not poller.done():
    poller.wait(5)
    print('Percentage complete {}%'.format(poller.task.progress))
```

Parameters **timeout** (*int*) – timeout between queries

Raises ***TaskRunFailed*** – refresh failed, possibly locked policy

Return type *TaskOperationPoller*

rename (*name*)
Rename the firewall engine, nodes, and internal gateway (VPN gw)

Returns None

routing
Find all routing nodes within engine:

```
for routing in engine.routing.all():
    for routes in routing:
        ...
```

Or just retrieve a routing configuration for a single interface:

```
interface = engine.routing.get(0)
```

Returns top level routing node

Return type *Routing*

routing_monitoring
Return route table for the engine, including gateway, networks and type of route (dynamic, static). Calling this can take a few seconds to retrieve routes from the engine.

Find all routes for engine resource:

```
>>> engine = Engine('sg_vm')
>>> for route in engine.routing_monitoring:
...   route
...
Route(route_network=u'0.0.0.0', route_netmask=0, route_gateway=u'10.0.0.1',␣
↪route_type=u'Static', dst_if=1, src_if=-1)
...
```

Raises ***EngineCommandFailed*** – routes cannot be retrieved

Returns list of route elements

Return type *SerializedIterable*(*Route*)

sandbox
Configure sandbox settings on the engine. Get current status:

```
engine.sandbox.status
```

Raises ***UnsupportedEngineFeature*** – not supported on virtual engine

Return type *Sandbox*

sidewinder_proxy
Configure Sidewinder Proxy settings on this engine. Sidewinder proxy is supported on layer 3 engines and require SMC and engine version >= 6.1. Get current status:

```
engine.sidewinder_proxy.status
```

Raises ***UnsupportedEngineFeature*** – requires layer 3 engine

Return type *SidewinderProxy*

snapshots
References to policy based snapshots for this engine, including the date the snapshot was made

Raises ***EngineCommandFailed*** – failure downloading, or IOError

Return type *SubElementCollection*(*Snapshot*)

snmp
SNMP engine settings. SNMP is supported on all engine types, however can be enabled only on NDI interfaces (interfaces that have assigned addresses).

Return type *SNMP*

switch_physical_interface
Get only switch physical interfaces for this engine node.

Raises ***UnsupportedInterfaceType*** – wireless interfaces are only supported on layer 3 engines

Returns list of dict entries with href,name,type, or None

tls_inspection
TLS Inspection settings manage certificates assigned to the engine for TLS server decryption (inbound) and TLS client decryption (outbound). In order to enable either, you must first assign certificates to the engine. Example of adding TLSServerCredentials to an engine:

```
>>> engine = Engine('myfirewall')
>>> tls = TLSServerCredential('server2.test.local')
>>> engine.tls_inspection.add_tls_credential([tls])
>>> engine.tls_inspection.server_credentials
[TLSServerCredential(name=server2.test.local)]
```

Return type *TLSInspection*

tunnel_interface
Get only tunnel interfaces for this engine node.

Raises ***UnsupportedInterfaceType*** – supported on layer 3 engine only

Return type *TunnelInterfaceCollection*

upload(*policy=None*, *timeout=5*, *wait_for_finish=False*, ***kw*)
Upload policy to engine. This is used when a new policy is required for an engine, or this is the first time a policy is pushed to an engine. If an engine already has a policy and the intent is to re-push, then use *refresh()* instead. The policy argument can use a wildcard * to specify in the event a full name is not known:

```
engine = Engine('myfw')
task = engine.upload('Amazon*')
for message in task.wait():
    print(message)
```

Parameters

- **policy** (*str*) – name of policy to upload to engine; if None, current policy
- **timeout** (*int*) – timeout between queries

Raises ***TaskRunFailed*** – upload failed with reason

Return type *TaskOperationPoller*

url_filtering

Configure URL Filtering settings on the engine. Get current status:

```
engine.url_filtering.status
```

Raises ***UnsupportedEngineFeature*** – not supported on virtual engines

Return type *UrlFiltering*

version

Version of this engine. Can be none if the engine has not been initialized yet.

Return type str or None

virtual_physical_interface

Master Engine virtual instance only

A virtual physical interface is for a master engine virtual instance. This interface type is just a subset of a normal physical interface but for virtual engines. This interface only sets Auth_Request and Outgoing on the interface.

To view all interfaces for a virtual engine:

```
for intf in engine.virtual_physical_interface:
    print(intf)
```

Raises ***UnsupportedInterfaceType*** – supported on virtual engines only

Return type *VirtualPhysicalInterfaceCollection*

virtual_resource

Available on a Master Engine only.

To get all virtual resources call:

```
engine.virtual_resource.all()
```

Raises ***UnsupportedEngineFeature*** – master engine only

Return type *CreateCollection*(*VirtualResource*)

vpn

VPN configuration for the engine.

Raises UnsupportedEngineFeature: VPN is only supported on layer 3 engines.

Return type *VPN*

vpn_endpoint

A VPN endpoint is an address assigned to a layer 3 interface that can be enabled to turn on VPN capabilities. As an interface may have multiple IP addresses assigned, the endpoints are returned based on the address. Endpoints are properties of the engines Internal Gateway.

Raises *UnsupportedEngineFeature* – only supported on layer 3 engines

Return type *SubElementCollection*(*InternalEndpoint*)

vpn_mappings
New in version 0.6.0: Requires SMC version >= 6.3.4

VPN policy mappings (by name) for this engine. This is a shortcut method to determine which VPN policies are used by the firewall.

Raises *UnsupportedEngineFeature* – requires a layer 3 firewall and SMC version >= 6.3.4.

Return type *VPNMappingCollection*(*VPNMapping*)

wireless_interface
Get only wireless interfaces for this engine node.

Raises *UnsupportedInterfaceType* – wireless interfaces are only supported on layer 3 engines

Returns list of dict entries with href,name,type, or None

class smc.core.engine.**VPN**(*engine*)
Bases: object

VPN is the top level interface to all engine based VPN settings. To enable IPSEC, SSL or SSL VPN on the engine, enable on the endpoint.

add_site(*name*, *site_elements=None*)
Add a VPN site with site elements to this engine. VPN sites identify the sites with protected networks to be included in the VPN. Add a network and new VPN site:

```
>>> net = Network.get_or_create(name='wireless', ipv4_network='192.168.5.0/24
↪')
>>> engine.vpn.add_site(name='wireless', site_elements=[net])
VPNSite(name=wireless)
>>> list(engine.vpn.sites)
[VPNSite(name=dingo - Primary Site), VPNSite(name=wireless)]
```

Parameters

- **name** (*str*) – name for VPN site
- **site_elements** (*list(str,Element)*) – network elements for VPN site

Raises

- *ElementNotFound* – if site element is not found
- *UpdateElementFailed* – failed to add vpn site

Return type *VPNSite*

Note: Update is immediate for this operation.

gateway_certificate
A Gateway Certificate is used by the engine for securing communications such as VPN. You can also check the expiration, view the signing CA and renew the certificate from this element.

Returns GatewayCertificate

Return type list

gateway_profile
Gateway Profile for this VPN. This is only a valid setting on layer 3 firewalls.

Return type *GatewayProfile*

gateway_settings
A gateway settings profile defines VPN specific settings related to timers such as negotiation retries (min, max) and mobike settings. Gateway settings are only present on layer 3 FW types.

Return type *GatewaySettings*

Note: This can return None on layer 3 firewalls if VPN is not enabled.

---

generate_certificate(*common_name*, *public_key_algorithm='rsa'*, *signature_algorithm='rsa_sha_512'*, *key_length=2048*, *signing_ca=None*)
Generate an internal gateway certificate used for VPN on this engine. Certificate request should be an instance of VPNCertificate.

Param str common_name: common name for certificate

Parameters

- **public_key_algorithm** (*str*) – public key type to use. Valid values rsa, dsa, ecdsa.
- **signature_algorithm** (*str*) – signature algorithm. Valid values dsa_sha_1, dsa_sha_224, dsa_sha_256, rsa_md5, rsa_sha_1, rsa_sha_256, rsa_sha_384, rsa_sha_512, ecdsa_sha_1, ecdsa_sha_256, ecdsa_sha_384, ecdsa_sha_512. (Default: rsa_sha_512)
- **key_length** (*int*) – length of key. Key length depends on the key type. For example, RSA keys can be 1024, 2048, 3072, 4096. See SMC documentation for more details.
- **signing_ca** (*str,VPNCertificateCA*) – by default will use the internal RSA CA

Raises ***CertificateError*** – error generating certificate

Returns GatewayCertificate

internal_endpoint
Internal endpoints to enable VPN for the engine.

Return type *SubElementCollection*(*InternalEndpoint*)

loopback_endpoint
Internal Loopback endpoints to enable VPN for the engine.

Return type *SubElementCollection*(*InternalEndpoint*)

rename(*name*)
Rename the internal gateway.

Parameters **name** (*str*) – new name for internal gateway

Returns None

sites
VPN sites configured for this engine. Using sub element methods simplify fetching sites of interest:

```
engine = Engine('sg_vm')
mysite = engine.vpn.sites.get_contains('inter')
print(mysite)
```

Return type *CreateCollection*(*VPNSite*)

vpn_client
VPN Client settings for this engine.

Alias for internal_gateway.

Return type *InternalGateway*

class smc.core.engine.**VPNMapping**
Bases: *smc.core.engine.VPNMapping*

A VPN Mapping represents Policy Based VPNs associated with this engine. This simplifies finding references where an engine is used within a VPN without iterating through existing VPNs to find the engine.

internal_gateway
Return the engines internal gateway as element

Return type *InternalGateway*

is_central_gateway
Is this engine a central gateway in the VPN policy

Return type bool

is_mobile_gateway
Is the engine specified as a mobile gateway in the Policy VPN configuration

Return type bool

is_satellite_gateway
Is this engine a satellite gateway in the VPN policy

Return type bool

vpn
The VPN policy for this engine mapping

Return type *PolicyVPN*

class smc.core.engine.**VPNMappingCollection**(*vpns*)
Bases: *smc.base.structs.BaseIterable*

13.5.1 AddOn

Engine feature add on functionality such as default NAT, Antivirus, File Reputation, etc. These are common settings that are located under the SMC AddOn or General properties.

Property features will have a common interface allowing you to *enable*, *disable* and check *status* from the engine reference. When property features are modified, they are done so against a local copy of the server intsance. To commit the change, you must call .update() on the engine instance.

For example, to view status of antivirus, given a specific engine:

```
engine.antivirus.status
```

Then enable or disable:

```
engine.antivirus.enable()
engine.antivirus.disable()
engine.update()
```

..note:: Engine property settings require that you call engine.update() after making / queuing your changes.

13.5.1.1 AntiVirus

class smc.core.addon.**AntiVirus**(*engine*)

Antivirus settings for the engine. In order to use AV, you must also have DNS server addresses configured on the engine.

Enable AV, use a proxy for updates and adjust update schedule:

```
engine.antivirus.enable()
engine.antivirus.update_frequency('daily')
engine.antivirus.update_day('tu')
engine.antivirus.log_level('transient')
engine.antivirus.http_proxy('10.0.0.1', proxy_port=8080, user='foo', password=
↪'password')
engine.update()
```

Variables

- **antivirus_enabled** (*bool*) – is antivirus enabled
- **antivirus_http_proxy** (*str*) – http proxy settings
- **antivirus_http_proxy_enabled** (*bool*) – is http proxy enabled
- **antivirus_proxy_port** (*int*) – http proxy port
- **antivirus_proxy_user** (*str*) – http proxy user
- **antivirus_update** (*str*) – how often to update
- **antivirus_update_day** (*str*) – if update set to weekly, which day to update
- **antivirus_update_time** (*int*) – time to update av signatures
- **virus_log_level** (*str*) – antivirus logging level

Note: You must call engine.update() to commit any changes.

---

disable()

Disable antivirus on the engine

enable()

Enable antivirus on the engine

http_proxy(*proxy*, *proxy_port*, *user=None*, *password=None*)

New in version 0.5.7: Requires SMC and engine version >= 6.4

Set http proxy settings for Antivirus updates.

Parameters

- **proxy** (*str*) – proxy IP address
- **proxy_port** (*str, int*) – proxy port
- **user** (*str*) – optional user for authentication

log_level(*level*)

Set the log level for antivirus alerting.

Parameters **log_level** (*str*) – none,transient,stored,essential,alert

status
Status of AV on this engine

Return type bool

update_day(*day*)
Update the day when updates should occur.

Parameters **day** (*str*) – only used if 'weekly' is specified. Which day or week to perform update. Valid options: mo, tu, we, th, fr, sa, su.

update_frequency(*when*)
Set the update frequency. By default this is daily.

Parameters **antivirus_update** (*str*) – how often to check for updates. Valid options are: 'never','1hour', 'startup', 'daily', 'weekly'

13.5.1.2 FileReputation

class smc.core.addon.**FileReputation**(*engine*)
Configure the engine to use File Reputation capabilities.

Enable file reputation and specify outbound http proxies for queries:

```
engine.file_reputation.enable(http_proxy=[HttpProxy('myproxy')])
engine.update()
```

Variables **file_reputation_context** (*str*) – file reputation context, either gti_cloud_only or disabled

Note: You must call engine.update() to commit any changes.

disable()
Disable any file reputation on the engine.

enable(*http_proxy=None*)
Enable GTI reputation on the engine. If proxy servers are needed, provide a list of proxy elements.

Parameters **http_proxy** (*list(str,HttpProxy)*) – list of proxies for GTI connections

http_proxy
Return any HTTP Proxies that are configured for File Reputation.

Returns list of http proxy instances

Return type list(*HttpProxy*)

status
Return the status of File Reputation on this engine.

Return type bool

13.5.1.3 SidewinderProxy

class smc.core.addon.**SidewinderProxy**(*engine*)

Sidewinder status on this engine. Sidewinder proxy can only be enabled on specific engine types and also requires SMC and engine version >= 6.1.

Enable Sidewinder proxy:

```
engine.sidewinder_proxy.enable()
```

Note: You must call engine.update() to commit any changes.

disable()

Disable Sidewinder proxy on the engine

enable()

Enable Sidewinder proxy on the engine

status

Status of Sidewinder proxy on this engine

Return type bool

13.5.1.4 UrlFiltering

class smc.core.addon.**UrlFiltering**(*engine*)

Enable URL Filtering on the engine.

Enable Url Filtering with next hop proxies:

```
engine.url_filtering.enable(http_proxy=[HttpProxy('myproxy')])
engine.update()
```

Disable Url Filtering:

```
engine.url_filtering.disable()
engine.update()
```

Note: You must call engine.update() to commit any changes.

disable()

Disable URL Filtering on the engine

enable(*http_proxy=None*)

Enable URL Filtering on the engine. If proxy servers are needed, provide a list of HTTPProxy elements.

Parameters **http_proxy** (*list(str,HttpProxy)*) – list of proxies for GTI connections

http_proxy

Return any HTTP Proxies that are configured for Url Filtering.

Returns list of http proxy instances

Return type list(*HttpProxy*)

status

Return the status of URL Filtering on the engine

Return type bool

13.5.1.5 Sandbox

class smc.core.addon.**Sandbox**(*engine*)

Engine based sandbox settings. Sandbox can be configured for local (on prem) or cloud based sandbox. To create file filtering policies that use sandbox, you must first enable it and provide license keys on the engine.

Enable cloud sandbox on the engine, specifying a proxy for outbound connections:

```
engine.sandbox.enable(
    license_key='123',
    license_token='456',
    http_proxy=[HttpProxy('myproxy')])
```

Note: You must call engine.update() to commit any changes.

disable()

Disable the sandbox on this engine.

enable(*license_key*, *license_token*, *sandbox_type='cloud_sandbox'*, *service='Automatic'*, *http_proxy=None*, *sandbox_data_center='Automatic'*)

Enable sandbox on this engine. Provide a valid license key and license token obtained from your engine licensing. Requires SMC version >= 6.3.

Note: Cloud sandbox is a feature that requires an engine license.

Parameters

- **license_key** (*str*) – license key for specific engine
- **license_token** (*str*) – license token for specific engine
- **sandbox_type** (*str*) – 'local_sandbox' or 'cloud_sandbox'
- **service** (*str*, *SandboxService*) – a sandbox service element from SMC. The service defines which location the engine is in and which data centers to use. The default is to use the 'US Data Centers' profile if undefined.
- **sandbox_data_center** (*str*, *SandboxDataCenter*) – sandbox data center to use if the service specified does not exist. Requires SMC >= 6.4.3

Returns None

http_proxy

Return any HTTP Proxies that are configured for Sandbox.

Returns list of http proxy instances

Return type list(*HttpProxy*)

status

Status of sandbox on this engine

Return type bool

13.5.1.6 TLSInspection

class smc.core.addon.**TLSInspection**(*engine*)

TLS Inspection settings control settings for doing inbound TLS decryption and outbound client TLS decryption. This provides an interface to manage TLSServerCredentials and TLSClientCredentials assigned to the engine.

Note: You must call engine.update() to commit any changes.

add_tls_credential(*credentials*)

Add a list of TLSServerCredential to this engine. TLSServerCredentials can be in element form or can also be the href for the element.

Parameters **credentials** (*list(str,TLSServerCredential)*) – list of pre-created TLSServerCredentials

Returns None

remove_tls_credential(*credentials*)

Remove a list of TLSServerCredentials on this engine.

Parameters **credentials** (*list(str,TLSServerCredential)*) – list of credentials to remove from the engine

Returns None

server_credentials

Return a list of assigned (if any) TLSServerCredentials assigned to this engine.

Return type list(*TLSServerCredential*)

13.5.2 Dynamic Routing

Represents classes responsible for configuring dynamic routing protocols

13.5.2.1 OSPF

For more information on creating OSPF elements and enabling on a layer 3 engine:

See also:

smc.routing.ospf

13.5.2.2 BGP

For more information on creating BGP elements and enabling on a layer 3 engine:

See also:

smc.routing.bgp

13.5.3 General

13.5.3.1 DefaultNAT

class smc.core.general.**DefaultNAT**(*engine*)

Default NAT on the engine is used to automatically create NAT configurations based on internal routing. This simplifies the need to create specific NAT rules, primarily for outbound traffic.

Note: You must call engine.update() to commit any changes.

disable()

Disable default NAT on this engine

enable()

Enable default NAT on this engine

status

Status of default nat on the engine.

Return type bool

13.5.3.2 RankedDNSAddress

class smc.core.general.**RankedDNSAddress**(*entries*)

A RankedDNSAddress represents a list of DNS entries used as a ranked list to provide an ordered way to perform DNS queries. DNS entries can be added as raw IP addresses, or as elements of type *smc.elements.network.Host* or *smc.elements.servers.DNSServer* (or combination of both). This is an iterable class yielding namedtuples of type *DNSEntry*.

Normal access is done through an engine reference:

```
>>> list(engine.dns)
[DNSEntry(rank=0,value=8.8.8.8,ne_ref=None),
 DNSEntry(rank=1,value=None,ne_ref=DNSServer(name=mydnsserver))]

>>> engine.dns.append(['8.8.8.8', '9.9.9.9'])
>>> engine.dns.prepend(['1.1.1.1'])
>>> engine.dns.remove(['8.8.8.8', DNSServer('mydnsserver')])
```

Note: You must call engine.update() to commit any changes.

append(*values*)

Add DNS entries to the engine at the end of the existing list (if any). A DNS entry can be either a raw IP Address, or an element of type *smc.elements.network.Host* or *smc.elements.servers.DNSServer*.

Parameters **values** (*list*) – list of IP addresses, Host and/or DNSServer elements.

Returns None

Note: If the DNS entry added already exists, it will not be added. It's not a valid configuration to enter the same DNS IP multiple times. This is also true if the element is assigned the same address as a raw IP address already defined.

prepend(*values*)
Prepend DNS entries to the engine at the beginning of the existing list (if any). A DNS entry can be either a raw IP Address, or an element of type *smc.elements.network.Host* or *smc.elements.servers.DNSServer*.

Parameters **values** (*list*) – list of IP addresses, Host and/or DNSServer elements.

Returns None

remove(*values*)
Remove DNS entries from this ranked DNS list. A DNS entry can be either a raw IP Address, or an element of type *smc.elements.network.Host* or *smc.elements.servers.DNSServer*.

Parameters **values** (*list*) – list of IP addresses, Host and/or DNSServer elements.

Returns None

class smc.core.general.**DNSEntry**
DNSEntry represents a single DNS entry within an engine DNSAddress list.

Variables

- ***value*** (*str*) – IP address value of this entry (None if type Element is used)
- **rank** (*int*) – order rank for the entry
- **ne_ref** (*str*) – network element href of entry. Use element property to resolve to type Element.
- **element** (Element) – If the DNS entry is an element type, this property will returned a resolved version of the ne_ref field.

13.5.3.3 DNS Relay

class smc.core.general.**DNSRelay**(*engine*)
DNS Relay allows the engine to provide DNS caching or specific host, IP and domain replies to clients. It can also be used to sinkhole specific DNS requests.

See also:

smc.elements.profiles.DNSRelayProfile

disable()
Disable DNS Relay on this engine

Returns None

enable(*interface_id*, *dns_relay_profile=None*)
Enable the DNS Relay service on this engine.

Parameters

- **interface_id** (*int*) – interface id to enable relay
- **dns_relay_profile** (*str,DNSRelayProfile*) – DNSRelayProfile element or str href

Raises

- ***EngineCommandFailed*** – interface not found
- ***ElementNotFound*** – profile not found

Returns None

status
Status of DNS Relay on this engine.

Return type bool

13.5.3.4 SNMP

class smc.core.general.**SNMP**(*engine*)
SNMP configuration details for applying SNMP on an engine. SNMP requires at minimum an assigned SNMPAgent configuration which defines the SNMP specific settings (version, community string, etc). You can also define specific interfaces to enable SNMP on. By default, if no addresses are specified, SNMP will be defined on all interfaces.

See also:

smc.elements.profiles.SNMPAgent

agent
The SNMP agent profile used for this engine.

Return type *SNMPAgent*

disable()
Disable SNMP on this engine. You must call *update* on the engine for this to take effect.

Returns None

enable(*snmp_agent*, *snmp_location=None*, *snmp_interface=None*)
Enable SNMP on the engine. Specify a list of interfaces by ID to enable only on those interfaces. Only interfaces that have NDI's are supported.

Parameters

- **snmp_agent** (*str*,Element) – the SNMP agent reference for this engine
- **snmp_location** (*str*) – the SNMP location identifier for the engine
- **snmp_interface** (*list*) – list of interface IDs to enable SNMP

Raises

- ***ElementNotFound*** – unable to resolve snmp_agent
- ***InterfaceNotFound*** – specified interface by ID not found

interface
Return a list of physical interfaces that the SNMP agent is bound to.

Return type list(*PhysicalInterface*)

location
Return the SNMP location string

Return type str

update_configuration(**kwargs*)
Update the SNMP configuration using any kwargs supported in the *enable* constructor. Return whether a change was made. You must call update on the engine to commit any changes.

Parameters **kwargs** (*dict*) – keyword arguments supported by enable constructor

Return type bool

13.5.3.5 Layer2Settings

class smc.core.general.**Layer2Settings**(*engine*)

Layer 2 Settings are only applicable on Layer 3 Firewall engines that want to run specific interfaces in layer 2 mode. This requires that a Layer 2 Interface Policy is applied to the engine. You can also set connection tracking and bypass on overload settings for these interfaces as well.

Set policy for the engine:

```
engine.l2fw_settings.enable(InterfacePolicy('mylayer2'))
```

Variables

- **bypass_overload_traffic** (*bool*) – whether to bypass traffic on overload
- **tracking_mode** (*str*) – connection tracking mode

Note: You must call engine.update() to commit any changes.

Warning: This feature requires SMC and engine version >= 6.3

bypass_on_overload(*value*)

Set the l2fw settings to bypass on overload.

Parameters **value** (*bool*) – boolean to indicate bypass setting

Returns None

connection_tracking(*mode*)

Set the connection tracking mode for these layer 2 settings.

Parameters **mode** (*str*) – normal, strict, loose

Returns None

disable()

Disable the layer 2 interface policy

enable(*policy*)

Set a layer 2 interface policy.

Parameters **policy** (*str*, Element) – an InterfacePolicy or str href

Raises

- ***LoadPolicyFailed*** – Invalid policy specified
- ***ElementNotFound*** – InterfacePolicy not found

Returns None

policy

Return the InterfacePolicy for this layer 3 firewall.

Return type *InterfacePolicy*

13.5.4 VPN

Provisioning a firewall for VPN consists of the following steps:

- Enable VPN an interface (*InternalEndpoint*)
- Optionally add VPN sites with protected networks

Note: By default Stonesoft FW's provide a capability that allows the protected VPN networks to be identified based on the routing table.

It is still possible to override this setting and add your own custom VPN sites as needed.

Once the firewall has VPN enabled, you must also assign the FW to a specified Policy VPN as a central or satellite gateway.

The entry point for enabling the VPN on an engine is under the engine resource *smc.core.engine.Engine.vpn*.

Enabling IPSEC on an interface is done by accessing the engine resource and finding the correct *InternalEndpoint* for which to enable the VPN. Internal Endpoints are not exactly interface maps, instead they identify all addresses on a given firewall capable for running VPN. It is possible for a single interface to have more than one internal endpoint if the interface has multiple IP addresses assigned.

```
>>> from smc.core.engine import Engine
>>> engine = Engine('myfirewall')
>>> for ie in engine.vpn.internal_endpoint:
...   ie
...
InternalEndpoint(name=6.6.6.6)
InternalEndpoint(name=10.10.0.1)
InternalEndpoint(name=11.11.11.11)
InternalEndpoint(name=4.4.4.4)
InternalEndpoint(name=10.10.10.1)
```

Notice that internal endpoints are referenced by their IP address and not their interface. The interface is available as an attribute on the endpoint to make it easier to find the correct interface:

```
>>> for ie in engine.vpn.internal_endpoint:
...   print(ie, ie.interface_id)
...
(InternalEndpoint(name=6.6.6.6), u'6')
(InternalEndpoint(name=10.10.0.1), u'0')
(InternalEndpoint(name=11.11.11.11), u'11')
(InternalEndpoint(name=4.4.4.4), u'2.200')
(InternalEndpoint(name=10.10.10.1), u'1')
```

If I want to enable VPN on interface 0, you can obtain the right endpoint and enable:

```
>>> for ie in engine.vpn.internal_endpoint:
...   if ie.interface_id == '0':
...     ie.ipsec_vpn = True
```

Note: Once you've enabled the interface for VPN, you must also call engine.update() to commit the change.

The second step (optional) is to add VPN sites to the firewall. VPN Sites define a group of protected networks that can be applied to the VPN.

For example, add a new VPN site called wireless with a new network element that we'll create beforehand.

```
>>> net = Network.get_or_create(name='wireless', ipv4_network='192.168.5.0/24')
>>> engine.vpn.add_site(name='wireless', site_elements=[net])
VPNSite(name=wireless)
>>> list(engine.vpn.sites)
[VPNSite(name=dingo - Primary Site), VPNSite(name=wireless)]
```

Once the engine is enabled for VPN, see *smc.vpn.policy.PolicyVPN* for information on how to create a PolicyVPN and add engines.

13.5.4.1 InternalEndpoint

class smc.core.engine.**InternalEndpoint**(***meta*)

Bases: *smc.base.model.SubElement*

An Internal Endpoint is an interface mapping that enables VPN on the associated interface. This also defines what type of VPN to enable such as IPSEC, SSL VPN, or SSL VPN Portal.

To see all available internal endpoint (VPN gateways) on a particular engine, use an engine reference:

```
>>> engine = Engine('sg_vm')
>>> for e in engine.vpn.internal_endpoint:
...   print(e)
...
InternalEndpoint(name=10.0.0.254)
InternalEndpoint(name=172.18.1.254)
```

Available attributes:

Variables

- ***enabled*** (*bool*) – enable this interface as a VPN endpoint (default: False)
- **nat_t** (*bool*) – enable NAT-T (default: False)
- ***force_nat_t*** (*bool*) – force NAT-T (default: False)
- **ssl_vpn_portal** (*bool*) – enable SSL VPN portal on the interface (default: False)
- **ssl_vpn_tunnel** (*bool*) – enable SSL VPN tunnel on the interface (default: False)
- **ipsec_vpn** (*bool*) – enable IPSEC VPN on the interface (default: False)
- **udp_encapsulation** (*bool*) – Allow UDP encapsulation (default: False)
- **balancing_mode** (*str*) – VPN load balancing mode. Valid options are: 'standby', 'aggregate', 'active' (default: 'active')

interface_id

Interface ID for this VPN endpoint

Returns str interface id

physical_interface

Physical interface for this endpoint.

Return type *PhysicalInterface*

13.5.4.2 InternalGateway

class smc.core.engine.**InternalGateway**(***meta*)
Bases: *smc.base.model.SubElement*

InternalGateway represents the VPN Client configuration endpoint on the NGFW. Settings under Internal Gateway reflect client settings such as requiring antivirus, windows firewall and setting the VPN client mode.

View settings through an engine reference:

```
>>> engine = Engine('dingo')
>>> vpn = engine.vpn
>>> vpn.name
u'dingo Primary'
>>> vpn.vpn_client.firewall
False
>>> vpn.vpn_client.antivirus
False
>>> vpn.vpn_client.vpn_client_mode
u'ipsec'
```

Enable client AV and windows FW:

```
engine.vpn.vpn_client.update(
    firewall=True, antivirus=True)
```

Variables

- **firewall** (*bool*) – require windows firewall
- ***antivirus*** (*bool*) – require client antivirus
- **vpn_client_mode** (*str*) –

internal_endpoint
Internal endpoints to enable VPN for the engine.

Return type *SubElementCollection*(*InternalEndpoint*)

13.5.5 Interfaces

Represents classes responsible for configuring interfaces on engines

13.5.5.1 InterfaceCollections

Changed in version 0.7.0.

Collections classes for interfaces provide searching and methods to simplify creation based on interface types.

You can iterate any interface type by specifying the type:

```
>>> for interface in engine.tunnel_interface:
...   interface
...
TunnelInterface(name=Tunnel Interface 1008)
TunnelInterface(name=Tunnel Interface 1003)
TunnelInterface(name=Tunnel Interface 1000)
```

Or iterate all interfaces which will also return their types:

```
>>> for interface in engine.interface:
...   interface
...
Layer3PhysicalInterface(name=Interface 3)
TunnelInterface(name=Tunnel Interface 1000)
Layer3PhysicalInterface(name=Interface 61)
Layer3PhysicalInterface(name=Interface 56)
Layer3PhysicalInterface(name=Interface 15)
Layer2PhysicalInterface(name=Interface 7 (Capture))
ModemInterfaceDynamic(name=Modem 0)
TunnelInterface(name=Tunnel Interface 1030)
SwitchPhysicalInterfaceDynamic(name=Switch 0)
...
```

Accessing interface methods for creating interfaces can also be done in multiple ways. The simplest is to use an engine reference to use this collection. The engine reference specifies the type of interface and indicates how it will be created for the engine.

For example, creating an interface on a virtual engine:

```
engine.virtual_physical_interface.add_layer3_interface(
    interface_id=1,
    address='14.14.14.119',
    network_value='14.14.14.0/24',
    comment='my comment',
    zone_ref='myzone')
```

The helper methods use the interface API to create the interface that is then submitted to the engine. You can optionally create the interface manually using the API which provides more customization capabilities.

Example of creating a VirtualPhysicalInterface for a virtual engine manually:

```
payload = {'comment': 'comment on this interface',
           'interfaces': [{'nodes': [{'address': '13.13.13.13', 'network_value': '13.
↪13.13.0/24'}]}]}

vinterface = VirtualPhysicalInterface(interface_id=1, **payload)
```

Pass this to update_or_create in the event that you want to potentially modify an existing interface should the same interface ID exist:

```
engine.virtual_physical_interface.update_or_create(
    vinterface)
```

Or create a new interface (this will fail if the interface exists):

```
engine.add_interface(vinterface)
```

Collections also provide a simple helper when you want to provide a pre-configured interface and apply an update_or_create logic. In the update or create case, if the interface exists any fields that have changed will be updated. If the interface does not exist it is created. Provide *with_status* to obtain the interface and status of the operation. The update or create will return a tuple of (Interface, modified, created), where created and modified are booleans indicating the operations performed:

```
>>> from smc.core.engine import Engine
>>> from smc.core.interfaces import Layer3PhysicalInterface
```

(continues on next page)

(continued from previous page)

```
>>> engine = Engine('myfw')
>>> interface = engine.interface.get(0)
>>> interface
Layer3PhysicalInterface(name=Interface 0)
>>> interface.addresses
[(u'11.11.11.11', u'11.11.11.0/24', u'0')]
>>> myinterface = Layer3PhysicalInterface(interface_id=0,
interfaces=[{'nodes': [{'address': '66.66.66.66', 'network_value': '66.66.66.0/24'}]}
↪], comment='changed today')
...
>>> interface, modified, created = engine.physical_interface.update_or_
↪create(myinterface)
>>> interface
Layer3PhysicalInterface(name=Interface 0)
>>> modified
True
>>> created
False
>>> interface.addresses
[(u'66.66.66.66', u'66.66.66.0/24', u'0')]
>>> interface.comment
u'changed today'
```

class smc.core.collection.**InterfaceCollection**(*engine*, *rel='interfaces'*)

Bases: *smc.base.structs.BaseIterable*

An interface collection provides top level search capabilities to iterate or get interfaces of the specified type. This also delegates all 'add' methods of an interface to the interface type specified. Collections are returned from an engine reference and not called directly.

For example, you can use this to obtain all interfaces of a given type from an engine:

```
>>> for interface in engine.interface.all():
...   print(interface.name, interface.addresses)
('Tunnel Interface 2001', [('169.254.9.22', '169.254.9.20/30', '2001')])
('Tunnel Interface 2000', [('169.254.11.6', '169.254.11.4/30', '2000')])
('Interface 2', [('192.168.1.252', '192.168.1.0/24', '2')])
('Interface 1', [('10.0.0.254', '10.0.0.0/24', '1')])
('Interface 0', [('172.18.1.254', '172.18.1.0/24', '0')])
```

Or only physical interface types:

```
for interface in engine.physical_interfaces:
    print(interface)
```

Get a specific interface directly:

```
engine.interface.get(10)
```

Or use delegation to create interfaces:

```
engine.physical_interface.add(2)
engine.physical_interface.add_layer3_interface(....)
...
```

Note: This can raise UnsupportedInterfaceType for unsupported engine types based on the interface context.

get (*interface_id*)
Get the interface by id, if known. The interface is retrieved from the top level Physical or Tunnel Interface. If the interface is an inline interface, you can specify only one of the two inline pairs and the same interface will be returned.

If interface type is unknown, use engine.interface for retrieving:

```
>>> engine = Engine('sg_vm')
>>> intf = engine.interface.get(0)
>>> print(intf, intf.addresses)
(PhysicalInterface(name=Interface 0), [('172.18.1.254', '172.18.1.0/24', '0
↪')])
```

Get an inline interface:

```
>>> intf = engine.interface.get('2-3')
```

Note: For the inline interface example, you could also just specify '2' or '3' and the fetch will return the pair.

Parameters **interface_id** (*str*, *int*) – interface ID to retrieve

Raises ***InterfaceNotFound*** – invalid interface specified

Returns interface object by type (Physical, Tunnel, VlanInterface)

update_or_create (*interface*)
Collections class update or create method that can be used as a shortcut to updating or creating an interface. The interface must first be defined and provided as the argument. The interface method must have an *update_interface* method which resolves differences and adds as necessary.

Parameters **interface** (Interface) – an instance of an interface type, either PhysicalInterface or TunnelInterface

Raises

- ***EngineCommandFailed*** – Failed to create new interface
- ***UpdateElementFailed*** – Failure to update element with reason

Return type tuple

Returns A tuple with (Interface, modified, created), where created and modified are booleans indicating the operations performed

class smc.core.collection.**LoopbackCollection**(*engine*)
Bases: *smc.base.structs.BaseIterable*

An loopback collection provides top level search capabilities to iterate or get loopback interfaces from a given engine.

All loopback interfaces can be fetched from the engine:

```
>>> engine = Engine('dingo')
>>> for lb in engine.loopback_interface:
...   lb
...
LoopbackInterface(address=172.20.1.1, nodeid=1, rank=1)
LoopbackInterface(address=172.31.1.1, nodeid=1, rank=2)
```

Or directly from the nodes:

```
>>> for node in engine.nodes:
...   for lb in node.loopback_interface:
...     lb
...
LoopbackInterface(address=172.20.1.1, nodeid=1, rank=1)
LoopbackInterface(address=172.31.1.1, nodeid=1, rank=2)
```

get(*address*)
Get a loopback address by it's address. Find all loopback addresses by iterating at either the node level or the engine:

```
loopback = engine.loopback_interface.get('127.0.0.10')
```

Parameters **address** (*str*) – ip address of loopback

Raises ***InterfaceNotFound*** – invalid interface specified

Return type *LoopbackInterface*

class smc.core.collection.**PhysicalInterfaceCollection**(*engine*)
Bases: *smc.core.collection.InterfaceCollection*

PhysicalInterface Collection provides an interface to retrieving existing interfaces and helper methods to shortcut the creation of an interface.

add(*interface_id*, *virtual_mapping=None*, *virtual_resource_name=None*, *zone_ref=None*, *comment=None*)
Add single physical interface with interface_id. Use other methods to fully add an interface configuration based on engine type. Virtual mapping and resource are only used in Virtual Engines.

Parameters

- **interface_id** (*str*, *int*) – interface identifier
- **virtual_mapping** (*int*) – virtual firewall id mapping See *smc.core.engine.VirtualResource.vfw_id*
- **virtual_resource_name** (*str*) – virtual resource name See smc.core.engine.VirtualResource.name

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

add_capture_interface(*interface_id*, *logical_interface_ref*, *inspect_unspecified_vlans=True*, *zone_ref=None*, *comment=None*)
Add a capture interface. Capture interfaces are supported on Layer 2 FW and IPS engines.

..note:: Capture interface are supported on Layer 3 FW/clusters for NGFW engines version >= 6.3 and SMC >= 6.3.

Parameters

- **interface_id** (*str*, *int*) – interface identifier
- **logical_interface_ref** (*str*) – logical interface name, href or LogicalInterface. If None, 'default_eth' logical interface will be used.
- **zone_ref** (*str*) – zone reference, can be name, href or Zone

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

See *smc.core.sub_interfaces.CaptureInterface* for more information

add_cluster_interface_on_master_engine(*interface_id*, *macaddress*, *nodes*, *zone_ref=None*, *vlan_id=None*, *comment=None*)

Add a cluster address specific to a master engine. Master engine clusters will not use "CVI" interfaces like normal layer 3 FW clusters, instead each node has a unique address and share a common macaddress. Adding multiple addresses to an interface is not supported with this method.

Parameters

- **interface_id** (*str,int*) – interface id to use
- **macaddress** (*str*) – mac address to use on interface
- **nodes** (*list*) – interface node list
- **is_mgmt** (*bool*) – is this a management interface
- **zone_ref** – zone to use, by name, str href or Zone
- **vlan_id** – optional VLAN id if this should be a VLAN interface

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

add_dhcp_interface(*interface_id*, *dynamic_index*, *zone_ref=None*, *vlan_id=None*, *comment=None*)

Add a DHCP interface on a single FW

Parameters

- **interface_id** (*int*) – interface id
- **dynamic_index** (*int*) – index number for dhcp interface
- **primary_mgt** (*bool*) – whether to make this primary mgt
- **zone_ref** (*str*) – zone reference, can be name, href or Zone

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

See DHCPInterface for more information

add_inline_interface(*interface_id*, *second_interface_id*, *logical_interface_ref=None*, *vlan_id=None*, *second_vlan_id=None*, *zone_ref=None*, *second_zone_ref=None*, *failure_mode='normal'*, *comment=None*, ***kw*)

Add an inline interface pair. This method is only for IPS or L2FW engine types.

Parameters

- **interface_id** (*str*) – interface id of first interface
- **second_interface_id** (*str*) – second interface pair id
- **href logical_interface_ref** (*str,*) – logical interface by href or name
- **vlan_id** (*str*) – vlan ID for first interface in pair
- **second_vlan_id** (*str*) – vlan ID for second interface in pair
- **href zone_ref** (*str,*) – zone reference by name or href for first interface
- **href second_zone_ref** (*str,*) – zone reference by nae or href for second interface

- **failure_mode** (*str*) – normal or bypass
- **comment** (*str*) – optional comment

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

add_inline_ips_interface(*interface_id*, *second_interface_id*, *logical_interface_ref=None*, *vlan_id=None*, *failure_mode='normal'*, *zone_ref=None*, *second_zone_ref=None*, *comment=None*)

New in version 0.5.6: Using an inline interface on a layer 3 FW requires SMC and engine version >= 6.3.

An inline IPS interface is a new interface type for Layer 3 NGFW engines version >=6.3. Traffic passing an Inline IPS interface will have a access rule default action of Allow. Inline IPS interfaces are bypass capable. When using bypass interfaces and NGFW is powered off, in an offline state or overloaded, traffic is allowed through without inspection regardless of the access rules.

If the interface does not exist and a VLAN id is specified, the logical interface and zones will be applied to the top level physical interface. If adding VLANs to an existing inline ips pair, the logical and zones will be applied to the VLAN.

Parameters

- **interface_id** (*str*) – first interface in the interface pair
- **second_interface_id** (*str*) – second interface in the interface pair
- **logical_interface_ref** (*str*) – logical interface name, href or LogicalInterface. If None, 'default_eth' logical interface will be used.
- **vlan_id** (*str*) – optional VLAN id for first interface pair
- **failure_mode** (*str*) – 'normal' or 'bypass' (default: normal). Bypass mode requires fail open interfaces.
- **zone_ref** – zone for first interface in pair, can be name, str href or Zone
- **second_zone_ref** – zone for second interface in pair, can be name, str href or Zone
- **comment** (*str*) – comment for this interface

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

Note: Only a single VLAN is supported on this inline pair type

---

add_inline_l2fw_interface(*interface_id*, *second_interface_id*, *logical_interface_ref=None*, *vlan_id=None*, *zone_ref=None*, *second_zone_ref=None*, *comment=None*)

New in version 0.5.6: Requires NGFW engine >=6.3 and layer 3 FW or cluster

An inline L2 FW interface is a new interface type for Layer 3 NGFW engines version >=6.3. Traffic passing an Inline Layer 2 Firewall interface will have a default action in access rules of Discard. Layer 2 Firewall interfaces are not bypass capable, so when NGFW is powered off, in an offline state or overloaded, traffic is blocked on this interface.

If the interface does not exist and a VLAN id is specified, the logical interface and zones will be applied to the top level physical interface. If adding VLANs to an existing inline ips pair, the logical and zones will be applied to the VLAN.

Parameters

- **interface_id** (*str*) – interface id; '1-2', '3-4', etc
- **logical_interface_ref** (*str*) – logical interface name, href or LogicalInterface. If None, 'default_eth' logical interface will be used.
- **vlan_id** (*str*) – optional VLAN id for first interface pair
- **vlan_id2** (*str*) – optional VLAN id for second interface pair
- **zone_ref_intf1** – zone for first interface in pair, can be name, str href or Zone
- **zone_ref_intf2** – zone for second interface in pair, can be name, str href or Zone

Raises *EngineCommandFailed* – failure creating interface

Returns None

Note: Only a single VLAN is supported on this inline pair type

---

add_layer3_cluster_interface(*interface_id*, *cluster_virtual=None*, *network_value=None*, *macaddress=None*, *nodes=None*, *cvi_mode='packetdispatch'*, *zone_ref=None*, *comment=None*, ***kw*)

Add cluster virtual interface. A "CVI" interface is used as a VIP address for clustered engines. Providing 'nodes' will create the node specific interfaces. You can also add a cluster address with only a CVI, or only NDI's.

Add CVI only:

```
engine.physical_interface.add_cluster_virtual_interface(
    interface_id=30,
    cluster_virtual='30.30.30.1',
    network_value='30.30.30.0/24',
    macaddress='02:02:02:02:02:06')
```

Add NDI's only:

```
engine.physical_interface.add_cluster_virtual_interface(
    interface_id=30,
    nodes=nodes)
```

Add CVI and NDI's:

```
engine.physical_interface.add_cluster_virtual_interface(
    cluster_virtual='5.5.5.1',
    network_value='5.5.5.0/24',
    macaddress='02:03:03:03:03:03',
    nodes=[{'address':'5.5.5.2', 'network_value':'5.5.5.0/24', 'nodeid':1},
           {'address':'5.5.5.3', 'network_value':'5.5.5.0/24', 'nodeid':2}])
```

Changed in version 0.6.1: Renamed from add_cluster_virtual_interface

Parameters

- **interface_id** (*str,int*) – physical interface identifier
- **cluster_virtual** (*str*) – CVI address (VIP) for this interface
- **network_value** (*str*) – network value for VIP; format: 10.10.10.0/24
- **macaddress** (*str*) – mandatory mac address if cluster_virtual and cluster_mask provided

- **nodes** (*list*) – list of dictionary items identifying cluster nodes
- **cvi_mode** (*str*) – packetdispatch is recommended setting
- **zone_ref** (*str*) – zone reference, can be name, href or Zone
- **kw** – key word arguments are valid NodeInterface sub-interface settings passed in during create time. For example, 'backup_mgt=True' to enable this interface as the management backup.

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

add_layer3_interface(*interface_id*, *address*, *network_value*, *zone_ref=None*, *comment=None*, ***kw*)

Add a layer 3 interface on a non-clustered engine. For Layer 2 FW and IPS engines, this interface type represents a layer 3 routed (node dedicated) interface. For clusters, use the cluster related methods such as `add_cluster_virtual_interface()`

Parameters

- **interface_id** (*str,int*) – interface identifier
- **address** (*str*) – ip address
- **network_value** (*str*) – network/cidr (12.12.12.0/24)
- **zone_ref** (*str*) – zone reference, can be name, href or Zone
- **kw** – keyword arguments are passed to the sub-interface during create time. If the engine is a single FW, the sub-interface type is *smc.core.sub_interfaces.SingleNodeInterface*. For all other engines, the type is *smc.core.sub_interfaces.NodeInterface* For example, pass 'backup_mgt=True' to enable this interface as the management backup.

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

Note: If an existing ip address exists on the interface and zone_ref is provided, this value will overwrite any previous zone definition.

---

add_layer3_vlan_cluster_interface(*interface_id*, *vlan_id*, *nodes=None*, *cluster_virtual=None*, *network_value=None*, *macaddress=None*, *cvi_mode='packetdispatch'*, *zone_ref=None*, *comment=None*, ***kw*)

Add IP addresses to VLANs on a firewall cluster. The minimum params required are `interface_id` and `vlan_id`. To create a VLAN interface with a CVI, specify `cluster_virtual`, `cluster_mask` and `macaddress`.

To create a VLAN with only NDI, specify `nodes` parameter.

Nodes data structure is expected to be in this format:

```
nodes=[{'address':'5.5.5.2', 'network_value':'5.5.5.0/24', 'nodeid':1},
       {'address':'5.5.5.3', 'network_value':'5.5.5.0/24', 'nodeid':2}]
```

Parameters

- **interface_id** (*str,int*) – interface id to assign VLAN.

- **vlan_id** (*str, int*) – vlan identifier
- **nodes** (*list*) – optional addresses for node interfaces (NDI's). For a cluster, each node will require an address specified using the nodes format.
- **cluster_virtual** (*str*) – cluster virtual ip address (optional). If specified, cluster_mask parameter is required
- **network_value** (*str*) – Specifies the network address, i.e. if cluster virtual is 1.1.1.1, cluster mask could be 1.1.1.0/24.
- **macaddress** (*str*) – (optional) if used will provide the mapping from node interfaces to participate in load balancing.
- **cvi_mode** (*str*) – cvi mode for cluster interface (default: packetdispatch)
- **zone_ref** – zone to assign, can be name, str href or Zone
- **kw** (*dict*) – keyword arguments are passed to top level of VLAN interface, not the base level physical interface. This is useful if you want to pass in a configuration that enables the DHCP server on a VLAN for example.

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

Note: If the `interface_id` specified already exists, it is still possible to add additional VLANs and interface addresses.

---

add_layer3_vlan_interface(*interface_id*, *vlan_id*, *address=None*, *network_value=None*, *virtual_mapping=None*, *virtual_resource_name=None*, *zone_ref=None*, *comment=None*, ***kw*)

Add a Layer 3 VLAN interface. Optionally specify an address and network if assigning an IP to the VLAN. This method will also assign an IP address to an existing VLAN, or add an additional address to an existing VLAN. This method may commonly be used on a Master Engine to create VLANs for virtual firewall engines.

Example of creating a VLAN and passing kwargs to define a DHCP server service on the VLAN interface:

```
engine = Engine('engine1')
engine.physical_interface.add_layer3_vlan_interface(interface_id=20, vlan_
↪id=20,
    address='20.20.20.20', network_value='20.20.20.0/24', comment='foocomment
↪',
    dhcp_server_on_interface={
        'default_gateway': '20.20.20.1',
        'default_lease_time': 7200,
        'dhcp_address_range': '20.20.20.101-20.20.20.120',
        'dhcp_range_per_node': [],
        'primary_dns_server': '8.8.8.8'})
```

Parameters

- **interface_id** (*str, int*) – interface identifier
- **vlan_id** (*int*) – vlan identifier
- **address** (*str*) – optional IP address to assign to VLAN
- **network_value** (*str*) – network cidr if address is specified. In format: 10.10.10.0/24.

- **zone_ref** (*str*) – zone to use, by name, href, or Zone
- **comment** (*str*) – optional comment for VLAN level of interface
- **virtual_mapping** (*int*) – virtual engine mapping id See *smc.core.engine.VirtualResource.vfw_id*
- **virtual_resource_name** (*str*) – name of virtual resource See smc.core.engine.VirtualResource.name
- **kw** (*dict*) – keyword arguments are passed to top level of VLAN interface, not the base level physical interface. This is useful if you want to pass in a configuration that enables the DHCP server on a VLAN for example.

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

class smc.core.collection.**TunnelInterfaceCollection**(*engine*)

Bases: *smc.core.collection.InterfaceCollection*

TunnelInterface Collection provides an interface to retrieving existing interfaces and helper methods to shortcut the creation of an interface.

add_cluster_virtual_interface(*interface_id*, *cluster_virtual=None*, *network_value=None*, *nodes=None*, *zone_ref=None*, *comment=None*)

Add a tunnel interface on a clustered engine. For tunnel interfaces on a cluster, you can specify a CVI only, NDI interfaces, or both. This interface type is only supported on layer 3 firewall engines.

```
Add a tunnel CVI and NDI:

engine.tunnel_interface.add_cluster_virtual_interface(
    interface_id_id=3000,
    cluster_virtual='4.4.4.1',
    network_value='4.4.4.0/24',
    nodes=nodes)

Add tunnel NDI's only:

engine.tunnel_interface.add_cluster_virtual_interface(
    interface_id=3000,
    nodes=nodes)

Add tunnel CVI only:

engine.tunnel_interface.add_cluster_virtual_interface(
    interface_id=3000,
    cluster_virtual='31.31.31.31',
    network_value='31.31.31.0/24',
    zone_ref='myzone')
```

Parameters

- **interface_id** (*str,int*) – tunnel identifier (akin to interface_id)
- **cluster_virtual** (*str*) – CVI ipaddress (optional)
- **network_value** (*str*) – CVI network; required if cluster_virtual set
- **nodes** (*list*) – nodes for clustered engine with address,network_value,nodeid
- **zone_ref** (*str*) – zone reference, can be name, href or Zone

- **comment** (*str*) – optional comment

add_layer3_interface(*interface_id*, *address*, *network_value*, *zone_ref=None*, *comment=None*)
Creates a tunnel interface with sub-type single_node_interface. This is to be used for single layer 3 firewall instances.

Parameters

- **interface_id** (*str, int*) – the tunnel id for the interface, used as nicid also
- **address** (*str*) – ip address of interface
- **network_value** (*str*) – network cidr for interface; format: 1.1.1.0/24
- **zone_ref** (*str*) – zone reference for interface can be name, href or Zone
- **comment** (*str*) – optional comment

Raises ***EngineCommandFailed*** – failure during creation

Returns None

class smc.core.collection.**VirtualPhysicalInterfaceCollection**(*engine*)
Bases: *smc.core.collection.InterfaceCollection*

PhysicalInterface Collection provides an interface to retrieving existing interfaces and helper methods to shortcut the creation of an interface.

add_layer3_interface(*interface_id*, *address*, *network_value*, *zone_ref=None*, *comment=None*, ***kw*)
Add a layer 3 interface on a virtual engine.

Parameters

- **interface_id** (*str, int*) – interface identifier
- **address** (*str*) – ip address
- **network_value** (*str*) – network/cidr (12.12.12.0/24)
- **zone_ref** (*str*) – zone reference, can be name, href or Zone
- **kw** – keyword arguments are passed are any value attribute values of type *smc.core.sub_interfaces.NodeInterface*

Raises ***EngineCommandFailed*** – failure creating interface

Returns None

Note: If an existing ip address exists on the interface and zone_ref is provided, this value will overwrite any previous zone definition.

---

add_tunnel_interface(*interface_id*, *address*, *network_value*, *zone_ref=None*, *comment=None*)
Creates a tunnel interface for a virtual engine.

Parameters

- **interface_id** (*str, int*) – the tunnel id for the interface, used as nicid also
- **address** (*str*) – ip address of interface
- **network_value** (*str*) – network cidr for interface; format: 1.1.1.0/24
- **zone_ref** (*str*) – zone reference for interface can be name, href or Zone

Raises ***EngineCommandFailed*** – failure during creation

Returns None

Interface module encapsulates interface types for security engines. All interface have a 'top level' such as Physical or Tunnel Interface. These top level interfaces have certain common settings that can be modified such as assigning a zone.

IP addresses, netmask, management settings, VLANs, etc are part of an interfaces 'sub' interface. Sub interfaces can be retrieved from an engine reference and call to *sub_interfaces()*

The interface hierarchy resembles:

```
    Interface
        |
Physical/Tunnel Interface
        |
        | - VlanInterface (is a PhysicalInterface)
        |      /
    Sub Interfaces (SingleNodeInterface, NodeInterface, InlineInterface, etc)
        |
    Attributes (address, network_value, vlan_id, etc)
```

Sub interfaces are documented in *smc.core.sub_interfaces*.

VLANs are properties of specific interfaces and can also be retrieved by first getting the top level interface, and calling *vlan_interface()* to view or modify specific aspects of a VLAN, such as addresses, etc.

class smc.core.interfaces.**Interface**(***meta*)

Interface settings common to all interface types.

addresses

Return 3-tuple with (address, network, nicid)

Returns address related information of interface as 3-tuple list

Return type list

all_interfaces

Access to all assigned sub-interfaces on this interface. A sub interface is the node level where IP addresses are assigned, or a inline interface is defined, VLANs, etc. Example usage:

```
>>> engine = Engine('dingo')
>>> itf = engine.interface.get(0)
>>> assigned = itf.all_interfaces
>>> list(assigned)
[SingleNodeInterface(address=1.1.1.1)]
>>> assigned.get(address='1.1.1.1')
SingleNodeInterface(address=1.1.1.1)
>>> itf = engine.interface.get(52)
>>> assigned = itf.all_interfaces
>>> list(assigned)
[Layer3PhysicalInterfaceVlan(name=VLAN 52.52),␣
↪Layer3PhysicalInterfaceVlan(name=VLAN 52.53)]
>>> vlan = assigned.get(vlan_id='52')
>>> vlan.addresses
[(u'52.52.52.52', u'52.52.52.0/24', u'52.52')]
```

Return type *BaseIterable*(AllInterfaces)

change_interface_id(*interface_id*)

Change the interface ID for this interface. This can be used on any interface type. If the interface is an

Inline interface, you must provide the `interface_id` in format '1-2' to define both interfaces in the pair. The change is committed after calling this method.

```
itf = engine.interface.get(0)
itf.change_interface_id(10)
```

Or inline interface pair 10-11:

```
itf = engine.interface.get(10)
itf.change_interface_id('20-21')
```

Parameters **interface_id** (*str*, *int*) – new interface ID. Format can be single value for non-inline interfaces or '1-2' format for inline.

Raises ***UpdateElementFailed*** – changing the interface failed with reason

Returns None

comment

Optional interface comment

Returns str or None

contact_addresses

Configure an interface contact address for this interface. Note that an interface may have multiple IP addresses assigned so you may need to iterate through contact addresses. Example usage:

```
>>> itf = engine.interface.get(0)
>>> itf.contact_addresses
[ContactAddressNode(interface_id=0, interface_ip=1.1.1.10),
 ContactAddressNode(interface_id=0, interface_ip=1.1.1.25)]
>>> for ca in itf.contact_addresses:
...   print("IP: %s, addresses: %s" % (ca.interface_ip, list(ca)))
...
IP: 1.1.1.10, addresses: []
IP: 1.1.1.25, addresses: [InterfaceContactAddress(address=172.18.1.20,␣
↪location=Default)]

>>> for ca in itf.contact_addresses:
...   if ca.interface_ip == '1.1.1.10':
...     ca.add_contact_address('10.5.5.5', location='remote')
```

Returns list of interface contact addresses

Return type *ContactAddressNode*

See also:

smc.core.contact_address

delete()

Override delete in parent class, this will also delete the routing configuration referencing this interface.

```
engine = Engine('vm')
interface = engine.interface.get(2)
interface.delete()
```

delete_invalid_route()
Delete any invalid routes for this interface. An invalid route is a left over when an interface is changed to a different network.

Returns None

get_boolean(*name*)
Get the boolean value for attribute specified from the sub interface/s.

has_interfaces
Does the interface have interface have any sub interface types assigned. For example, a physical interface with no IP addresses would return False.

Returns Does this interface have actual types assigned

Return type bool

has_vlan
Does the interface have VLANs

Returns Whether VLANs are configured

Return type bool

interface_id
The Interface ID automatically maps to a physical network port of the same number during the initial configuration of the engine, but the mapping can be changed as necessary. Call *change_interface_id()* to change inline, VLAN, cluster and single interface ID's.

Note: It is not possible to change an interface ID from a VlanInterface. You must call on the parent PhysicalInterface.

Parameters **value** (*str*) – interface_id

Return type str

interfaces
Access to assigned *sub-interfaces* on this interface. A sub interface is the node level where IP addresses are assigned, or a layer 2 interface is defined.

```
>>> itf = engine.interface.get(20)
>>> assigned = itf.interfaces
>>> list(assigned)
[SingleNodeInterface(address=20.20.20.20), SingleNodeInterface(address=21.21.
↪21.21)]
>>> assigned.get(address='20.20.20.20')
SingleNodeInterface(address=20.20.20.20)
```

Return type *BaseIterable*(*SubInterfaceCollection*)

name
Read only name tag

reset_interface()
Reset the interface by removing all assigned addresses and VLANs. This will not delete the interface itself, only the sub interfaces that may have addresses assigned. This will not affect inline or capture interfaces. Note that if this interface is used as a primary control, auth request or outgoing interface, the

update will fail. You should move that functionality to another interface before calling this. See also:: `smc.core.engine.interface_options`.

Raises ***UpdateElementFailed*** – failed to update the interfaces. This is usually caused when the interface is assigned as a control, outgoing, or auth_request interface.

Returns None

sub_interfaces()

Flatten out all top level interfaces and only return sub interfaces. It is recommended to use *all_interfaces()*, *interfaces()* or `vlan_interfaces()` which return collections with helper methods to get sub interfaces based on index or attribute value pairs.

Return type list(*SubInterface*)

update(**args*, ***kw*)

Update/save this interface information back to SMC. When interface changes are made, especially to sub interfaces, call *update* on the top level interface.

Example of changing the IP address of an interface:

```
>>> engine = Engine('sg_vm')
>>> interface = engine.physical_interface.get(1)
>>> interface.zone_ref = zone_helper('mynewzone')
>>> interface.update()
```

Raises ***UpdateElementFailed*** – failure to save changes

Returns Interface

update_interface(*other_interface*, *ignore_mgmt=True*)

Update an existing interface by comparing values between two interfaces. If a VLAN interface is defined in the other interface and it doesn't exist on the existing interface, it will be created.

Parameters

- **Interface** (*other_interface*) – an instance of an interface where values in this interface will be used to as the template to determine changes. This only has to provide attributes that need to change (or not).
- **ignore_mgmt** (*bool*) – ignore resetting management fields. These are generally better set after creation using *engine.interface_options*

Raises ***UpdateElementFailed*** – Failed to update the element

Returns (Interface, modified, created)

Return type tuple

Note: Interfaces with multiple IP addresses are ignored

---

vlan_interface

Access VLAN interfaces for this interface, if any. Example usage:

```
>>> itf = engine.interface.get(52)
>>> assigned = itf.vlan_interface
>>> list(assigned)
[Layer3PhysicalInterfaceVlan(name=VLAN 52.52),␣
↪Layer3PhysicalInterfaceVlan(name=VLAN 52.53)]
```

(continues on next page)

(continued from previous page)

```
>>> vlan = assigned.get(vlan_id='52')
>>> vlan.addresses
[(u'52.52.52.52', u'52.52.52.0/24', u'52.52')]
>>> assigned.get(address='12.12.12.13')
SingleNodeInterface(address=12.12.12.13, vlan_id=1)
>>> assigned.get(vlan_id='1')
SingleNodeInterface(address=12.12.12.12, vlan_id=1)
>>> assigned.get(vlan_id='2')
SingleNodeInterface(address=36.35.35.37, vlan_id=2)
```

Return type *BaseIterable*(VlanInterface)

zone
Return the Zone for this interface, otherwise None

Returns Zone or None

zone_ref
Zone for this physical interface.

Parameters **value** (*str*) – href of zone, set to None to remove existing zone

Return type str

13.5.5.2 InterfaceOptions

class smc.core.interfaces.**InterfaceOptions**(*engine*)
Interface Options allow you to define settings related to the roles of the firewall interfaces:

- Which IP addresses are used as the primary and backup Control IP address
- Which interfaces are used as the primary and backup heartbeat interface
- The default IP address for outgoing traffic

You can optionally change which interface is used for each of these purposes, and define a backup Control IP address and backup Heartbeat Interface. If calling the *set* methods, using a value of None will unset the option.

Note: Setting an interface option will commit the change immediately.

---

auth_request
Return the interface for authentication requests. Can be either a PhysicalInterface or LoopbackInterface

Returns interface id

Return type str

backup_heartbeat
Obtain the interface specified as the backup heartbeat interface. This may return None if a backup has not been specified or this is not a cluster.

Returns interface id

Return type str

backup_mgt
Obtain the interface specified as the backup management interface. This can return None if no backup has been defined

Returns interface id

Return type str

outgoing
Obtain the interface specified as the "Default IP address for outgoing traffic". This will always return a value.

Returns interface id

Return type str

primary_heartbeat
Obtain the interface specified as the primary heartbeat interface. This will return None if this is not a clustered engine.

Returns interface id

Return type str

primary_mgt
Obtain the interface specified as the primary management interface. This will always return a value as you must have at least one physical interface specified for management.

Returns interface id

Return type str

set_auth_request(*interface_id*, *address=None*)
Set the authentication request field for the specified engine.

set_backup_heartbeat(*interface_id*)
Set this interface as the backup heartbeat interface. Clusters and Master NGFW Engines only.

Parameters **interface_id** (*str*, *int*) – interface as backup

Raises

- ***InterfaceNotFound*** – specified interface is not found
- ***UpdateElementFailed*** – failure to update interface

Returns None

set_backup_mgt(*interface_id*)
Set this interface as a backup management interface.

Backup management interfaces cannot be placed on an interface with only a CVI (requires node interface/s). To 'unset' the specified interface address, set interface id to None

```
engine.interface_options.set_backup_mgt(2)
```

Set backup on interface 1, VLAN 201:

```
engine.interface_options.set_backup_mgt('1.201')
```

Remove management backup from engine:

```
engine.interface_options.set_backup_mgt(None)
```

Parameters **interface_id** (*str*, *int*) – interface identifier to make the backup management server.

Raises

- ***InterfaceNotFound*** – specified interface is not found
- ***UpdateElementFailed*** – failure to make modification

Returns None

set_outgoing(*interface_id*)

Specifies the IP address that the engine uses to initiate connections (such as for system communications and ping) through an interface that has no Node Dedicated IP Address. In clusters, you must select an interface that has an IP address defined for all nodes. Setting primary_mgt also sets the default outgoing address to the same interface.

Parameters **interface_id** (*str*, *int*) – interface to set outgoing

Raises

- ***InterfaceNotFound*** – specified interface is not found
- ***UpdateElementFailed*** – failure to make modification

Returns None

set_primary_heartbeat(*interface_id*)

Set this interface as the primary heartbeat for this engine. This will 'unset' the current primary heartbeat and move to specified interface_id. Clusters and Master NGFW Engines only.

Parameters **interface_id** (*str*, *int*) – interface specified for primary mgmt

Raises

- ***InterfaceNotFound*** – specified interface is not found
- ***UpdateElementFailed*** – failed modifying interfaces

Returns None

set_primary_mgt(*interface_id*, *auth_request=None*, *address=None*)

Specifies the Primary Control IP address for Management Server contact. For single FW and cluster FW's, this will enable 'Outgoing', 'Auth Request' and the 'Primary Control' interface. For clusters, the primary heartbeat will NOT follow this change and should be set separately using *set_primary_heartbeat()*. For virtual FW engines, only auth_request and outgoing will be set. For master engines, only primary control and outgoing will be set.

Primary management can be set on an interface with single IP's, multiple IP's or VLANs.

```
engine.interface_options.set_primary_mgt(1)
```

Set primary management on a VLAN interface:

```
engine.interface_options.set_primary_mgt('1.100')
```

Set primary management and different interface for auth_request:

```
engine.interface_options.set_primary_mgt(
    interface_id='1.100', auth_request=0)
```

Set on specific IP address of interface VLAN with multiple addresses:

```
engine.interface_options.set_primary_mgt(
    interface_id='3.100', address='50.50.50.1')
```

Parameters

- **interface_id** (*str, int*) – interface id to make management
- **address** (*str*) – if the interface for management has more than one ip address, this specifies which IP to bind to.
- **auth_request** (*str, int*) – if setting primary mgt on a cluster interface with no CVI, you must pick another interface to set the auth_request field to (default: None)

Raises

- ***InterfaceNotFound*** – specified interface is not found
- ***UpdateElementFailed*** – updating management fails

Returns None

Note: Setting primary management on a cluster interface with no CVI requires you to set the interface for auth_request.

13.5.5.3 QoS

class smc.core.interfaces.**QoS**(*interface*)

QoS can be placed on physical interfaces, physical VLAN interfaces and tunnel interfaces. It is possible to have multiple QoS policies defined if using VLANs on a physical interface as QoS can be attached directly at the interface level or VLAN level. You obtain the QoS reference after retrieving the interface:

```
itf = engine.interface.get(0)
itf.qos.full_qos(100000, QoSPolicy('testqos'))
itf.update()
```

Disable QoS:

```
itf = engine.interface.get(0)
itf.qos.disable()
itf.update()
```

On a tunnel interface:

```
itf = engine.interface.get(1000)
itf.qos.full_qos(100000, QoSPolicy('testqos'))
itf.update()
```

Or a VLAN:

```
itf = engine.interface.get('1.100')
itf.qos.full_qos(100000, QoSPolicy('testqos'))
itf.update()
```

Note: You must call *update* on the interface to commit the change

---

disable()

Disable QoS on this interface

dscp_marking_and_throttling(*qos_policy*)
Enable DSCP marking and throttling on the interface. This requires that you provide a QoS policy to which identifies DSCP tags and how to prioritize that traffic.

Parameters **qos_policy** (QoSPolicy) – the qos policy to apply to the interface

full_qos(*qos_limit*, *qos_policy*)
Enable full QoS on the interface. Full QoS requires that you set a bandwidth limit (in Mbps) for the interface. You must also provide a QoS policy to which identifies the parameters for prioritizing traffic.

Parameters

- **qos_limit** (*int*) – max bandwidth in Mbps
- **qos_policy** (QoSPolicy) – the qos policy to apply to the interface

qos_limit
QoS Limit for this interface. The limit represents the number in bps. For example, 100000 represents 100Mbps.

Return type int

qos_mode
QoS mode in string format

Return type str

qos_policy
QoS Policy for this interface/vlan. A QoS policy will only be present if DSCP throttling or Full QoS is specified.

Return type *QoSPolicy*

statistics_only()
Set interface to collect QoS statistics only. No enforcement is being done but visiblity will be provided in dashboards against applications tagged by QoS.

13.5.5.4 LoopbackInterface

class smc.core.sub_interfaces.**LoopbackInterface**(*data*, *engine=None*)
Bases: *smc.core.sub_interfaces.NodeInterface*

Loopback interface for a physical or virtual single firewall. To create a loopback interface, call from the engine node:

```
engine.loopback_interface.add_single(...)
```

add_single(*address*, *rank=1*, *nodeid=1*, *ospf_area=None*, ***kwargs*)
Add a single loopback interface to this engine. This is used for single or virtual FW engines.

Parameters

- **address** (*str*) – ip address for loopback
- **nodeid** (*int*) – nodeid to apply. Default to 1 for single FW
- **Element ospf_area** (*str,*) – ospf area href or element

Raises ***UpdateElementFailed*** – failure to create loopback address

Returns None

delete()
Delete a loopback interface from this engine. Changes to the engine configuration are done immediately.

A simple way to obtain an existing loopback is to iterate the loopbacks or to get by address:

```
lb = engine.loopback_interface.get('127.0.0.10')
lb.delete()
```

Warning: When deleting a loopback assigned to a node on a cluster all loopbacks with the same rank will also be removed.

Raises *UpdateElementFailed* – failure to delete loopback interface

Returns None

13.5.5.5 LoopbackClusterInterface

class smc.core.sub_interfaces.**LoopbackClusterInterface**(*data*, *engine=None*)
Bases: *smc.core.sub_interfaces.ClusterVirtualInterface*

This represents the CVI Loopback IP address. A CVI loopback IP address is used for loopback traffic that is sent to the whole cluster. It is shared by all the nodes in the cluster.

add_cvi_loopback(*address*, *ospf_area=None*, ***kw*)
Add a loopback interface as a cluster virtual loopback. This enables the loopback to 'float' between cluster members. Changes are committed immediately.

Parameters

- **address** (*str*) – ip address for loopback
- **rank** (*int*) – rank of this entry
- **ospf_area** (*str,Element*) – optional ospf_area to add to loopback

Raises *UpdateElementFailed* – failure to save loopback address

Returns None

add_node_loopback(*nodes*, *ospf_area=None*)
Add loopback interfaces to a cluster. When adding a loopback on a cluster, every cluster node must have a loopback defined or you can optionally configure a loopback CVI address.

Nodes should be in the format:

```
{'address': '127.0.0.10', 'nodeid': 1,
 'address': '127.0.0.11', 'nodeid': 2}
```

Parameters

- **nodes** (*dict*) – nodes defintion for cluster nodes
- **ospf_area** (*str*) – optional OSPF area for this loopback

Raises *EngineCommandFailed* – failed creating loopback

delete()
Delete a loopback cluster virtual interface from this engine. Changes to the engine configuration are done immediately.

You can find cluster virtual loopbacks by iterating at the engine level:

```
for loopbacks in engine.loopback_interface:
    ...
```

Raises *UpdateElementFailed* – failure to delete loopback interface

Returns None

13.5.5.6 PhysicalInterface

class smc.core.interfaces.**PhysicalInterface**(*engine=None*, *meta=None*, ***interface*)

Bases: *smc.core.interfaces.Interface*

Physical Interfaces on NGFW. This represents the following base configuration for the following interface types:

- Single Node Interface
- Node Interface
- Capture Interface
- Inline Interface
- Cluster Virtual Interface
- Virtual Physical Interface (used on Virtual Engines)
- DHCP Interface

This should be used to add interfaces to an engine after it has been created. First get the engine context by loading the engine then get the engine property for physical interface:

```
engine = Engine('myfw')
engine.physical_interface.add_layer3_interface(.....)
engine.physical_interface.add(5) #single unconfigured physical interface
engine.physical_interface.add_inline_ips_interface('5-6', ....)
....
```

When making changes, the etag used should be the top level engine etag.

aggregate_mode

LAGG configuration mode for this interface. Values are 'ha' or 'lb' (load balancing). This can return None if LAGG is not configured.

Returns aggregate mode set, if any

Return type str, None

arp_entry

Return any manually configured ARP entries for this physical interface

Returns arp entries as dict

Return type list

change_vlan_id(*original*, *new*)

Change VLAN ID for a single VLAN, cluster VLAN or inline interface. When changing a single or cluster FW vlan, you can specify the original VLAN and new VLAN as either single int or str value. If modifying an inline interface VLAN when the interface pair has two different VLAN identifiers per interface, use a str value in form: '10-11' (original), and '20-21' (new).

Single VLAN id:

```
>>> engine = Engine('singlefw')
>>> itf = engine.interface.get(1)
>>> itf.vlan_interfaces()
[PhysicalVlanInterface(vlan_id=11), PhysicalVlanInterface(vlan_id=10)]
>>> itf.change_vlan_id(11, 100)
>>> itf.vlan_interfaces()
[PhysicalVlanInterface(vlan_id=100), PhysicalVlanInterface(vlan_id=10)]
```

Inline interface with unique VLAN on each interface pair:

```
>>> itf = engine.interface.get(2)
>>> itf.vlan_interfaces()
[PhysicalVlanInterface(vlan_id=2-3)]
>>> itf.change_vlan_id('2-3', '20-30')
>>> itf.vlan_interfaces()
[PhysicalVlanInterface(vlan_id=20-30)]
```

Parameters

- **original** (*str*, *int*) – original VLAN to change.
- **new** (*str*, *int*) – new VLAN identifier/s.

Raises

- ***InterfaceNotFound*** – VLAN not found
- ***UpdateElementFailed*** – failed updating the VLAN id

Returns None

enable_aggregate_mode (*mode*, *interfaces*)

Enable Aggregate (LAGG) mode on this interface. Possible LAGG types are 'ha' and 'lb' (load balancing). For HA, only one secondary interface ID is required. For load balancing mode, up to 7 additional are supported (8 max interfaces).

Parameters

- **mode** (*str*) – 'lb' or 'ha'
- **interfaces** (*list(str, int)*) – secondary interfaces for this LAGG

Raises ***UpdateElementFailed*** – failed adding aggregate

Returns None

is_auth_request

Is this physical interface tagged as the interface for authentication requests

Return type bool

is_backup_heartbeat

Is this physical interface tagged as the backup heartbeat interface for this cluster.

Returns is backup heartbeat

Return type bool

is_backup_mgt

Is this physical interface tagged as the backup management interface for this cluster.

Returns is backup heartbeat

Return type bool

is_outgoing
Is this the default interface IP used for outgoing for system communications.

Returns is dedicated outgoing IP interface

Return type bool

is_primary_heartbeat
Is this physical interface tagged as the primary heartbeat interface for this cluster.

Returns is backup heartbeat

Return type bool

is_primary_mgt
Is this physical interface tagged as the backup management interface for this cluster.

Returns is backup heartbeat

Return type bool

mtu
Set MTU on interface. Enter a value between 400-65535. The same MTU is automatically applied to any VLANs created under this physical interface

Parameters **value** (*int*) – MTU

Return type int

multicast_ip
Enter a multicast address, that is, an IP address from the range 224.0.0.0-239.255.255.255. The address is used for automatically calculating a MAC address. Required only if multicastigmp cvi mode is selected as the cvi_mode.

Parameters **value** (*str*) – address

Return type str

ndi_interfaces
Return a formatted dict list of NDI interfaces on this engine. This will ignore CVI or any inline or layer 2 interface types. This can be used to identify to indicate available IP addresses for a given interface which can be used to run services such as SNMP or DNS Relay.

Returns list of dict items [{'address':x, 'nicid':y}]

Return type list(dict)

qos
The QoS settings for this physical interface

Return type *QoS*

second_interface_id
Peer interfaces used in LAGG configuration.

Parameters **value** (*str*) – comma seperated nic id's for LAGG peers

Return type str

static_arp_entry(*ipaddress*, *macaddress*, *arp_type='static'*, *netmask=32*)
Add an arp entry to this physical interface.

```
interface = engine.physical_interface.get(0)
interface.static_arp_entry(
    ipaddress='23.23.23.23',
```

(continues on next page)

(continued from previous page)

```
    arp_type='static',
    macaddress='02:02:02:02:04:04')
interface.save()
```

Parameters

- **ipaddress** (*str*) – ip address for entry
- **macaddress** (*str*) – macaddress for ip address
- **arp_type** (*str*) – type of entry, 'static' or 'proxy' (default: static)
- **netmask** (*str, int*) – netmask for entry (default: 32)

Returns None

virtual_engine_vlan_ok
Whether to allow VLAN creation on the Virtual Engine. Only valid for master engine.

Parameters **value** (*bool*) – enable/disable

Return type bool

virtual_mapping
The virtual mapping id. Required if Virtual Resource chosen. See *smc.core.engine.VirtualResource.vfw_id*

Parameters **value** (*int*) – vfw_id

Return type int

virtual_resource_name
Virtual Resource name used on Master Engine to map a virtual engine. See `smc.core.engine.VirtualResource.name`

Parameters **value** (*str*) – virtual resource name

Return type str

13.5.5.7 Layer3PhysicalInterface

class smc.core.interfaces.**Layer3PhysicalInterface**(*engine=None*, *meta=None*, ***interface*)

Bases: *smc.core.interfaces.PhysicalInterface*

Represents a routed layer 3 interface on an any engine type.

Example interface:

```
interface = {
    'comment': u'Regular interface',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'address': u'5.5.5.2',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1},
                              {'address': u'5.5.5.3',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1}]}],
     'zone_ref': 'foozone'}
```

Layer3 VLAN interface:

```
interface = {
    'comment': u'Interface with VLAN',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'address': u'5.5.5.2',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1},
                              {'address': u'5.5.5.3',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1}],
                    'vlan_id': 10}],
     'zone_ref': 'foozone'}
```

DHCP interface on a VLAN (use *dynamic* and specify *dynamic_index*):

```
interface = {
    'comment': u'Interface with VLAN',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'dynamic': True,
                               'dynamic_index': 2,
                               'nodeid': 1}],
                    'vlan_id': 10}],
     'zone_ref': 'foozone'}
```

When an interface is created, the first key level is applied to the "top" level physical interface. The *interfaces* list specifies the node and addressing information using the *nodes* parameter. If *vlan_id* is specified as a key/value in the interfaces dict, the list dict keys are applied to the nested physical interface VLAN.

Parameters

- **interface_id** (*str*) – id for interface
- **interface** (*str*) – specifies the type of interface to create. The interface type defaults to 'node_interface' and applies to all engine types except a single FW. For single FW, specify *single_node_interface*
- **interfaces** (*list*) – interface attributes, *cluster_virtual*, *network_value*, *nodes*, etc
- **nodes** (*dict*) – nodes dict should contain keys *address*, *network_value* and *nodeid*. Overridden sub interface settings can also be set here
- **zone_ref** (*str*) – zone reference, name or zone
- **comment** (*str*) – comment for interface

13.5.5.8 Layer2PhysicalInterface

class smc.core.interfaces.**Layer3PhysicalInterface**(*engine=None*, *meta=None*, ***interface*)

Bases: *smc.core.interfaces.PhysicalInterface*

Represents a routed layer 3 interface on an any engine type.

Example interface:

```
interface = {
    'comment': u'Regular interface',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'address': u'5.5.5.2',
```

(continues on next page)

(continued from previous page)

```
                                'network_value': u'5.5.5.0/24',
                                'nodeid': 1},
                               {'address': u'5.5.5.3',
                                'network_value': u'5.5.5.0/24',
                                'nodeid': 1}]}],
     'zone_ref': 'foozone'}
```

Layer3 VLAN interface:

```
interface = {
    'comment': u'Interface with VLAN',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'address': u'5.5.5.2',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1},
                              {'address': u'5.5.5.3',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1}],
                    'vlan_id': 10}],
     'zone_ref': 'foozone'}
```

DHCP interface on a VLAN (use *dynamic* and specify *dynamic_index*):

```
interface = {
    'comment': u'Interface with VLAN',
    'interface_id': u'67',
    'interfaces': [{'nodes': [{'dynamic': True,
                               'dynamic_index': 2,
                               'nodeid': 1}],
                    'vlan_id': 10}],
     'zone_ref': 'foozone'}
```

When an interface is created, the first key level is applied to the "top" level physical interface. The *interfaces* list specifies the node and addressing information using the *nodes* parameter. If *vlan_id* is specified as a key/value in the interfaces dict, the list dict keys are applied to the nested physical interface VLAN.

Parameters

- **interface_id** (*str*) – id for interface
- **interface** (*str*) – specifies the type of interface to create. The interface type defaults to 'node_interface' and applies to all engine types except a single FW. For single FW, specify *single_node_interface*
- **interfaces** (*list*) – interface attributes, *cluster_virtual*, *network_value*, *nodes*, etc
- **nodes** (*dict*) – nodes dict should contain keys *address*, *network_value* and *nodeid*. Overridden sub interface settings can also be set here
- **zone_ref** (*str*) – zone reference, name or zone
- **comment** (*str*) – comment for interface

13.5.5.9 ClusterPhysicalInterface

class smc.core.interfaces.**ClusterPhysicalInterface**(*engine=None*, *meta=None*, ***interface*)

Bases: *smc.core.interfaces.PhysicalInterface*

A ClusterPhysicalInterface represents an interface on a cluster that is a physical interface type. A cluster interface can have a CVI, NDI's, or CVI's and NDI's.

Example interface format, with CVI and 2 nodes:

```
 interface = {
     'interface_id': '23',
     'comment': 'my comment',
     'zone_ref': 'zone1',
     'cvi_mode': 'packetdispatch',
     'macaddress': '02:08:08:02:02:06',
     'interfaces': [{'cluster_virtual': '241.241.241.250',
                     'network_value': '241.241.241.0/24',
                     'nodes': [{'address': '241.241.241.2', 'network_value': '241.
↪241.241.0/24', 'nodeid': 1},
                               {'address': '241.241.241.3', 'network_value': '241.
↪241.241.0/24', 'nodeid': 2}]
                     }]}

Example interface with VLAN and CVI / NDI::

interface = {
    'interface_id': '24',
    'cvi_mode': 'packetdispatch',
    'macaddress': '02:02:08:08:08:06',
    'interfaces': [{'cluster_virtual': '242.242.242.250',
                    'network_value': '242.242.242.0/24',
                    'nodes': [{'address': '242.242.242.2', 'network_value': '242.
↪242.242.0/24', 'nodeid': 1},
                              {'address': '242.242.242.3', 'network_value': '242.
↪242.242.0/24', 'nodeid': 2}],
                    'vlan_id': 24,
                    'zone_ref': 'vlanzone',
                    'comment': 'comment on vlan'}],
    'zone_ref': zone_helper('myzone'),
    'comment': 'top level interface'}
```

When an interface is created, the first key level is applied to the "top" level physical interface. The *interfaces* list specifies the node and addressing information using the *nodes* parameter. If *vlan_id* is specified as a key/value in the interfaces dict, the list dict keys are applied to the nested physical interface VLAN.

Parameters

- **interface_id** (*str*) – id for interface
- **cvi_mode** – cvi mode type (i.e. packetdispatch), required when using CVI
- **macaddress** (*str*) – mac address for top level physical interface. Required if CVI set
- **interfaces** (*list*) – interface attributes, *cluster_virtual*, *network_value*, *nodes*, etc
- **nodes** (*dict*) – nodes dict should contain keys *address*, *network_value* and *nodeid*. Overridden sub interface settings can also be set here
- **zone_ref** (*str,href*) – zone reference, name or zone. If zone does not exist it will be created
- **comment** (*str*) – comment for interface

Note: Values for dict match the FirewallCluster.create constructor

cvi_mode
HA Cluster mode.

Returns possible values: packetdispatch, unicast, multicast, multicastgmp

Return type str

macaddress
MAC Address for cluster virtual interface. Not required for NDI only interfaces.

Parameters **value** (*str*) – macaddress

Return type str

13.5.5.10 VirtualPhysicalInterface

class smc.core.interfaces.**VirtualPhysicalInterface**(*engine=None*, *meta=None*, ***interface*)
Bases: *smc.core.interfaces.Layer3PhysicalInterface*

This interface type is used by virtual engines and has subtle differences to a normal interface. For a VE in layer 3 firewall, it also specifies a Single Node Interface as the physical interface sub-type. When creating the VE, one of the interfaces must be designated as the source for Auth Requests and Outgoing.

13.5.5.11 TunnelInterface

class smc.core.interfaces.**TunnelInterface**(*engine=None*, *meta=None*, ***interface*)
Bases: *smc.core.interfaces.Interface*

This interface type represents a tunnel interface that is typically used for route based VPN traffic. Nested interface nodes can be SingleNodeInterface (for L3 NGFW), a NodeInterface (for cluster's with only NDI's) or ClusterVirtualInterface (CVI) for cluster VIP. Tunnel Interfaces are only available under layer 3 routed interfaces and do not support VLANs.

Example tunnel interface format on cluster FW:

```
cluster_tunnel_interface = {
    'comment': u'My Tunnel on cluster',
    'interface_id': u'1000',
    'interfaces': [{'cluster_virtual': u'77.77.77.70',
                    'network_value': u'77.77.77.0/24',
                    'nodes': [{'address': u'5.5.5.2',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1},
                              {'address': u'5.5.5.3',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 2}]}],
     'zone_ref': 'foozone'}
```

Tunnel interface on single FW with multiple tunnel IPs:

```
single_fw_interface = {
    'comment': u'Tunnel with two addresses on single FW',
    'interface_id': u'1000',
    'interfaces': [{'nodes': [{'address': u'5.5.5.2',
                               'network_value': u'5.5.5.0/24',
                               'nodeid': 1},
                              {'address': u'5.5.5.3',
```

(continues on next page)

(continued from previous page)

```
                                'network_value': u'5.5.5.0/24',
                                'nodeid': 1}]
                            }],
    'zone_ref': 'foozone'}
```

qos
The QoS settings for this tunnel interface

Return type *QoS*

13.5.5.12 Sub-Interfaces

Module provides an interface to sub-interfaces on an engine. A 'top level' interface is linked from the engine and will be PhysicalInterface, TunnelInterface, etc. Within the top level interface, there are sub-interface configurations that identify the basic settings such as ip address, network, administrative settings etc. These are not called directly but used as a reference to the top level interface. All sub interfaces are type dict.

class smc.core.sub_interfaces.**CaptureInterface**(*data*)
Capture Interface (SPAN) This is a single physical interface type that can be installed on either layer 2 or IPS engine roles. It enables the NGFW to capture traffic on the wire without actually blocking it (although blocking is possible).

Variables

- **inspect_unspecified_vlans** (*boolean*) – promiscuous SPAN on unspecified VLANs
- **logical_interface_ref (required)** (*str*) – logical interface to use, by href
- **reset_interface_nicid** (*int*) – if sending passive RST back, interface id to use
- ***nicid*** (*str,int*) – nicid for this capture interface

class smc.core.sub_interfaces.**ClusterVirtualInterface**(*data*)
These interfaces (CVI) are used on cluster devices and applied to layer 3 interfaces. They specify a 'VIP' (or shared IP) to be used for traffic load balancing or high availability. Each engine will still also have a 'node' interface for communication to/from the engine itself. The following getter/setter properties are available:

Variables

- **address** (*str*) – address of the CVI
- ***auth_request*** (*boolean*) – interface for authentication requests (only 1)
- **network_value** (*str*) – network address for interface, i.e. 1.1.1.0/24
- ***nicid*** (*int*) – nic interface identifier
- **relayed_by_dhcp** (*boolean*) – is the interface using DHCP
- **igmp_mode** (*str*) – IGMP mode (upstream/downstream/None)

vlan_id
VLAN ID for this interface, if any

Returns VLAN identifier

Return type str

class smc.core.sub_interfaces.**InlineIPSInterface**(*data*)
An Inline IPS Interface is a new interface type introduced in SMC version 6.3. This interface type is the same as a normal IPS interface except that it is applied on a Layer 3 Firewall.

New in version 0.5.6: Requires SMC 6.3.

class smc.core.sub_interfaces.**InlineInterface**(*data*)

This interface type is used on layer 2 or IPS related engines. It requires that you specify two interfaces to be part of the inline pair. These interfaces do not need to be sequential. It is also possible to add VLANs and zones to the inline interfaces. The logical interface reference needs to be unique for inline and capture interfaces when they are applied on the same engine.

Variables

- **inspect_unspecified_vlans** (*boolean*) – promiscuous SPAN on unspecified VLANs
- **logical_interface_ref (required)** (*str*) – logical interface to use, by href
- **failure_mode** (*str*) – normal or bypass
- ***nicid*** (*str*) – interfaces for inline pair, for example, '4.50-5.55', '5-6'
- ***vlan_id*** (*str*) – vlan identifier for interface
- ***zone_ref* (optional)** (*str*) – zone for second interface in pair

change_interface_id(*newid*)

Change the inline interface ID. The current format is nicid='1-2', where '1' is the top level interface ID (first), and '2' is the second interface in the pair. Consider the existing nicid in case this is a VLAN.

Parameters **newid** (*str*) – string defining new pair, i.e. '3-4'

Returns None

change_vlan_id(*vlan_id*)

Change a VLAN id for an inline interface.

Parameters **vlan_id** (*str*) – New VLAN id. Can be in format '1-2' or a single numerical value. If in '1-2' format, this specifies the vlan ID for the first inline interface and the rightmost for the second.

Returns None

vlan_id

VLAN ID for this interface, if any

Returns VLAN identifier

Return type str

class smc.core.sub_interfaces.**InlineL2FWInterface**(*data*)

An Inline L2FW Interface is a new interface type introduced in SMC version 6.3. This interface type is the a layer 2 FW interface on a layer 3 firewall. By default this interface type does not support bypass mode and will discard on overload.

New in version 0.5.6: Requires SMC 6.3.

class smc.core.sub_interfaces.**LoopbackClusterInterface**(*data*, *engine=None*)

This represents the CVI Loopback IP address. A CVI loopback IP address is used for loopback traffic that is sent to the whole cluster. It is shared by all the nodes in the cluster.

add_cvi_loopback(*address*, *ospf_area=None*, ***kw*)

Add a loopback interface as a cluster virtual loopback. This enables the loopback to 'float' between cluster members. Changes are committed immediately.

Parameters

- **address** (*str*) – ip address for loopback

- **rank** (*int*) – rank of this entry
- **ospf_area** (*str,Element*) – optional ospf_area to add to loopback

Raises *UpdateElementFailed* – failure to save loopback address

Returns None

add_node_loopback(*nodes*, *ospf_area=None*)

Add loopback interfaces to a cluster. When adding a loopback on a cluster, every cluster node must have a loopback defined or you can optionally configure a loopback CVI address.

Nodes should be in the format:

```
{'address': '127.0.0.10', 'nodeid': 1,
 'address': '127.0.0.11', 'nodeid': 2}
```

Parameters

- **nodes** (*dict*) – nodes defintion for cluster nodes
- **ospf_area** (*str*) – optional OSPF area for this loopback

Raises *EngineCommandFailed* – failed creating loopback

delete()

Delete a loopback cluster virtual interface from this engine. Changes to the engine configuration are done immediately.

You can find cluster virtual loopbacks by iterating at the engine level:

```
for loopbacks in engine.loopback_interface:
    ...
```

Raises *UpdateElementFailed* – failure to delete loopback interface

Returns None

class smc.core.sub_interfaces.**LoopbackInterface**(*data*, *engine=None*)

Loopback interface for a physical or virtual single firewall. To create a loopback interface, call from the engine node:

```
engine.loopback_interface.add_single(...)
```

add_single(*address*, *rank=1*, *nodeid=1*, *ospf_area=None*, ***kwargs*)

Add a single loopback interface to this engine. This is used for single or virtual FW engines.

Parameters

- **address** (*str*) – ip address for loopback
- **nodeid** (*int*) – nodeid to apply. Default to 1 for single FW
- **Element ospf_area** (*str,*) – ospf area href or element

Raises *UpdateElementFailed* – failure to create loopback address

Returns None

delete()

Delete a loopback interface from this engine. Changes to the engine configuration are done immediately.

A simple way to obtain an existing loopback is to iterate the loopbacks or to get by address:

```
lb = engine.loopback_interface.get('127.0.0.10')
lb.delete()
```

Warning: When deleting a loopback assigned to a node on a cluster all loopbacks with the same rank will also be removed.

Raises *UpdateElementFailed* – failure to delete loopback interface

Returns None

class smc.core.sub_interfaces.**NodeInterface**(*data*)

Node Interface Node dedicated interface (NDI) is used on specific engine types and represents an interface used for management (IPS and layer 2 engines), or as normal layer 3 interfaces such as on a layer 3 firewall cluster.

For Layer 2 Firewall/IPS these are used as individual interfaces. On clusters, these are used to define the node specific address for each node member, along with a cluster virtual interface.

Variables

- **address** (*str*) – ip address of this interface
- **network_value** (*str*) – network for this interface, i.e. 1.1.1.0/24
- **or int nicid** (*str*) – nic interface id
- ***nodeid*** (*int*) – node identifier for interface (in a cluster, each node will be unique)
- ***outgoing*** (*boolean*) – This option defines the IP address that the nodes use if they have to initiate connections (system communications, ping, etc.) through an interface that has no Node Dedicated IP Address. In Firewall Clusters, you must select an interface that has an IP address defined for all nodes.
- ***primary_heartbeat*** (*boolean*) – Whether interface is the primary heartbeat interface for communications between the nodes. It is recommended that you use a Physical Interface, not a VLAN Interface. It is also recommended that you do not direct any other traffic through this interface.
- ***primary_mgt*** (*boolean*) – Is it the Primary Control Interface for Management Server contact. There must be one and only one Primary Control Interface
- ***auth_request*** (*boolean*) – whether to specify this interface as interface for authentication requests. Should be set on interface acting as management
- **auth_request_source** (*boolean*) – If the authentication requests are sent to an external authentication server over VPN, select an interface with a Node Dedicated IP address that you want use for the authentication requests
- **reverse_connection** (*boolean*) – Reverse connection enables engine to contact SMC versus other way around
- ***vlan_id*** (*str*) – VLAN id for interface if assigned
- ***backup_mgt*** (*boolean*) – Whether interface is a backup control interface that is used if the primary control interface is not available
- ***backup_heartbeat*** (*boolean*) – Whether the interface is a backup heartbeat. It is not mandatory to configure a backup heartbeat interface.
- **dynamic** (*boolean*) – Whether this is a DHCP interface

- **dynamic_index** (*int*) – The dynamic index of the DHCP interface. The value is between 1 and 16. Only used when 'dynamic' is set to True.
- **igmp_mode** (*str*) – IGMP mode (upstream/downstream/None)
- **vrrp** (*boolean*) – Enable VRRP
- **vrrp_address** (*str*) – IP address if VRRP is enabled
- **vrrp_id** (*int*) – The VRRP ID. Required only for VRRP mode
- **vrrp_priority** (*int*) – The VRRP Priority. Required only for VRRP mode

vlan_id
VLAN ID for this interface, if any

Returns VLAN identifier

Return type str

class smc.core.sub_interfaces.**SingleNodeInterface**(*data*)
This interface is used by single node Layer 3 Firewalls. This type of interface can be a management interface as well as a non-management routed interface.

Variables

- **dynamic** (*bool*) – is this interface a dynamic DHCP interface
- **dynamic_index** (*int*) – dynamic interfaces index value
- **automatic_default_route** (*boolean*) – Flag to know if the dynamic default route will be automatically created for this dynamic interface. Used in DHCP interfaces only

class smc.core.sub_interfaces.**SubInterface**(*data*)

change_interface_id(*interface_id*)
Generic change interface ID for VLAN interfaces that are not Inline Interfaces (non-VLAN sub interfaces do not have an interface_id field).

Parameters **int interface_id** (*str,*) – interface ID value

change_vlan_id(*vlan_id*)
Change a VLAN id

Parameters **vlan_id** (*str*) – new vlan

class smc.core.sub_interfaces.**SubInterfaceCollection**(*interface*)
A Sub Interface collection for non-VLAN interfaces.

13.5.5.13 InterfaceContactAddress

A ContactAddress is used by elements to provide an alternate address for communication between engine and management/log server. This is typically used when the SMC sits behind a NAT address and the SMC needs to contact the engine directly (this is a default behavior). In this case, you would add the public IP in front of the engine as a contact address to the engine interface.

Obtain all eligible interfaces for contact addressess:

```
>>> engine = Engine('dingo')
>>> for ca in engine.contact_addresses:
...   ca
...
```

(continues on next page)

(continued from previous page)

```
ContactAddressNode(interface_id=11, interface_ip=10.10.10.20)
ContactAddressNode(interface_id=120, interface_ip=120.120.120.100)
ContactAddressNode(interface_id=0, interface_ip=1.1.1.1)
ContactAddressNode(interface_id=12, interface_ip=3.3.3.3)
ContactAddressNode(interface_id=12, interface_ip=17.17.17.17)
```

Retrieve a specific contact address interface for modification:

```
>>> ca = engine.contact_addresses.get(interface_id=12, interface_ip='3.3.3.3')
>>> ca
ContactAddressNode(interface_id=12, interface_ip=3.3.3.3)
>>> list(ca)
[InterfaceContactAddress(location=Default,address=4.4.4.4),␣
↪InterfaceContactAddress(location=Foo,address=3.4.5.6)]
```

Add a new contact address to the fetched interface:

```
>>> ca.add_contact_address('23.23.23.23', location='mynewlocation')
>>> list(ca)
[InterfaceContactAddress(location=Default,address=4.4.4.4),␣
↪InterfaceContactAddress(location=Foo,address=3.4.5.6),
 InterfaceContactAddress(location=mynewlocation,address=23.23.23.23)]
```

Remove a contact address:

```
>>> ca.remove_contact_address('23.23.23.23')
>>> list(ca)
[InterfaceContactAddress(location=Default,address=4.4.4.4),␣
↪InterfaceContactAddress(location=Foo,address=3.4.5.6)]
```

Note: Contact Addresses for servers (Management/Log Server) do not use this same object definition

class smc.core.contact_address.**ContactAddressCollection**(*resource*)

Bases: *smc.base.collection.SubElementCollection*

A contact address collection provides all available interfaces that can be used to configure a contact address. An eligible interface is one that is a layer 3 interface with an address assigned (including VLANs):

```
for ca in engine.contact_addresses:
    ...
```

Note: All eligible interfaces are returned, regardless of whether a contact address is assigned or not.

get(*interface_id*, *interface_ip=None*)

Get will return a list of interface references based on the specified interface id. Multiple references can be returned if a single interface has multiple IP addresses assigned.

Returns If interface_ip is provided, a single ContactAddressNode element is returned if found. Otherwise a list will be returned with all contact address nodes for the given interface_id.

class smc.core.contact_address.**ContactAddressNode**(***meta*)

Bases: *smc.base.model.SubElement*

A mapping of contact address to interface. This is specific to assigning the contact address on the engine.

add_contact_address(*contact_address*, *location='Default'*)
Add a contact address to this specified interface. A contact address is an alternative address which is typically applied when NAT is used between the NGFW and another component (such as management server). Adding a contact address operation is committed immediately.

Parameters **contact_address** (*str*) – IP address for this contact address.

Raises ***EngineCommandFailed*** – invalid contact address

Returns ContactAddressNode

delete(*location_name*)
Remove a given location by location name. This operation is performed only if the given location is valid, and if so, *update* is called automatically.

Parameters **location** (*str*) – location name or location ref

Raises ***UpdateElementFailed*** – failed to update element with reason

Return type bool

interface_id
The interface ID for this contact address interface

Return type str

interface_ip
The IP address for this contact address interface

Return type str

remove_contact_address(*location*)
Remove a contact address from an interface by the location name. There is a one to one relationship between a contact address and

Parameters **contact_address** (*str*) – ip for contact address

Raises ***EngineCommandFailed*** – problem removing address

Returns status of delete as boolean

Return type bool

update_or_create(*location*, *contact_address*, *with_status=False*, ***kw*)
Update an existing contact address or create if the location does not exist.

Parameters

- **location** (*str*) – name of the location, the location will be added if it doesn't exist
- **contact_address** (*str*) – contact address IP. Can be the string 'dynamic' if this should be a dynamic contact address (i.e. on DHCP interface)
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises ***UpdateElementFailed*** – failed to update element with reason

Return type *ContactAddressNode*

class smc.core.contact_address.**InterfaceContactAddress**(*data=None*, ***kwargs*)
Bases: *smc.elements.other.ContactAddress*

An interface contact address is used on engine interfaces to provide an alternative location to address mapping. This is frequently used when the engine sits behind a NAT and you need a public NAT mapping, as might be the case with site to site VPN.

13.5.6 Node

Node level actions for an engine. Once an engine is loaded, all methods and resources are available to that particular engine.

For example, to load an engine and run node level commands:

```
engine = Engine('myfw')
for node in engine.nodes:
    node.reboot()
    node.bind_license()
    node.go_online()
    node.go_offline()
    ...
    ...
```

class smc.core.node.**Node**(***meta*)

Bases: *smc.base.model.SubElement*

Node settings to make each engine node controllable individually. Obtain a reference to a Node by loading an Engine resource. Engine will have a 'has-a' relationship with node and stored as the nodes attribute.

```
>>> for node in engine.nodes:
...   node
...
Node(name=fwcluster node 1)
Node(name=fwcluster node 2)
```

appliance_info()

New in version 0.5.7: Requires SMC version >= 6.3

Retrieve appliance info for this engine.

Raises ***NodeCommandFailed*** – Appliance info not supported on this node

Return type *ApplianceInfo*

bind_license(*license_item_id=None*)

Auto bind license, uses dynamic if POS is not found

Parameters **license_item_id** (*str*) – license id

Raises ***LicenseError*** – binding license failed, possibly no licenses

Returns None

cancel_unbind_license()

Cancel unbind for license

Raises ***LicenseError*** – unbind failed with reason

Returns None

certificate_info()

Get the certificate info of this node. This can return None if the engine type does not directly have a certificate, like a virtual engine where the master engine manages certificates.

Returns dict with links to cert info

change_ssh_pwd(*pwd=None*, *comment=None*)

Executes a change SSH password operation on the specified node

Parameters

- **pwd** (*str*) – changed password value
- **comment** (*str*) – optional comment for audit log

Raises ***NodeCommandFailed*** – cannot change ssh password

Returns None

debug(*filter_enabled=False*)

View all debug settings for this node. This will return a debug object. View the debug object repr to identify settings to enable or disable and submit the object to *set_debug()* to enable settings.

Add filter_enabled=True argument to see only enabled settings

Parameters **filter_enabled** (*bool*) – returns all enabled diagnostics

Raises ***NodeCommandFailed*** – failure getting diagnostics

Return type *Debug*

See also:

Debug for example usage

dynamic_element_update(*name_cache_object*)

fetch_license()

Fetch the node level license

Raises ***LicenseError*** – fetching license failure with reason

Returns None

go_offline(*comment=None*)

Executes a Go-Offline operation on the specified node

Parameters **comment** (*str*) – optional comment to audit

Raises ***NodeCommandFailed*** – offline not available

Returns None

go_online(*comment=None*)

Executes a Go-Online operation on the specified node typically done when the node has already been forced offline via *go_offline()*

Parameters **comment** (*str*) – (optional) comment to audit

Raises ***NodeCommandFailed*** – online not available

Returns None

go_standby(*comment=None*)

Executes a Go-Standby operation on the specified node. To get the status of the current node/s, run *status()*

Parameters **comment** (*str*) – optional comment to audit

Raises ***NodeCommandFailed*** – engine cannot go standby

Returns None

hardware_status

Obtain hardware statistics for various areas of this node.

See *HardwareStatus* for usage.

Raises ***NodeCommandFailed*** – failure to retrieve current status

Return type *HardwareStatus*

health
Basic status for individual node. Specific information such as node name dynamic package version, configuration status, platform and version.

Return type *ApplianceStatus*

initial_contact(*enable_ssh=True*, *time_zone=None*, *keyboard=None*, *install_on_server=None*, *filename=None*, *as_base64=False*)
Allows to save the initial contact for for the specified node

Parameters

- **enable_ssh** (*bool*) – flag to know if we allow the ssh daemon on the specified node
- **time_zone** (*str*) – optional time zone to set on the specified node
- **keyboard** (*str*) – optional keyboard to set on the specified node
- **install_on_server** (*bool*) – optional flag to know if the generated configuration needs to be installed on SMC Install server (POS is needed)
- **filename** (*str*) – filename to save initial_contact to
- **as_base64** (*bool*) – return the initial config in base 64 format. Useful for cloud based engine deployments as userdata

Raises ***NodeCommandFailed*** – IOError handling initial configuration data

Returns initial contact text information

Return type str

interface_status
Obtain the interface status for this node. This will return an iterable that provides information about the existing interfaces. Retrieve a single interface status:

```
>>> node = engine.nodes[0]
>>> node
Node(name=ngf-1065)
>>> node.interface_status
<smc.core.node.InterfaceStatus object at 0x103b2f310>
>>> node.interface_status.get(0)
InterfaceStatus(aggregate_is_active=False, capability=u'Normal Interface',
    flow_control=u'AutoNeg: off Rx: off Tx: off',
    interface_id=0, mtu=1500, name=u'eth0_0', port=u'Copper',
    speed_duplex=u'1000 Mb/s / Full / Automatic', status=u'Up')
```

Or iterate and get all interfaces:

```
>>> for stat in node.interface_status:
...   stat
...
InterfaceStatus(aggregate_is_active=False, capability=u'Normal Interface', ...
...
```

Raises ***NodeCommandFailed*** – failure to retrieve current status

Return type *InterfaceStatus*

lock_offline(*comment=None*)
Executes a Lock-Offline operation on the specified node Bring back online by running *go_online()*.

Parameters **comment** (*str*) – comment for audit

Raises ***NodeCommandFailed*** – lock offline failed

Returns None

lock_online(*comment=None*)
Executes a Lock-Online operation on the specified node

Parameters **comment** (*str*) – comment for audit

Raises ***NodeCommandFailed*** – cannot lock online

Returns None

loopback_interface
Loopback interfaces for this node. This will return empty if the engine is not a layer 3 firewall type:

```
>>> engine = Engine('dingo')
>>> for node in engine.nodes:
...   for loopback in node.loopback_interface:
...     loopback
...
LoopbackInterface(address=172.20.1.1, nodeid=1, rank=1)
LoopbackInterface(address=172.31.1.1, nodeid=1, rank=2)
LoopbackInterface(address=2.2.2.2, nodeid=1, rank=3)
```

Return type list(*LoopbackInterface*)

nodeid
ID of this node

power_off()
New in version 0.5.6: Requires engine version >=6.3

Power off engine.

Raises ***NodeCommandFailed*** – online not available

Returns None

reboot(*comment=None*)
Send reboot command to this node.

Parameters **comment** (*str*) – comment to audit

Raises ***NodeCommandFailed*** – reboot failed with reason

Returns None

rename(*name*)
Rename this node

Parameters **name** (*str*) – new name for node

reset_to_factory()
New in version 0.5.6: Requires engine version >=6.3

Reset the engine to factory defaults.

Raises ***NodeCommandFailed*** – online not available

Returns None

reset_user_db(*comment=None*)
Executes a Send Reset LDAP User DB Request operation on this node.

Parameters **comment** (*str*) – comment to audit

Raises ***NodeCommandFailed*** – failure resetting db

Returns None

set_debug(*debug*)
Set the debug settings for this node. This should be a modified *Debug* instance. This will take effect immediately on the specified node.

Parameters **debug** (Debug) – debug object with specified settings

Raises ***NodeCommandFailed*** – fail to communicate with node

Returns None

See also:

Debug for example usage

sginfo(*include_core_files=False*, *include_slapcat_output=False*, *filename='sginfo.gz'*)
Get the SG Info of the specified node. Optionally provide a filename, otherwise default to 'sginfo.gz'. Once you run gzip -d <filename>, the inner contents will be in .tar format.

Parameters

- **include_core_files** – flag to include or not core files
- **include_slapcat_output** – flag to include or not slapcat output

Raises ***NodeCommandFailed*** – failed getting sginfo with reason

Returns string path of download location

Return type str

ssh(*enable=True*, *comment=None*)
Enable or disable SSH

Parameters

- **enable** (*bool*) – enable or disable SSH daemon
- **comment** (*str*) – optional comment for audit

Raises ***NodeCommandFailed*** – cannot enable SSH daemon

Returns None

status()
Basic status for individual node. Specific information such as node name dynamic package version, configuration status, platform and version.

Return type *ApplianceStatus*

time_sync()
Send a time sync command to this node.

Raises ***NodeCommandFailed*** – time sync not supported on node

Returns None

type
Node type

unbind_license()
Unbind a bound license on this node.

Raises *LicenseError* – failure with reason

Returns None

version
Engine version. If the node is not yet initialized, this will return None.

Returns str or None

13.5.6.1 Appliance Info

class smc.core.node.**ApplianceInfo**(*cloud_id*, *cloud_type*, *first_upload_time*, *hardware_version*, *initial_contact_time*, *product_name*, *initial_license_remaining_days*, *proof_of_serial*, *software_features*, *software_version*)

Bases: tuple

Appliance specific information about the given engine node. Appliance info is specific to the engine itself and will provide additional details about the hardware model, applied license features, if the engine has made initial contact and when initial policy upload was made.

Retrieve appliance info engine nodes:

```
engine = Engine('dingo')
for node in engine.nodes:
    node.appliance_info()
```

Variables

- **cloud_id** (*str*) – N/A
- **cloud_type** (*str*) – N/A
- **first_upload_time** (*long*) – policy first upload time in ms
- **hardware_version** (*float*) – hardware version of appliance
- **initial_contact_time** (*long*) – when node contacted SMC, in ms
- **intial_license_remaining_days** (*int*) – validity in days of current license
- **product_name** (*str*) – name of hardware model
- **proof_of_serial** (*str*) – proof of serial for this hardware
- **software_features** (*str*) – feature string
- **software_version** (*str*) – initial software version on base image

13.5.6.2 Appliance Status

class smc.core.node.**ApplianceStatus**(*configuration_status*, *dyn_up*, *installed_policy*, *name*, *platform*, *state*, *status*, *version*, *engine_node_status*, *monitoring_state*, *monitoring_status*)

Bases: tuple

Appliance status attributes define specifics about the hardware platform itself, including version, dynamic package, current configuration status and installed policy. Retrieve appliance status for engine nodes:

```
for node in engine.nodes:
    node.health
```

Variables

- **dyn_up** (*str*) – Dynamic update package version
- ***installed_policy*** (*str*) – Installed policy by name
- ***name*** (*str*) – Name of engine
- ***platform*** (*str*) – Underlying platform, x86, etc
- ***version*** (*str*) – Version of software installed
- **configuration_status** (*str*) –

  Valid values:

  – Initial (no initial configuration file is yet generated)

  – Declared (initial configuration file is generated)

  – Configured (initial configuration is done with the engine)

  – Installed (policy is installed on the engine)
- ***status*** (*str*) –

  Valid values: Not Monitored/Unknown/Online/Going Online/Locked Online/ Going Locked Online/Offline/Going Offline/Locked Offline/ Going Locked Offline/Standby/Going Standby/No Policy Installed
- ***state*** (*str*) –

  Valid values: INITIAL/READY/ERROR/SERVER_ERROR/NO_STATUS/TIMEOUT/ DELETED/DUMMY

Changed in version 0.6.2: Attribute status changed to monitoring_status and state to monitoring_state in SMC 6.5

13.5.6.3 Hardware Status

class smc.core.node.**HardwareStatus**(*status*)

Bases: *smc.base.structs.SerializedIterable*

Provides an interface to methods that simplify viewing hardware statuses on this node. Example of usage:

```
>>> engine = Engine('sg_vm')
>>> node = engine.nodes[0]
>>> node
Node(name=ngf-1065)
>>> node.hardware_status
HardwareStatus(Anti-Malware, File Systems, GTI Cloud, Sandbox, Logging subsystem,␣
↪MLC Connection, Web Filtering)
>>> node.hardware_status.filesystem
HardwareCollection(File Systems, items: 5)
>>> for stats in node.hardware_status.filesystem:
...   stats
...
Status(label=u'Root', param=u'Partition Size', status=-1, sub_system=u'File␣
↪Systems', value=u'600 MB')
```

(continues on next page)

(continued from previous page)

```
Status(label=u'Data', param=u'Usage', status=-1, sub_system=u'File Systems',␣
↪value=u'6.3%')
Status(label=u'Data', param=u'Size', status=-1, sub_system=u'File Systems',␣
↪value=u'1937 MB')
Status(label=u'Spool', param=u'Usage', status=-1, sub_system=u'File Systems',␣
↪value=u'4.9%')
Status(label=u'Spool', param=u'Size', status=-1, sub_system=u'File Systems',␣
↪value=u'9729 MB')
Status(label=u'Tmp', param=u'Usage', status=-1, sub_system=u'File Systems',␣
↪value=u'0.0%')
Status(label=u'Tmp', param=u'Size', status=-1, sub_system=u'File Systems', value=u
↪'3941 MB')
Status(label=u'Swap', param=u'Usage', status=-1, sub_system=u'File Systems',␣
↪value=u'0.0%')
Status(label=u'Swap', param=u'Size', status=-1, sub_system=u'File Systems',␣
↪value=u'1887 MB')
```

filesystem
A collection of filesystem related statuses

Return type *Status*

logging_subsystem
A collection of logging subsystem statuses

Return type *Status*

class smc.core.node.**Status**(*label*, *param*, *status*, *sub_system*, *value*)
Bases: tuple

Status fields for hardware status. These fields have generic titles which are used to represent the field and values for each hardware type.

Variables

- **label** (*str*) – name for this field
- **param** (*str*) – field this measures
- ***status*** (*int*) – unused
- **sub_system** (*str*) – category for this hardware status
- ***value*** (*str*) – value for this field

13.5.6.4 Interface Status

class smc.core.node.**InterfaceStatus**(*status*)
Bases: *smc.base.structs.SerializedIterable*

An iterable that provides a collections interface to interfaces and current status on the specified node.

Interface status fields:

Variables

- **aggregate_is_active** (*bool*) – Is link aggregation enabled on this interface
- **capability** (*str*) – What type of interface this is, i.e. “Normal Interface”
- **flow_control** (*str*) – Autonegotiation, etc

- ***interface_id*** (*int*) – Physical interface id
- ***mtu*** (*int*) – Max transmission unit
- ***name*** (*str*) – Name of the interface, i.e. eth0_0, etc
- ***port*** (*str*) – Type of physical port, i.e. Copper, Fiber
- **speed_duplex** (*str*) – Negotiated speed on the interface
- ***status*** (*str*) – Status of interface, Up, Down, etc.

get (*interface_id*)
Get a specific interface by the interface id

Parameters **interface_id** (*int*) – interface ID

Return type *InterfaceStatus*

13.5.6.5 Debug

class smc.core.node.**Debug**(*diag*)
Debug settings that can be enabled on the engine. To view available options, print the repr of this object. All diagnostic values can be set as an attribute of this class instance. Set the values to either True or False and submit this object back to the node to change settings. Setting changes are in effect immediately and does not require a policy push. Example usage:

```
>>> node = engine.nodes[0]
>>> node
Node(name=ngf-1065)
>>> debug = node.debug()
>>> debug
Debug(access_guardian=False, accounting=False, anti_malware=False,␣
↪authentication=False,
    blacklisting=False, browser_based_user_authentication=False, cluster_
↪daemon=False,
    cluster_protocol=False, connection_tracking=False, data_synchronization=False,
    dhcp_client=False, dhcp_relay=False, dhcp_service=False, dns_resolution=False,
    dynamic_routing=False, endpoint_integration=False, file_reputation=False,
    inspection=False, invalid=False, ipsec_vpn=False, licensing=False,
    load_balancing_filter=False, log_server=False, logging_system=False,␣
↪management=False,
    mcafee_logon_collector=False, monitoring=False, multicast_routing=False,␣
↪nat=False,
    netlink_incoming_ha=False, packet_filtering=False, protocol_agent=False,
    radius_forwarder=False, sandbox=False, server_pool_load_balancing=False,␣
↪snmp=False,
    ssl_vpn=False, ssl_vpn_portal=False, ssl_vpn_session_manager=False,
    state_synchronisation=False, syslog=False, system_utilities=False,␣
↪tester=False,
    user_agent=False, wireless_access_point=False)
    >>> debug.management=True
    >>> debug.sandbox=True
    >>> node.set_debug(debug)
```

13.5.7 Pending Changes

class smc.core.resource.**PendingChanges**(*engine*)
Bases: *smc.base.structs.SerializedIterable*

Pending changes apply to the engine having changes that have not yet been committed. Retrieve from the engine level:

```
>>> for changes in engine.pending_changes.all():
...   print(changes, changes.resolve_element)
...
(ChangeRecord(approved_on=u'', changed_on=u'2017-07-12 15:24:40 (GMT)',
element=u'http://172.18.1.150:8082/6.2/elements/fw_cluster/116',
event_type=u'stonegate.object.update', modifier=u'admin'),
FirewallCluster(name=sg_vm))
```

Approve all changes:

```
>>> engine.pending_changes.approve_all()
```

Conversely, reject all pending changes:

```
>>> engine.pending_changes.disapprove_all()
```

Raises ***ActionCommandFailed*** – failure to retrieve pending changes

Return type *ChangeRecord*

approve_all()
Approve all pending changes

Raises ***ActionCommandFailed*** – possible permissions issue

Returns None

disapprove_all()
Disapprove all pending changes

Raises ***ActionCommandFailed*** – possible permissions issue

Returns None

class smc.core.resource.**ChangeRecord**
Bases: *smc.core.resource.ChangeRecord*

Change record details for any pending changes.

Parameters

- **approved_on** – approved on datetime, may be empty if not approved
- **change_on** – changed on datetime
- **element** – element affected
- **event_type** – type of change, update, delete, etc.
- **modifier** – account making the modification

13.5.8 Routing

Route module encapsulates functions related to static routing and related configurations on NGFW. When retrieving routing, it is done from the engine context.

For example, retrieve all routing for an engine in context:

```
>>> engine = Engine('sg_vm')
>>> for route_node in engine.routing:
...   print(route_node)
...
Routing(name=Interface 0,level=interface)
Routing(name=Interface 1,level=interface)
Routing(name=Interface 2,level=interface)
Routing(name=Tunnel Interface 2000,level=interface)
Routing(name=Tunnel Interface 2001,level=interface)
```

Routing nodes are nested, starting with the engine level. Routing node nesting is made up of 'levels' and can be represented as a tree:

```
engine (root)
    |
    --> interface
            |
            --> network
                    |
                    --> gateway
                            |
                            --> any
```

You can get a representation of the routing or antispoofing tree nodes by calling as_tree:

```
>>> print(engine.routing.as_tree())
Routing(name=myfw,level=engine_cluster)
--Routing(name=Interface 0,level=interface)
----Routing(name=network-1.1.1.0/24,level=network)
------Routing(name=mypeering,level=gateway)
------Routing(name=mynetlink,level=gateway)
--------Routing(name=router-1.1.1.1,level=any)
------Routing(name=mystatic,level=gateway)
--Routing(name=Interface 1,level=interface)
----Routing(name=network-10.10.10.0/24,level=network)
------Routing(name=anotherpeering,level=gateway)
--Routing(name=Tunnel Interface 1000,level=interface)
----Routing(name=network-2.2.2.0/24,level=network)
--Routing(name=Tunnel Interface 1001,level=interface)
--Routing(name=Interface 2,level=interface)
----Routing(name=Network (IPv4),level=network)
------Routing(name=dynamic_netlink-myfw-Interface 2,level=gateway)
--------Routing(name=Any network,level=any)
```

If nested routes exist, you can iterate a given node to get specific information:

```
>>> interface = engine.routing.get(1)
>>> for routes in interface:
...   print(routes)
...
Routing(name=network-10.0.0.0/24,level=network)
```

(continues on next page)

(continued from previous page)

```
...
>>> for networks in interface:
...   networks
...   for gateways in networks:
...     print gateways, gateways.ip
...
Routing(name=network-172.18.1.0/24,level=network)
Routing(name=asus-wireless,level=gateway) 172.18.1.200
```

If BGP, OSPF or a Traffic Handler (netlink) need to be added to an interface that has multiple IP addresses assigned and you want to bind to only one, you can provide the `network` parameter to `add_` methods. The network can be obtained for an interface:

```
>>> engine = Engine('sg_vm')
>>> interface0 = engine.routing.get(0)
>>> for network in interface0:
...   network, network.ip
...
(Routing(name=network-172.18.1.0/24,level=network), '172.18.1.0/24')
```

Then add using:

```
>>> engine = Engine('sg_vm')
>>> interface0 = engine.routing.get(0)
>>> interface0.add_traffic_handler(StaticNetlink('foo'), network='172.18.1.0/24')
```

Note: If the `network` keyword is omitted and the interface has multiple IP addresses assigned, this will bind OSPF, BGP or the Traffic Handler to all address assigned.

Adding a basic static route can be done from the engine directly if it is a simple source network to destination route:

```
engine.add_route(gateway='192.168.1.254/32', network='172.18.1.0/24')
```

The route gateway will be mapped to an interface with an address range in the 192.168.1.x network automatically.

For more complex static routes such as ones that may use group elements, use the routing node:

```
>>> engine = Engine('ve-1')
>>> interface0 = engine.routing.get(0)
>>> interface0.add_static_route(Router('tmprouter'), destination=[Group('routegroup
↪')])
```

When a routing gateway is added to an IPv6 network, the gateway is validated before adding. For example, if you have a single interface that has both an IPv4 and IPv6 address assigned, a static route using a Router gateway with only an IPv4 address will only bind to the IPv4 network. In this case, you can optionally add both an IPv4 and IPv6 to the router element, or run this operation for each network respectively.

See also:

Routing.add_static_route()

Note: When changing are made to a routing node, i.e. adding OSPF, BGP, Netlink's, the configuration is updated immediately without calling .update()

---

class smc.core.route.**RoutingTree**(*data=None*, ***meta*)

RoutingTree is the base class for both Routing and Antispoofing nodes. This provides a commmon API for operations that affect how routing table and antispoofing operate.

all()

Return all routes for this engine.

Returns current route entries as *Routing* element

Return type list

as_tree(*level=0*)

Display the routing tree representation in string format

Return type str

delete()

Delete the element

Raises ***DeleteElementFailed*** – possible dependencies, record locked, etc

Returns None

dynamic_nicid

NIC id for this dynamic interface

Returns nic identifier, if this is a DHCP interface

Return type str or None

get(*interface_id*)

Obtain routing configuration for a specific interface by ID.

Note: If interface is a VLAN, you must use a str to specify the interface id, such as '3.13' (interface 3, VLAN 13)

Parameters **interface_id** (*str*, *int*) – interface identifier

Raises ***InterfaceNotFound*** – invalid interface for engine

Returns Routing element, or None if not found

Return type *Routing*

ip

IP network / host for this route

Returns IP address of this routing level

Return type str

level

Routing nodes have multiple 'levels' where routes can be nested. Most routes are placed at the interface level. This setting can mostly be ignored, but provides an informative view of how the route is nested.

Returns routing node level (interface,network,gateway,any)

Return type str

name

Interface name / ID for routing level

Returns name of routing node

Return type str

nicid
NIC id for this interface

Returns nic identifier

Return type str

related_element_type
New in version 0.6.0: Requires SMC version >= 6.4

Related element type defines the 'type' of element at this routing or antispoofing node level.

Return type str

update()
Update the existing element and clear the instance cache. Removing the cache will ensure subsequent calls requiring element attributes will force a new fetch to obtain the latest copy.

Calling update() with no args will assume the element has already been modified directly and the data cache will be used to update. You can also override the following attributes: href, etag and json. If json is sent, it is expected to the be a complete payload to satisfy the update.

For kwargs, if attribute values are a list, you can pass 'append_lists=True' to add to an existing list, otherwise overwrite (default: overwrite)

See also:

To see different ways to utilize this method for updating, see: *Update*.

Parameters

- **exception** – pass a custom exception to throw if failure
- **kwargs** – optional kwargs to update request data to server.

Raises

- ***ModificationFailed*** – raised if element is tagged as System element
- ***UpdateElementFailed*** – failed to update element with reason

Returns href of the element modified

Return type str

13.5.8.1 Routing

class smc.core.route.**Routing**(*data=None*, ***meta*)
Bases: *smc.core.route.RoutingTree*

Routing represents the Engine routing configuration and provides the ability to view and add features to routing nodes such as OSPF.

add_bgp_peering(*bgp_peering*, *external_bgp_peer=None*, *network=None*)
Add a BGP configuration to this routing interface. If the interface has multiple ip addresses, all networks will receive the BGP peering by default unless the `network` parameter is specified.

Example of adding BGP to an interface by ID:

```
interface = engine.routing.get(0)
interface.add_bgp_peering(
    BGPPeering('mypeer'),
    ExternalBGPPeer('neighbor'))
```

Parameters

- **bgp_peering** (BGPPeering) – BGP Peer element
- **external_bgp_peer** (ExternalBGPPeer,Engine) – peer element or href
- **network** (*str*) – if network specified, only add OSPF to this network on interface

Raises

- ***ModificationAborted*** – Change must be made at the interface level
- ***UpdateElementFailed*** – failed to add BGP

Returns Status of whether the route table was updated

Return type bool

add_dynamic_gateway(*networks*)

A dynamic gateway object creates a router object that is attached to a DHCP interface. You can associate networks with this gateway address to identify networks for routing on this interface.

```
route = engine.routing.get(0)
route.add_dynamic_gateway([Network('mynetwork')])
```

Parameters **Network** (*list*) – list of network elements to add to this DHCP gateway

Raises

- ***ModificationAborted*** – Change must be made at the interface level
- ***UpdateElementFailed*** – failure to update routing table

Returns Status of whether the route table was updated

Return type bool

add_ospf_area(*ospf_area*, *ospf_interface_setting=None*, *network=None*, *communication_mode='NOT_FORCED'*, *unicast_ref=None*)

Add OSPF Area to this routing node.

Communication mode specifies how the interface will interact with the adjacent OSPF environment. Please see SMC API documentation for more in depth information on each option.

If the interface has multiple networks nested below, all networks will receive the OSPF area by default unless the `network` parameter is specified. OSPF cannot be applied to IPv6 networks.

Example of adding an area to interface routing node:

```
area = OSPFArea('area0') #obtain area resource

#Set on routing interface 0
interface = engine.routing.get(0)
interface.add_ospf_area(area)
```

Note: If UNICAST is specified, you must also provide a unicast_ref of element type Host to identify the remote host. If no unicast_ref is provided, this is skipped

Parameters

- **ospf_area** (OSPFArea) – OSPF area instance or href
- **ospf_interface_setting** (OSPFInterfaceSetting) – used to override the OSPF settings for this interface (optional)
- **network** (*str*) – if network specified, only add OSPF to this network on interface
- **communication_mode** (*str*) – NOT_FORCED|POINT_TO_POINT|PASSIVE|UNICAST
- **unicast_ref** (Element) – Element used as unicast gw (required for UNICAST)

Raises

- ***ModificationAborted*** – Change must be made at the interface level
- ***UpdateElementFailed*** – failure updating routing
- ***ElementNotFound*** – ospf area not found

Returns Status of whether the route table was updated

Return type bool

add_static_route(*gateway*, *destination*, *network=None*)

Add a static route to this route table. Destination can be any element type supported in the routing table such as a Group of network members. Since a static route gateway needs to be on the same network as the interface, provide a value for *network* if an interface has multiple addresses on different networks.

```
>>> engine = Engine('ve-1')
>>> itf = engine.routing.get(0)
>>> itf.add_static_route(
        gateway=Router('tmprouter'),
        destination=[Group('routegroup')])
```

Parameters

- **gateway** (Element) – gateway for this route (Router, Host)
- **destination** (*list*(Host, Router, ...)) – destination network/s for this route.

Raises

- ***ModificationAborted*** – Change must be made at the interface level
- ***UpdateElementFailed*** – failure to update routing table

Returns Status of whether the route table was updated

Return type bool

add_traffic_handler(*netlink*, *netlink_gw=None*, *network=None*)

Add a traffic handler to a routing node. A traffic handler can be either a static netlink or a multilink traffic handler. If `network` is not specified and the interface has multiple IP addresses, the traffic handler will be added to all ipv4 addresses.

Add a pre-defined netlink to the route table of interface 0:

```
engine = Engine('vm')
rnode = engine.routing.get(0)
rnode.add_traffic_handler(StaticNetlink('mynetlink'))
```

Add a pre-defined netlink only to a specific network on an interface with multiple addresses. Specify a netlink_gw for the netlink:

```
rnode = engine.routing.get(0)
rnode.add_traffic_handler(
    StaticNetlink('mynetlink'),
    netlink_gw=[Router('myrtr'), Host('myhost')],
    network='172.18.1.0/24')
```

Parameters

- **netlink** (StaticNetlink,Multilink) – netlink element
- **netlink_gw** (*list(Element)*) – list of elements that should be destinations for this netlink. Typically these may be of type host, router, group, server, network or engine.
- **network** (*str*) – if network specified, only add OSPF to this network on interface

Raises

- ***UpdateElementFailed*** – failure updating routing
- ***ModificationAborted*** – Change must be made at the interface level
- ***ElementNotFound*** – ospf area not found

Returns Status of whether the route table was updated

Return type bool

bgp_peerings

BGP Peerings applied to a routing node. This can be called from the engine, interface or network level. Return is a tuple of (interface, network, bgp_peering). This simplifies viewing and removing BGP Peers from the routing table:

```
>>> for bgp in engine.routing.bgp_peerings:
...   bgp
...
(Routing(name=Interface 0,level=interface,type=physical_interface),
 Routing(name=network-1.1.1.0/24,level=network,type=network),
 Routing(name=mypeering,level=gateway,type=bgp_peering))
(Routing(name=Interface 1,level=interface,type=physical_interface),
 Routing(name=network-2.2.2.0/24,level=network,type=network),
 Routing(name=mypeering,level=gateway,type=bgp_peering))
```

See also:

netlinks() and *ospf_areas()* for obtaining other routing element types

Return type tuple(*Routing*)

netlinks

Netlinks applied to a routing node. This can be called from the engine, interface or network level. Return is a tuple of (interface, network, netlink). This simplifies viewing and removing Netlinks from the routing table:

```
>>> interface = engine.routing.get(1)
>>> for static_netlink in interface.netlinks:
...   interface, network, netlink = static_netlink
...   netlink
...   netlink.delete()
...
Routing(name=mylink,level=gateway,type=netlink)
```

See also:

bgp_peerings() and *ospf_areas()* for obtaining other routing element types

Return type tuple(*Routing*)

ospf_areas

OSPFv2 areas applied to a routing node. This can be called from the engine, interface or network level. Return is a tuple of (interface, network, bgp_peering). This simplifies viewing and removing BGP Peers from the routing table:

```
>>> for ospf in engine.routing.ospf_areas:
...   ospf
...
(Routing(name=Interface 0,level=interface,type=physical_interface),
 Routing(name=network-1.1.1.0/24,level=network,type=network),
 Routing(name=area10,level=gateway,type=ospfv2_area))
```

See also:

bgp_peerings() and *netlinks()* for obtaining other routing element types

Return type tuple(*Routing*)

remove_route_gateway(*element*, *network=None*)

Remove a route element by href or Element. Use this if you want to remove a netlink or a routing element such as BGP or OSPF. Removing is done from within the routing interface context.

```
interface0 = engine.routing.get(0)
interface0.remove_route_gateway(StaticNetlink('mynetlink'))
```

Only from a specific network on a multi-address interface:

```
interface0.remove_route_gateway(
    StaticNetlink('mynetlink'),
    network='172.18.1.0/24')
```

Parameters

- **element** (*str,Element*) – element to remove from this routing node
- **network** (*str*) – if network specified, only add OSPF to this network on interface

Raises

- ***ModificationAborted*** – Change must be made at the interface level
- ***UpdateElementFailed*** – failure to update routing table

Returns Status of whether the entry was removed (i.e. or not found)

Return type bool

routing_node_element
A routing node element will reference the element used to represent the node (i.e. router, host, network, netlink, bgp peering, etc). Although the routing node already resolves the element and provides the *ip* property to obtain the address/network, use this property to obtain access to modifying the element itself:

```
>>> interface0 = engine.routing.get(0)
>>> for networks in interface0:
...   for gateway in networks:
...     gateway.routing_node_element
...
Router(name=router-1.1.1.1)
StaticNetlink(name=mystatic)
BGPPeering(name=anotherpeering)
BGPPeering(name=mypeering)
>>>
```

13.5.8.2 Antispoofing

class smc.core.route.**Antispoofing**(*data=None*, ***meta*)
Bases: *smc.core.route.RoutingTree*

Anti-spoofing is configured by default based on interface networks directly attached. It is possible to override these settings by adding additional networks as valid source networks on a given interface.

Antispoofing is nested similar to routes. Iterate the antispoofing configuration:

```
for entry in engine.antispoofing.all():
    print(entry)
```

add(*element*)
Add an entry to this antispoofing node level. Entry can be either href or network elements specified in *smc.elements.network*

```
if0 = engine.antispoofing.get(0)
if0.add(Network('foonet'))
```

Parameters **element** (Element) – entry to add, i.e. Network('mynetwork'), Host(..)

Raises

- ***CreateElementFailed*** – failed adding entry
- ***ElementNotFound*** – element entry specified not in SMC

Returns whether entry was added

Return type bool

autogenerated
Was the entry auto generated by a route entry or added manually as an override

Return type bool

remove(*element*)
Remove a specific user added element from the antispoofing tables of a given interface. This will not remove autogenerated or system level entries.

Parameters **element** (Element) – element to remove

Returns remove element if it exists and return bool

Return type bool

validity
Enabled or disabled antispoofing entry

Returns validity of this entry (enable,disable,absolute)

Return type str

13.5.8.3 Route Table

class smc.core.route.**Route**(*data*)
Active routes obtained from a running engine. Obtain routes from an engine reference:

```
>>> engine = Engine('sg_vm')
>>> for route in engine.routing_monitoring:
...     route
```

Variables

- ***route_network*** (*str*) – network for this route
- **route_netmask** (*int*) – netmask for the route
- **route_gateway** (*str*) – route gateway, may be None if it's a local network only
- ***route_type*** (*str*) – status of the route
- **dst_if** (*int*) – destination interface index
- **src_if** (*int*) – source interface index

13.5.8.4 Policy Routing

class smc.core.route.**PolicyRoute**(*engine*)
An iterable providing an interface to policy based routing on the engine. You must call engine.udpate() after performing an add or delete:

```
>>> engine = Engine('myfw')
>>> engine.policy_route
PolicyRoute(items: 1)
>>> for rt in engine.policy_route:
...   rt
...
PolicyRoute(source=u'172.18.1.0/24', destination=u'172.18.1.0/24', gateway_ip=u
↪'172.18.1.1', comment=None)
>>> engine.policy_route.create(source='172.18.2.0/24', destination='192.168.3.0/24
↪', gateway_ip='172.18.2.1')
>>> engine.update()
'http://172.18.1.151:8082/6.4/elements/single_fw/746'
>>> for rt in engine.policy_route:
...   rt
...
PolicyRoute(source=u'172.18.1.0/24', destination=u'172.18.1.0/24', gateway_ip=u
↪'172.18.1.1', comment=None)
```

(continues on next page)

(continued from previous page)

```
PolicyRoute(source=u'172.18.2.0/24', destination=u'192.168.3.0/24', gateway_ip=u
↪'172.18.2.1', comment=None)
>>> engine.policy_route.delete(source='172.18.2.0/24')
>>> engine.update()
'http://172.18.1.151:8082/6.4/elements/single_fw/746'
>>> for rt in engine.policy_route:
...   rt
...
PolicyRoute(source=u'172.18.1.0/24', destination=u'172.18.1.0/24', gateway_ip=u
↪'172.18.1.1', comment=None)
```

Variables

- ***source*** (*str*) – source network/cidr for the route
- ***destination*** (*str*) – destination network/cidr for the route
- **gateway_ip** (*str*) – gateway IP address, must be on source network
- ***comment*** (*str*) – optional comment

create(*source*, *destination*, *gateway_ip*, *comment=None*)

Add a new policy route to the engine.

Parameters

- **source** (*str*) – network address with /cidr
- **destination** (*str*) – network address with /cidr
- **gateway** (*str*) – IP address, must be on source network
- **comment** (*str*) – optional comment

delete(***kw*)

Delete a policy route from the engine. You can delete using a single field or multiple fields for a more exact match. Use a keyword argument to delete a route by any valid attribute.

Parameters **kw** – use valid Route keyword values to delete by exact match

13.5.9 Snapshot

class smc.core.resource.**Snapshot**(***meta*)

Bases: *smc.base.model.SubElement*

Policy snapshots currently held on the SMC. You can retrieve all snapshots at the engine level and view details of each:

```
for snapshot in engine.snapshots:
    print(snapshot)
```

Snapshots can be generated manually, but also will be generated automatically when a policy is pushed:

```
engine.generate_snapshot(filename='mysnapshot.zip')
```

Snapshots can also be downloaded:

```
for snapshot in engine.snapshots:
    if snapshot.name == 'blah snapshot':
        snapshot.download()
```

Snapshot filename will be <snapshot_name>.zip if not specified.

download(*filename=None*)
Download snapshot to filename

Parameters **filename** (*str*) – fully qualified path including filename .zip

Raises ***EngineCommandFailed*** – IOError occurred downloading snapshot

Returns None

13.5.10 VirtualResource

class smc.core.engine.**VirtualResource**(***meta*)
Bases: *smc.base.model.SubElement*

A Virtual Resource is a container placeholder for a virtual engine within a Master Engine. When creating a virtual engine, each virtual engine must have a unique virtual resource for mapping. The virtual resource has an identifier (vfw_id) that specifies the engine ID for that instance.

This is called as a resource of an engine. To view all virtual resources:

```
list(engine.virtual_resource.all())
```

Available attributes:

Variables

- **connection_limit** (*int*) – Maximum number of connections for this virtual engine. 0 means unlimited (default: 0)
- **show_master_nic** (*bool*) – Show the master engine NIC id's in the virtual engine.

When updating this element, make modifications and call update()

allocated_domain_ref
Domain that this virtual engine is allocated in. 'Shared Domain' is is the default if no domain is specified.

```
>>> for resource in engine.virtual_resource:
...   resource, resource.allocated_domain_ref
...
(VirtualResource(name=ve-1), AdminDomain(name=Shared Domain))
(VirtualResource(name=ve-8), AdminDomain(name=Shared Domain))
```

Returns AdminDomain element

Return type *AdminDomain*

create(*name*, *vfw_id*, *domain='Shared Domain'*, *show_master_nic=False*, *connection_limit=0*, *comment=None*)
Create a new virtual resource. Called through engine reference:

```
engine.virtual_resource.create(....)
```

Parameters

- **name** (*str*) – name of virtual resource
- **vfw_id** (*int*) – virtual fw identifier
- **domain** (*str*) – name of domain to install, (default Shared)
- **show_master_nic** (*bool*) – whether to show the master engine NIC ID's in the virtual instance
- **connection_limit** (*int*) – whether to limit number of connections for this instance

Returns href of new virtual resource

Return type str

set_admin_domain(*admin_domain*)

Virtual Resources can be members of an Admin Domain to provide delegated administration features. Assign an admin domain to this resource. Admin Domains must already exist.

Parameters **admin_domain** (*str,AdminDomain*) – Admin Domain to add

Returns None

vfw_id

Read-Only virtual engine identifier. This is unique per virtual engine and is set when the virtual resource is created.

Returns vfw id

Return type int

13.6 Engine Types

13.6.1 IPS

class smc.core.engines.**IPS**(*name*, ***meta*)

Creates an IPS engine with a default inline interface pair

classmethod create(*name*, *mgmt_ip*, *mgmt_network*, *mgmt_interface=0*, *inline_interface='1-2'*, *logical_interface='default_eth'*, *log_server_ref=None*, *domain_server_address=None*, *zone_ref=None*, *enable_antivirus=False*, *enable_gti=False*, *comment=None*)

Create a single IPS engine with management interface and inline pair

Parameters

- **name** (*str*) – name of ips engine
- **mgmt_ip** (*str*) – ip address of management interface
- **mgmt_network** (*str*) – management network in cidr format
- **mgmt_interface** (*int*) – (optional) interface for management from SMC to fw
- **inline_interface** (*str*) – interfaces to use for first inline pair
- **logical_interface** (*str*) – name, str href or LogicalInterface (created if it doesn't exist)
- **log_server_ref** (*str*) – (optional) href to log_server instance
- **domain_server_address** (*list*) – (optional) DNS server addresses

- **zone_ref** (*str*) – zone name, str href or Zone for management interface (created if not found)
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI

Raises ***CreateEngineFailed*** – Failure to create with reason

Returns *smc.core.engine.Engine*

13.6.2 Layer3Firewall

class smc.core.engines.**Layer3Firewall**(*name*, ***meta*)

Represents a Layer 3 Firewall configuration. A layer 3 single FW is a standalone FW instance (not a cluster). You can use the *create* constructor and add interfaces after the engine exists, or use *create_bulk* to fully create the engine and interfaces in a single operation.

classmethod create(*name*, *mgmt_ip*, *mgmt_network*, *mgmt_interface=0*, *log_server_ref=None*, *default_nat=False*, *reverse_connection=False*, *domain_server_address=None*, *zone_ref=None*, *enable_antivirus=False*, *enable_gti=False*, *location_ref=None*, *enable_ospf=False*, *sidewinder_proxy_enabled=False*, *ospf_profile=None*, *snmp=None*, *comment=None*, ***kw*)

Create a single layer 3 firewall with management interface and DNS. Provide the *interfaces* keyword argument if adding multiple additional interfaces. Interfaces can be one of any valid interface for a layer 3 firewall. Unless the interface type is specified, physical_interface is assumed.

Valid interface types:

- physical_interface (default if not specified)
- tunnel_interface

If providing all engine interfaces in a single operation, see *create_bulk()* for the proper format.

Parameters

- **name** (*str*) – name of firewall engine
- **mgmt_ip** (*str*) – ip address of management interface
- **mgmt_network** (*str*) – management network in cidr format
- **log_server_ref** (*str*) – (optional) href to log_server instance for fw
- **mgmt_interface** (*int*) – (optional) interface for management from SMC to fw
- **domain_server_address** (*list*) – (optional) DNS server addresses
- **zone_ref** (*str*) – zone name, str href or zone name for management interface (created if not found)
- **reverse_connection** (*bool*) – should the NGFW be the mgmt initiator (used when behind NAT)
- **default_nat** (*bool*) – (optional) Whether to enable default NAT for outbound
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI
- **sidewinder_proxy_enabled** (*bool*) – Enable Sidewinder proxy functionality

- **location_ref** (*str*) – location href or not for engine if needed to contact SMC behind NAT (created if not found)
- **enable_ospf** (*bool*) – whether to turn OSPF on within engine
- **ospf_profile** (*str*) – optional OSPF profile to use on engine, by ref
- **kw** – optional keyword arguments specifying additional interfaces

Raises *CreateEngineFailed* – Failure to create with reason

Returns *smc.core.engine.Engine*

classmethod create_bulk(*name*, *interfaces=None*, *primary_mgt=None*, *backup_mgt=None*, *log_server_ref=None*, *domain_server_address=None*, *location_ref=None*, *default_nat=False*, *enable_antivirus=False*, *enable_gti=False*, *sidewinder_proxy_enabled=False*, *enable_ospf=False*, *ospf_profile=None*, *comment=None*, *snmp=None*, ***kw*)

Create a Layer 3 Firewall providing all of the interface configuration. This method provides a way to fully create the engine and all interfaces at once versus using *create()* and creating each individual interface after the engine exists.

Example interfaces format:

```
interfaces=[
    {'interface_id': 1},
    {'interface_id': 2,
     'interfaces':[{'nodes': [{'address': '2.2.2.2', 'network_value': '2.2.2.
↪0/24'}]}]},
     'zone_ref': 'myzone'},
    {'interface_id': 3,
     'interfaces': [{'nodes': [{'address': '3.3.3.3', 'network_value': '3.3.3.
↪0/24'}],
                    'vlan_id': 3,
                    'zone_ref': 'myzone'},
                    {'nodes': [{'address': '4.4.4.4', 'network_value': '4.4.
↪4.0/24'}],
                     'vlan_id': 4}]},
    {'interface_id': 4,
     'interfaces': [{'vlan_id': 4,
                    'zone_ref': 'myzone'}]},
    {'interface_id': 5,
     'interfaces': [{'vlan_id': 5}]},
    {'interface_id': 1000,
     'interfaces': [{'nodes': [{'address': '10.10.10.1', 'network_value': '10.
↪10.10.0/24'}]}]},
     'type': 'tunnel_interface'}]
```

classmethod create_dynamic(*name*, *interface_id*, *dynamic_index=1*, *reverse_connection=True*, *automatic_default_route=True*, *domain_server_address=None*, *loopback_ndi='127.0.0.1'*, *location_ref=None*, *log_server_ref=None*, *zone_ref=None*, *enable_gti=False*, *enable_antivirus=False*, *sidewinder_proxy_enabled=False*, *default_nat=False*, *comment=None*, ***kw*)

Create a single layer 3 firewall with only a single DHCP interface. Useful when creating virtualized FW's such as in Microsoft Azure.

Parameters

- **name** (*str*) – name of engine
- **interface_id** (*str, int*) – interface ID used for dynamic interface and management
- **reverse_connection** (*bool*) – specifies the dynamic interface should initiate connections to management (default: True)
- **automatic_default_route** (*bool*) – allow SMC to create a dynamic netlink for the default route (default: True)
- **domain_server_address** (*list*) – list of IP addresses for engine DNS
- **loopback_ndi** (*str*) – IP address for a loopback NDI. When creating a dynamic engine, the *auth_request* must be set to a different interface, so loopback is created
- **location_ref** (*str*) – location by name for the engine
- **log_server_ref** (*str*) – log server reference, will use the

13.6.3 Layer2Firewall

class smc.core.engines.**Layer2Firewall**(*name*, ***meta*)

Creates a Layer 2 Firewall with a default inline interface pair To instantiate and create, call 'create' classmethod as follows:

```
engine = Layer2Firewall.create(name='myinline',
                               mgmt_ip='1.1.1.1',
                               mgmt_network='1.1.1.0/24')
```

classmethod create(*name*, *mgmt_ip*, *mgmt_network*, *mgmt_interface=0*, *inline_interface='1-2'*, *logical_interface='default_eth'*, *log_server_ref=None*, *domain_server_address=None*, *zone_ref=None*, *enable_antivirus=False*, *enable_gti=False*, *comment=None*)

Create a single layer 2 firewall with management interface and inline pair

Parameters

- **name** (*str*) – name of firewall engine
- **mgmt_ip** (*str*) – ip address of management interface
- **mgmt_network** (*str*) – management network in cidr format
- **mgmt_interface** (*int*) – (optional) interface for management from SMC to fw
- **inline_interface** (*str*) – interfaces to use for first inline pair
- **logical_interface** (*str*) – name, str href or LogicalInterface (created if it doesn't exist)
- **log_server_ref** (*str*) – (optional) href to log_server instance
- **domain_server_address** (*list*) – (optional) DNS server addresses
- **zone_ref** (*str*) – zone name, str href or Zone for management interface (created if not found)
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI

Raises ***CreateEngineFailed*** – Failure to create with reason

Returns *smc.core.engine.Engine*

13.6.4 Layer3VirtualEngine

class smc.core.engines.**Layer3VirtualEngine**(*name*, ***meta*)

Create a layer3 virtual engine and map to specified Master Engine Each layer 3 virtual firewall will use the same virtual resource that should be pre-created.

To instantiate and create, call 'create' as follows:

```
engine = Layer3VirtualEngine.create(
                        name='myips',
                        master_engine='mymaster_engine',
                        virtual_engine='ve-3',
                        interfaces=[{'interface_id': 0,
                                     'address': '5.5.5.5',
                                     'network_value': '5.5.5.5/30',
                                     'zone_ref': ''}]
```

classmethod create(*name*, *master_engine*, *virtual_resource*, *interfaces*, *default_nat=False*, *outgoing_intf=0*, *domain_server_address=None*, *enable_ospf=False*, *ospf_profile=None*, *comment=None*, ***kw*)

Create a Layer3Virtual engine for a Master Engine. Provide interfaces as a list of dict items specifying the interface details in format:

```
{'interface_id': 1, 'address': '1.1.1.1', 'network_value': '1.1.1.0/24',
 'zone_ref': zone_by_name,href, 'comment': 'my interface comment'}
```

Parameters

- **name** (*str*) – Name of this layer 3 virtual engine
- **master_engine** (*str*) – Name of existing master engine
- **virtual_resource** (*str*) – name of pre-created virtual resource
- **interfaces** (*list*) – dict of interface details
- **default_nat** (*bool*) – Whether to enable default NAT for outbound
- **outgoing_intf** (*int*) – outgoing interface for VE. Specifies interface number
- **interfaces** – interfaces mappings passed in
- **enable_ospf** (*bool*) – whether to turn OSPF on within engine
- **ospf_profile** (*str*) – optional OSPF profile to use on engine, by ref

Raises

- ***CreateEngineFailed*** – Failure to create with reason
- ***LoadEngineFailed*** – master engine not found

Returns *smc.core.engine.Engine*

13.6.5 FirewallCluster

class smc.core.engines.**FirewallCluster**(*name*, ***meta*)

Firewall Cluster Creates a layer 3 firewall cluster engine with CVI and NDI's. Once engine is created, you can later add additional interfaces using the *engine.physical_interface* reference.

See also:

smc.core.physical_interface.add_layer3_cluster_interface()

classmethod create(*name*, *cluster_virtual*, *network_value*, *macaddress*, *interface_id*, *nodes*, *vlan_id=None*, *cluster_mode='balancing'*, *backup_mgt=None*, *primary_heartbeat=None*, *log_server_ref=None*, *domain_server_address=None*, *location_ref=None*, *zone_ref=None*, *default_nat=False*, *enable_antivirus=False*, *enable_gti=False*, *comment=None*, *snmp=None*, ***kw*)

Create a layer 3 firewall cluster with management interface and any number of nodes. If providing keyword arguments to create additional interfaces, use the same constructor arguments and pass an *interfaces* keyword argument. The constructor defined interface will be assigned as the primary management interface by default. Otherwise the engine will be created with a single interface and interfaces can be added after.

Changed in version 0.6.1: Chgnged *cluster_nic* to *interface_id*, and *cluster_mask* to *network_value*

Parameters

- **name** (*str*) – name of firewall engine
- **cluster_virtual** (*str*) – ip of cluster CVI
- **network_value** (*str*) – ip netmask of cluster CVI
- **macaddress** (*str*) – macaddress for packet dispatch clustering
- **interface_id** (*str*) – nic id to use for primary interface
- **nodes** (*list*) – address/network_value/nodeid combination for cluster nodes
- **vlan_id** (*str*) – optional VLAN id for the management interface, i.e. '15'.
- **cluster_mode** (*str*) – 'balancing' or 'standby' mode (default: balancing)
- **primary_heartbeat** (*str,int*) – optionally set the primary_heartbeat. This is automatically set to the management interface but can be overridden to use another interface if defining additional interfaces using *interfaces*.
- **backup_mgt** (*str,int*) – optionally set the backup management interface. This is unset unless you define additional interfaces using *interfaces*.
- **log_server_ref** (*str*) – (optional) href to log_server instance
- **domain_server_address** (*list*) – (optional) DNS server addresses
- **location_ref** (*str*) – location href or not for engine if needed to contact SMC behind NAT (created if not found)
- **zone_ref** (*str*) – zone name, str href or Zone for management interface (created if not found)
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI
- **interfaces** (*list*) – optional keyword to supply additional interfaces
- **snmp** (*dict*) – SNMP dict should have keys *snmp_agent* str defining name of SNMPAgent, *snmp_interface* which is a list of interface IDs, and optionally *snmp_location* which is a string with the SNMP location name.

Raises ***CreateEngineFailed*** – Failure to create with reason

Returns *smc.core.engine.Engine*

Example nodes parameter input:

```
[{'address':'5.5.5.2', 'network_value':'5.5.5.0/24', 'nodeid':1},
 {'address':'5.5.5.3', 'network_value':'5.5.5.0/24', 'nodeid':2},
 {'address':'5.5.5.4', 'network_value':'5.5.5.0/24', 'nodeid':3}]
```

You can also create additional CVI+NDI, or NDI only interfaces by providing the keyword argument interfaces using the same keyword values from the constructor:

```
interfaces=[
   {'interface_id': 1,
    'macaddress': '02:02:02:02:02:03',
    'interfaces': [{'cluster_virtual': '2.2.2.1',
                    'network_value': '2.2.2.0/24',
                    'nodes':[{'address': '2.2.2.2', 'network_value': '2.2.2.0/
↪24', 'nodeid': 1},
                             {'address': '2.2.2.3', 'network_value': '2.2.2.0/
↪24', 'nodeid': 2}]
                   }]
    },
   {'interface_id': 2,
    'interfaces': [{'nodes':[{'address': '3.3.3.2', 'network_value': '3.3.3.0/
↪24', 'nodeid': 1},
                             {'address': '3.3.3.3', 'network_value': '3.3.3.0/
↪24', 'nodeid': 2}]
                   }]
    }]
```

It is also possible to define VLAN interfaces by providing the *vlan_id* keyword. Example VLAN with NDI only interfaces. If nesting the zone_ref within the interfaces list, the zone will be applied to the VLAN versus the top level interface:

```
interfaces=[
   {'interface_id': 2,
    'interfaces': [{'nodes':[{'address': '3.3.3.2', 'network_value': '3.3.3.0/
↪24', 'nodeid': 1},
                             {'address': '3.3.3.3', 'network_value': '3.3.3.0/
↪24', 'nodeid': 2}],
                    'vlan_id': 22,
                    'zone_ref': 'private-network'
                   },
                   {'nodes': [{'address': '4.4.4.1', 'network_value': '4.4.4.0/
↪24', 'nodeid': 1},
                              {'address': '4.4.4.2', 'network_value': '4.4.4.0/
↪24', 'nodeid': 2}],
                    'vlan_id': 23,
                    'zone_ref': 'other_vlan'
                  }]
}]
```

Tunnel interfaces can also be created. As all interfaces defined are assumed to be a physical interface type, you must specify the *type* parameter to indicate the interface is a tunnel interface. Tunnel interfaces do not have a macaddress or VLANs. They be configured with NDI interfaces by omitting the *cluster_virtual* and *network_value* top level attributes:

```
interfaces=[
    {'interface_id': 1000,
     'interfaces': [{'cluster_virtual': '100.100.100.1',
                     'network_value': '100.100.100.0/24',
```

(continues on next page)

(continued from previous page)

```
                    'nodes':[{'address': '100.100.100.2', 'network_value':
↪'100.100.100.0/24', 'nodeid': 1},
                             {'address': '100.100.100.3', 'network_value':
↪'100.100.100.0/24', 'nodeid': 2}]
                   }],
    'zone_ref': 'AWStunnel',
    'type': 'tunnel_interface'
   }]
```

If setting primary_heartbeat or backup_mgt to a specific interface (the primary interface configured in the constructor will have these roles by default), you must define the interfaces in the *interfaces* keyword argument list.

Note: If creating additional interfaces, you must at minimum provide the *interface_id* and *nodes* to create an NDI only interface.

---

classmethod create_bulk(*name*, *interfaces=None*, *nodes=2*, *cluster_mode='balancing'*, *primary_mgt=None*, *backup_mgt=None*, *primary_heartbeat=None*, *log_server_ref=None*, *domain_server_address=None*, *location_ref=None*, *default_nat=False*, *enable_antivirus=False*, *enable_gti=False*, *comment=None*, *snmp=None*, ***kw*)

Parameters **snmp** (*dict*) – SNMP dict should have keys *snmp_agent* str defining name of SNMPAgent, *snmp_interface* which is a list of interface IDs, and optionally *snmp_location* which is a string with the SNMP location name.

13.6.6 MasterEngine

class smc.core.engines.**MasterEngine**(*name*, ***meta*)

Creates a master engine in a firewall role. Layer3VirtualEngine should be used to add each individual instance to the Master Engine.

classmethod create(*name*, *master_type*, *mgmt_ip*, *mgmt_network*, *mgmt_interface=0*, *log_server_ref=None*, *zone_ref=None*, *domain_server_address=None*, *enable_gti=False*, *enable_antivirus=False*, *comment=None*)

Create a Master Engine with management interface

Parameters

- **name** (*str*) – name of master engine engine
- **master_type** (*str*) – firewalll
- **mgmt_ip** (*str*) – ip address for management interface
- **mgmt_network** (*str*) – full netmask for management
- **mgmt_interface** (*str*) – interface to use for mgmt (default: 0)
- **log_server_ref** (*str*) – (optional) href to log_server instance
- **domain_server_address** (*list*) – (optional) DNS server addresses
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI

Raises ***CreateEngineFailed*** – Failure to create with reason

Returns *smc.core.engine.Engine*

13.6.7 MasterEngineCluster

class smc.core.engines.**MasterEngineCluster**(*name*, ***meta*)

Master Engine Cluster Clusters are currently supported in an active/standby configuration only.

classmethod create(*name*, *master_type*, *macaddress*, *nodes*, *mgmt_interface=0*, *log_server_ref=None*, *domain_server_address=None*, *enable_gti=False*, *enable_antivirus=False*, *comment=None*, ***kw*)

Create Master Engine Cluster

Parameters

- **name** (*str*) – name of master engine engine
- **master_type** (*str*) – firewalll
- **mgmt_ip** (*str*) – ip address for management interface
- **mgmt_netmask** (*str*) – full netmask for management
- **mgmt_interface** (*str*) – interface to use for mgmt (default: 0)
- **nodes** (*list*) – address/network_value/nodeid combination for cluster nodes
- **log_server_ref** (*str*) – (optional) href to log_server instance
- **domain_server_address** (*list*) – (optional) DNS server addresses
- **enable_antivirus** (*bool*) – (optional) Enable antivirus (required DNS)
- **enable_gti** (*bool*) – (optional) Enable GTI

Raises ***CreateEngineFailed*** – Failure to create with reason

Returns *smc.core.engine.Engine*

Example nodes parameter input:

```
[{'address':'5.5.5.2',
  'network_value':'5.5.5.0/24',
  'nodeid':1},
 {'address':'5.5.5.3',
  'network_value':'5.5.5.0/24',
  'nodeid':2},
 {'address':'5.5.5.4',
  'network_value':'5.5.5.0/24',
  'nodeid':3}]
```

13.7 Dynamic Routing Elements

13.7.1 RouteMap

Route map rules and match condition elements for dynamic routing policies.

A RouteMap can be created and subsequent rules can be inserted within the route map policy.

A MatchCondition is the subject of the rule providing criteria to specify how a match is made. Elements used in match conditions are *next_hop*, *peer_address*, *access_list* and type *metric*.

See also:

MatchCondition for more details on how to add match conditions to a rule or modify an existing rule.

Example of creating a RouteMap and subsequent rule, specifying match condition options as keyword arguments:

```
>>> from smc.routing.route_map import RouteMap
>>> from smc.routing.access_list import IPAccessList
>>> from smc.routing.bgp import ExternalBGPPeer
...
>>> rm = RouteMap.create(name='myroutemap')
>>> rm
RouteMap(name=myroutemap)
>>> rm.route_map_rules.create(name='rule1', action='permit',
        next_hop=IPAccessList('myacl'), peer_address=ExternalBGPPeer('bgppeer'),
        metric=20)
RouteMapRule(name=rule1)
...
>>> rule1 = rm.route_map_rules.get(0) # retrieve rule 1 from the route map
>>> for condition in rule1.match_condition:
...   condition
...
Condition(rank=1, element=ExternalBGPPeer(name=bgppeer), type=u'peer_address')
Condition(rank=2, element=IPAccessList(name=myacl), type='access_list')
Condition(rank=3, element=Metric(value=20), type=u'metric')
```

Instead of providing singular match condition keywords to the *create* constructor, you can also optionally provide a MatchCondition instance when creating a rule:

```
>>> from smc.routing.route_map import MatchCondition
>>> condition = MatchCondition()
>>> condition.add_access_list(IPAccessList('myacl'))
>>> condition.add_peer_address(ExternalBGPPeer('bgppeer'))
>>> condition.add_metric(20)
>>> condition
MatchCondition(entries=3)
>>> rm.route_map_rules.create(
...         name='foo2',
...         finish=False,
...         match_condition=condition)
RouteMapRule(name=foo2)
```

To remove a match condition, first obtain it's rank. After making the modification be sure to call update on the rule element:

```
>>> rule = rm.route_map_rules.get(0)
>>> rule.match_condition.remove_condition(rank=2)
>>> rule.update()
```

You can also delete a rule by obtaining the rule, either through the route_map_rules collection reference or by iteration:

```
rule = rm.route_map_rules.get(1)
rule.delete()
```

Or by the name:

```
rule = rm.route_map_rules.get_exact('foo')
rule.delete()
```

See also:

smc.base.collection.rule_collection

class smc.routing.route_map.**MatchCondition**(*rule=None*)
Bases: object

MatchCondition is an iterable container class that holds the match conditions for the route map rule. The list of conditions are ranked in order. You can add, remove and view conditions currently configured in this rule. After making modifications, call update on the rule to commit back to SMC.

When iterating over a match condition, a namedtuple is returned that provides the rank and element type for the condition. It is then possible to add by rank (ie: insert conditions in between others), or remove based on rank. If not rank is provided when adding new conditions, the condition is added to the bottom of the rank list.

Return type list(*Condition*)

add_access_list(*accesslist*, *rank=None*)
Add an access list to the match condition. Valid access list types are IPAccessList (v4 and v6), IPPrefixList (v4 and v6), AS Path, CommunityAccessList, ExtendedCommunityAccessList.

add_metric(*value*, *rank=None*)
Add a metric to this match condition

Parameters **value** (*int*) – metric value

add_next_hop(*access_or_prefix_list*, *rank=None*)
Add a next hop condition. Next hop elements must be of type IPAccessList or IPPrefixList.

Raises ***ElementNotFound*** – If element specified does not exist

add_peer_address(*ext_bgp_peer_or_fw*, *rank=None*)
Add a peer address. Peer address types are ExternalBGPPeer or Engine.

Raises ***ElementNotFound*** – If element specified does not exist

remove_condition(*rank*)
Remove a condition element using it's rank. You can find the rank and element for a match condition by iterating the match condition:

```
>>> rule1 = rm.route_map_rules.get(0)
>>> for condition in rule1.match_condition:
...   condition
...
Condition(rank=1, element=ExternalBGPPeer(name=bgppeer))
Condition(rank=2, element=IPAccessList(name=myacl))
Condition(rank=3, element=Metric(value=20))
```

Then delete by rank. Call update on the rule after making the modification.

Parameters **rank** (*int*) – rank of the condition to remove

Raises ***UpdateElementFailed*** – failed to update rule

Returns None

class smc.routing.route_map.**RouteMap**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Use Route Map elements in more complex networks to control or manipulate routes. You can use Access List elements as a Matching Condition in a Route Map rule. RouteMaps are rule lists similar to normal policies and can be iterated:

```
>>> from smc.routing.route_map import RouteMap
>>> rm = RouteMap('myroutemap')
>>> for rule in rm.route_map_rules:
...   rule
...
RouteMapRule(name=Rule @115.13)
RouteMapRule(name=Rule @117.0)
```

classmethod create(*name*, *comment=None*)
Create a new route map. After creation, you can add a rule and subsequent match conditions.

Parameters

- **name** (*str*) – name of route map
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed creating route map

Return type *RouteMap*

route_map_rules
IPv6NAT Rule entry point

Return type rule_collection(*IPv6NATRule*)

search_rule(*search*)
Search the RouteMap policy using a search string

Parameters **search** (*str*) – search string for a contains match against the rule name and comments field

Return type list(*RouteMapRule*)

class smc.routing.route_map.**RouteMapRule**(**meta*)
Bases: smc.policy.rule.RuleCommon, *smc.base.model.SubElement*

A route map rule represents the rules to be processed for a route map assigned to a specific BGP network. A match condition can be provided which encapsulates using dynamic routing element types such as IPAccessList, IPPrefixList, etc.

action
Action for this route map rule. Valid actions are 'permit' and 'deny'.

Return type str

call_route_map(*route_map*)
Call another route map after match of this rule. Call update on the rule to save after modification.

Parameters **route_map** (RouteMap) – Pass the route map element

Raises ***ElementNotFound*** – invalid RouteMap reference passed

Returns None

comment
Get and set the comment for this rule.

Parameters **value** (*str*) – string comment

Return type str

create(*name*, *action='permit'*, *goto=None*, *finish=False*, *call=None*, *comment=None*, *add_pos=None*, *after=None*, *before=None*, ***match_condition*)

Create a route map rule. You can provide match conditions by using keyword arguments specifying the required types. You can also create the route map rule and add match conditions after.

Parameters

- **name** (*str*) – name for this rule
- **action** (*str*) – permit or deny
- **goto** (*str*) – specify a rule section to goto after if there is a match condition. This will override the finish parameter
- **finish** (*bool*) – finish stops the processing after a match condition. If finish is False, processing will continue to the next rule.
- **call** (RouteMap) – call another route map after matching.
- **comment** (*str*) – optional comment for the rule
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.
- **match_condition** – keyword values identifying initial values for the match condition. Valid keyword arguments are 'access_list', 'next_hop', 'metric' and 'peer_address'. You can also optionally pass the keyword 'match_condition' with an instance of MatchCondition.

Raises

- ***CreateRuleFailed*** – failure to insert rule with reason
- ***ElementNotFound*** – if references elements in a match condition this can be raised when the element specified is not found.

See also:

MatchCondition for valid elements and expected values for each type.

finish

Is rule action goto set to finish on this rule match. If finish is False, then the policy will proceed to the next rule.

Return type bool

goto

If the rule is set to goto a rule section, return the rule section, otherwise it will return None. Check the value of finish to determine if the rule is set to finish on match.

Returns RouteMap or None

goto_rule_section(*rule_section*)

Set this rule to goto a specific rule section after match. If goto is None, then check value of finish.

Parameters **rule_section** (RouteMapRule) – pass rule section

Returns None

is_disabled
Is the rule disabled

Return type bool

match_condition
Return the match condition for this rule. This can then be modified in place. Be sure to call update on the rule to save.

Return type *MatchCondition*

class smc.routing.route_map.**Metric**(*value*)
Bases: tuple

A metric is a simple namedtuple for returning a Metric route map element

Variables ***value*** (*int*) – metric value for this BGP route

class smc.routing.route_map.**Condition**(*rank*, *element*, *type*)
Bases: tuple

A condition defines the type of dynamic element that is used in the match condition field of a route map.

Variables

- **rank** (*str*) – the rank in the match condition list
- **element** (*str*) – the dynamic element type for this condition
- ***type*** (*str*) – type defines the type of entry, i.e. metric, peer_address, next_hop, access_list

13.7.2 IPAccessList

AccessList module represents functionality that support dynamic routing filters based on IPv4 or IPv6 access lists such as OSPF and BGP.

class smc.routing.access_list.**AccessList**
Bases: object

AccessList provides an iterable container that allows simple iteration over existing IPAccessList (v4 and v6), IPPrefixList (v4 and v6), CommunityAccessList and ExtendedCommunityAccessList entries. When using the *create* constructor, validate the keyword arguments based on the specific access list requirements.

Returns namedtuple based on access list type

add_entry(***kw*)
Add an entry to an AccessList. Use the supported arguments for the inheriting class for keyword arguments.

Raises ***UpdateElementFailed*** – failure to modify with reason

Returns None

classmethod create(*name*, *entries=None*, *comment=None*, ***kw*)
Create an Access List Entry.

Depending on the access list type you are creating (IPAccessList, IPv6AccessList, IPPrefixList, IPv6PrefixList, CommunityAccessList, ExtendedCommunityAccessList), entries will define a dict of the valid attributes for that ACL type. Each class has a defined list of attributes documented in it's class.

You can optionally leave entries blank and use the *add_entry()* method after creating the list container.

Parameters

- **name** (*str*) – name of IP Access List

- **entries** (*list*) – access control entry
- **kw** – optional keywords that might be necessary to create the ACL (see specific Access Control List documentation for options)

Raises ***CreateElementFailed*** – cannot create element

Returns The access list based on type

remove_entry (***field_value*)
Remove an AccessList entry by field specified. Use the supported arguments for the inheriting class for keyword arguments.

Raises ***UpdateElementFailed*** – failed to modify with reason

Returns None

classmethod update_or_create (*with_status=False*, *overwrite_existing=False*, ***kw*)
Update or create the Access List. This method will not attempt to evaluate whether the access list has differences, instead it will update with the contents of the payload entirely. If the intent is to only add or remove a single entry, use *~add_entry* and *~remove_entry* methods.

Parameters

- **with_status** (*bool*) – return with 3-tuple of (Element, modified, created) holding status
- **overwrite_existing** (*bool*) – if the access list exists but instead of an incremental update you want to overwrite with the newly defined entries, set this to True (default: False)

Returns Element or 3-tuple with element and status

class smc.routing.access_list.**IPAccessList** (*name*, ***meta*)
Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

IPAccessList is used by dynamic routing protocols to allow filtering of routes. Protocols like OSPF and BGP allow inbound and outbound filters using these.

Create an IPAccessList. When providing values for *entries* to the create constructor, use valid attributes as defined in *AccessListEntry*:

```
>>> ip = IPAccessList.create(name='mylist', entries=[
    {'subnet': '1.1.1.0/24', 'action': 'permit'}, {'subnet': '2.2.2.0/24', 'action
↪': 'deny'}])
...
>>> ip.add_entry(subnet='3.3.3.0/24', action='permit')
>>> ip.remove_entry(subnet='1.1.1.0/24')
>>> ip.update()
'https://172.18.1.151:8082/6.4/elements/ip_access_list/13'
>>> for entry in ip:
...   entry
...
AccessListEntry(subnet=u'2.2.2.0/24', action=u'deny', comment=None)
AccessListEntry(subnet=u'3.3.3.0/24', action=u'permit', comment=None)
...
>>> ip.delete()
```

This is an iterable container yielding *AccessListEntry*

See also:

AccessListEntry for valid *create* and add/remove parameters

class smc.routing.access_list.**IPv6AccessList**(*name*, ***meta*)
Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

IPv6AccessList is used by dynamic routing protocols to allow filtering of routes. Protocols like OSPF and BGP allow inbound and outbound filters using these.

```
>>> acl6 = IPv6AccessList.create(name='aclv6', entries=[
...           {'subnet': '2001:db8:1::1/128', 'action': 'permit'}])
>>> acl6
IPv6AccessList(name=aclv6)
>>> for entry in acl6:
...   entry
...
AccessListEntry(subnet=u'2001:db8:1::1/128', action=u'permit', comment=None)
```

This is an iterable container yielding *AccessListEntry*

See also:

IPAccessList for using this element.

class smc.routing.access_list.**AccessListEntry**(*subnet*, *action*, *comment*)
Bases: tuple

An AccessListEntry defines a simple entry for an IPAccessList used in dynamic routing configurations.

Variables

- **subnet** (*str*) – subnet associated with this entry
- ***action*** (*str*) – action for the entry
- ***comment*** (*str*) – optional comment for the entry

13.7.3 IPPrefixList

IP Prefix module represents prefix lists that can be used to filter networks for OSPF routing.

class smc.routing.prefix_list.**IPPrefixList**(*name*, ***meta*)
Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

An IP prefix list specifies a list of networks. When you apply an IP prefix list to a neighbor, the device sends or receives only a route whose destination is in the IP prefix list.

Creating and modifying an IPAccessList is similar to other access list methods:

```
>>> prefix = IPPrefixList.create(name='mylist', entries=[
...   {'subnet': '10.0.0.0/8', 'min_prefix_length': 16, 'max_prefix_length': 32,
↪'action': 'deny'},
...   {'subnet': '192.16.1.0/24', 'min_prefix_length': 25, 'max_prefix_length':␣
↪32, 'action': 'permit'}])
>>> prefix
IPPrefixList(name=mylist)
...
>>> prefix.add_entry(subnet='192.17.1.0/24', min_prefix_length=25, max_prefix_
↪length=32, action='deny')
>>> prefix.update()
'https://172.18.1.151:8082/6.4/elements/ip_prefix_list/16'
>>> prefix.remove_entry(subnet='192.16.1.0/24')
>>> prefix.update()
'https://172.18.1.151:8082/6.4/elements/ip_prefix_list/16'
```

(continues on next page)

(continued from previous page)

```
>>> for entry in prefix:
...   entry
...
PrefixListEntry(subnet=u'10.0.0.0/8', action=u'deny', min_prefix_length=16, max_
↪prefix_length=32, comment=None)
PrefixListEntry(subnet=u'192.17.1.0/24', action=u'deny', min_prefix_length=25,␣
↪max_prefix_length=32, comment=None)
```

You can also create a PrefixList without using the min_prefix_length and max_prefix_length fields:

```
>>> prefix = IPPrefixList.create(name='mylist', entries=[
...   {'subnet': '10.0.0.0/8', 'action': 'deny'},
...   {'subnet': '192.16.1.0/24', 'action': 'permit'}])
```

This is an iterable container yielding *PrefixListEntry*

See also:

PrefixListEntry for valid *create* and add/remove parameters

class smc.routing.prefix_list.**IPv6PrefixList**(*name*, ***meta*)

Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

An IP prefix list specifies a list of networks. When you apply an IP prefix list to a neighbor, the device sends or receives only a route whose destination is in the IP prefix list.

```
>>> prefix6 = IPv6PrefixList.create(name='myipv6', entries=[
...   {'subnet': 'ab00::/64', 'min_prefix_length': 65, 'max_prefix_length': 128,
↪'action': 'deny'}])
>>> prefix6
IPv6PrefixList(name=myipv6)
>>> for entry in prefix6:
...   entry
...
PrefixListEntry(subnet=u'ab00::/64', action=u'deny', min_prefix_length=65, max_
↪prefix_length=128, comment=None)
```

You can also create a PrefixList without using the min_prefix_length and max_prefix_length fields:

```
>>> prefix = IPPrefixList.create(name='mylist', entries=[
...   {'subnet': 'ab00::/64', 'action': 'deny'}
```

This is an iterable container yielding *PrefixListEntry*

See also:

IPPrefixList for other common operations

class smc.routing.prefix_list.**PrefixListEntry**(*subnet*, *action*, *min_prefix_length*, *max_prefix_length*, *comment*)

Bases: `tuple`

A PrefixListEntry defines a simple entry for an PrefixList used in dynamic routing configurations.

Variables

- **subnet** (*str*) – subnet associated with this entry
- ***action*** (*str*) – action for the entry
- **min_prefix_length** (*int*) – minimum mask bits

- **max_prefix_length** (*int*) – maximum mask bits
- ***comment*** (*str*) – optional comment for the entry

13.7.4 BGP Elements

BGP Module representing BGP settings for Stonesoft NGFW layer 3 engines. BGP can be enabled and run on either single/cluster layer 3 firewalls or virtual FW's.

For adding BGP configurations, several steps are required:

- Enable BGP on the engine and specify the BGP Profile
- Create or use an existing OSPFArea to be used
- Modify the routing interface and add the BGP Peering

Enable BGP on an existing engine using the default BGP system profile:

```
engine.bgp.enable(
    autonomous_system=AutonomousSystem('myAS')
    announced_networks=[Network('172.18.1.0/24'), Network('1.1.1.0/24')])
```

Create a BGP Peering using the default BGP Connection Profile:

```
BGPPeering.create(name='mypeer')
```

Add the BGP Peering to the routing interface:

```
interface = engine.routing.get(0)
interface.add_bgp_peering(
    BGPPeering('mypeer'),
    ExternalBGPPeer('neighbor'))
```

Disable BGP on an engine:

```
engine.bgp.disable()
```

Finding profiles or elements can also be done through collections:

```
>>> list(BGPProfile.objects.all())
[BGPProfile(name=Default BGP Profile)]

>>> list(ExternalBGPPeer.objects.all())
[ExternalBGPPeer(name=bgp-02), ExternalBGPPeer(name=Amazon AWS),␣
↪ExternalBGPPeer(name=bgp-01)]
```

The BGP relationship can be represented as:

```
Engine --uses an--> (BGP Profile --and--> Autonomous System --and--> Announced␣
↪Networks)
Engine Routing --uses-an--> BGP Peering --has-a--> External BGP Peer
```

Only Layer3Firewall and Layer3VirtualEngine types can support running BGP.

See also:

smc.core.engines.Layer3Firewall and *smc.core.engines.Layer3VirtualEngine*

class smc.routing.bgp.**BGP**(*data=None*)

BGP represents the BGP configuration on a given engine. An instance is returned from an engine reference:

```
engine = Engine('myengine')
engine.dynamic_routing.bgp.status
engine.dynamic_routing.bgp.announced_networks
...
```

When making changes to the BGP configuration, any methods called that change the configuration also require that engine.update() is called once changes are complete. This way you can make multiple changes without refreshing the engine cache.

For example, adding advertised networks to the configuration:

```
engine.dynamic_routing.bgp.update_configuration(announced_network=[Network('foo
↪')])
engine.update()
```

Variables

- **autonomous_system** (AutonomousSystem) – AS reference for this BGP configuration
- **profile** (BGPProfile) – BGP profile reference for this configuration

announced_networks

Show all announced networks for the BGP configuration. Returns tuple of advertised network, routemap. Route map may be None.

```
for advertised in engine.bgp.advertisements:
    net, route_map = advertised
```

Returns list of tuples (advertised_network, route_map).

disable()

Disable BGP on this engine.

Returns None

enable(*autonomous_system*, *announced_networks*, *router_id=None*, *bgp_profile=None*)

Enable BGP on this engine. On master engine, enable BGP on the virtual firewall. When adding networks to *announced_networks*, the element types can be of type *smc.elements.network.Host*, *smc.elements.network.Network* or *smc.elements.group.Group*. If passing a Group, it must have element types of host or network.

Within announced_networks, you can pass a 2-tuple that provides an optional *smc.routing.route_map.RouteMap* if additional policy is required for a given network.

```
engine.dynamic_routing.bgp.enable(
    autonomous_system=AutonomousSystem('aws_as'),
    announced_networks=[Network('bgpnet'),Network('inside')],
    router_id='10.10.10.10')
```

Parameters

- **autonomous_system** (*str*,AutonomousSystem) – provide the AS element or str href for the element

- **bgp_profile** (*str,BGPProfile*) – provide the BGPProfile element or str href for the element; if None, use system default
- **announced_networks** (*list*) – list of networks to advertise via BGP Announced networks can be single networks,host or group elements or a 2-tuple with the second tuple item being a routemap element
- **router_id** (*str*) – router id for BGP, should be an IP address. If not set, automatic discovery will use default bound interface as ID.

Raises ***ElementNotFound*** – OSPF, AS or Networks not found

Returns None

Note: For arguments that take str or Element, the str value should be the href of the element.

---

router_id
Get the router ID for this BGP configuration. If None, then the ID will use the interface IP.

Returns str or None

status
Is BGP enabled on this engine.

Return type bool

update_configuration(***kwargs*)
Update configuration using valid kwargs as defined in the enable constructor.

Parameters **kwargs** (*dict*) – kwargs to satisfy valid args from *enable*

Return type bool

13.7.4.1 AutonomousSystem

class smc.routing.bgp.**AutonomousSystem**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Autonomous System for BGP routing. AS is a required setting when enabling BGP on an engine and specifies a unique identifier for routing communications.

as_number
The AS Number for this autonomous system

Returns AS number

Return type int

classmethod create(*name*, *as_number*, *comment=None*)
Create an AS to be applied on the engine BGP configuration. An AS is a required parameter when creating an ExternalBGPPeer. You can also provide an AS number using an 'asdot' syntax:

```
AutonomousSystem.create(name='myas', as_number='200.600')
```

Parameters

- **name** (*str*) – name of this AS
- **as_number** (*int*) – AS number preferred

- **comment** (*str*) – optional string comment

Raises

- ***CreateElementFailed*** – unable to create AS
- **ValueError** – If providing AS number in dotted format and low/high order bytes are > 65535.

Returns instance with meta

Return type *AutonomousSystem*

classmethod update_or_create(*with_status=False*, ***kwargs*)

Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to `get()`. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

13.7.4.2 ExternalBGPPeer

class smc.routing.bgp.**ExternalBGPPeer**(*name*, ***meta*)

Bases: *smc.base.model.Element*

An External BGP represents the AS and IP settings for a remote BGP peer. Creating a BGP peer requires that you also pre-create an *AutonomousSystem* element:

```
AutonomousSystem.create(name='neighborA', as_number=500)
ExternalBGPPeer.create(name='name',
                       neighbor_as_ref=AutonomousSystem('neighborA'),
                       neighbor_ip='1.1.1.1')
```

Variables **neighbor_as** (AutonomousSystem) – AS for this external BGP peer

classmethod create(*name*, *neighbor_as*, *neighbor_ip*, *neighbor_port=179*, *comment=None*)
Create an external BGP Peer.

Parameters

- **name** (*str*) – name of peer
- **neighbor_as_ref** (*str*,AutonomousSystem) – AutonomousSystem element or href.
- **neighbor_ip** (*str*) – ip address of BGP peer
- **neighbor_port** (*int*) – port for BGP, default 179.

Raises ***CreateElementFailed*** – failed creating

Returns instance with meta

Return type *ExternalBGPPeer*

neighbor_ip
IP address of the external BGP Peer

Returns ipaddress of external bgp peer

Return type str

neighbor_port
Port used for neighbor AS

Returns neighbor port

Return type int

13.7.4.3 BGPPeering

class smc.routing.bgp.**BGPPeering**(*name*, ***meta*)
Bases: *smc.base.model.Element*

BGP Peering is applied directly to an interface and defines basic connection settings. A BGPConnectionProfile is required to create a BGPPeering and if not provided, the default profile will be used.

The most basic peering can simply specify the name of the peering and leverage the default BGPConnection-Profile:

```
BGPPeering.create(name='my-aws-peer')
```

Variables **connection_profile** (BGPConnectionProfile) – BGP connection profile for this peering

classmethod create(*name*, *connection_profile_ref=None*, *md5_password=None*, *local_as_option='not_set'*, *max_prefix_option='not_enabled'*, *send_community='no'*, *connected_check='disabled'*, *orf_option='disabled'*, *next_hop_self=True*, *override_capability=False*, *dont_capability_negotiate=False*, *remote_private_as=False*, *route_reflector_client=False*, *soft_reconfiguration=True*, *ttl_option='disabled'*, *comment=None*)

Create a new BGPPeering configuration.

Parameters

- **name** (*str*) – name of peering
- **connection_profile_ref** (*str,BGPConnectionProfile*) – required BGP connection profile. System default used if not provided.
- **md5_password** (*str*) – optional md5_password
- **local_as_option** (*str*) – the local AS mode. Valid options are: 'not_set', 'prepend', 'no_prepend', 'replace_as'
- **max_prefix_option** (*str*) – The max prefix mode. Valid options are: 'not_enabled', 'enabled', 'warning_only'
- **send_community** (*str*) – the send community mode. Valid options are: 'no', 'standard', 'extended', 'standard_and_extended'
- **connected_check** (*str*) – the connected check mode. Valid options are: 'disabled', 'enabled', 'automatic'
- **orf_option** (*str*) – outbound route filtering mode. Valid options are: 'disabled', 'send', 'receive', 'both'
- **next_hop_self** (*bool*) – next hop self setting
- **override_capability** (*bool*) – is override received capabilities
- **dont_capability_negotiate** (*bool*) – do not send capabilities
- **remote_private_as** (*bool*) – is remote a private AS
- **route_reflector_client** (*bool*) – Route Reflector Client (iBGP only)
- **soft_reconfiguration** (*bool*) – do soft reconfiguration inbound
- **ttl_option** (*str*) – ttl check mode. Valid options are: 'disabled', 'ttl-security'

Raises ***CreateElementFailed*** – failed creating profile

Returns instance with meta

Return type *BGPPeering*

13.7.4.4 BGPProfile

class smc.routing.bgp.**BGPProfile**(*name*, ***meta*)

Bases: *smc.base.model.Element*

A BGP Profile specifies settings specific to an engine level BGP configuration. A profile specifies engine specific settings such as distance, redistribution, and aggregation and port.

These settings are always in effect:

- BGP version 4/4+

- No autosummary
- No synchronization
- Graceful restart

Example of creating a custom BGP Profile with default administrative distances and custom subnet distances:

```
Network.create(name='inside', ipv4_network='1.1.1.0/24')
BGPProfile.create(
    name='bar',
    internal_distance=100,
    external_distance=200,
    local_distance=50,
    subnet_distance=[(Network('inside'), 100)])
```

classmethod create(*name*, *port=179*, *external_distance=20*, *internal_distance=200*, *local_distance=200*, *subnet_distance=None*)

Create a custom BGP Profile

Parameters

- **name** (*str*) – name of profile
- **port** (*int*) – port for BGP process
- **external_distance** (*int*) – external administrative distance; (1-255)
- **internal_distance** (*int*) – internal administrative distance (1-255)
- **local_distance** (*int*) – local administrative distance (aggregation) (1-255)
- **subnet_distance** (*list*) – configure specific subnet's with respective distances

Raises ***CreateElementFailed*** – reason for failure

Returns instance with meta

Return type *BGPProfile*

external_distance

External administrative distance (eBGP)

Returns distance setting

Return type int

internal_distance

Internal administrative distance (iBGP)

Returns internal distance setting

Return type int

local_distance

Local administrative distance (aggregation)

Returns local distance setting

Return type int

port

Specified port for BGP

Returns value of BGP port

Return type int

subnet_distance
Specific subnet administrative distances

Returns list of tuple (subnet, distance)

13.7.4.5 BGPConnectionProfile

class smc.routing.bgp.**BGPConnectionProfile**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A BGP Connection Profile will specify timer based settings and is used by a BGPPeering configuration.

Create a custom profile:

```
BGPConnectionProfile.create(
    name='fooprofile',
    md5_password='12345',
    connect_retry=200,
    session_hold_timer=100,
    session_keep_alive=150)
```

connect_retry
The connect retry timer, in seconds

Returns connect retry in seconds

Return type int

classmethod create(*name*, *md5_password=None*, *connect_retry=120*, *session_hold_timer=180*, *session_keep_alive=60*)
Create a new BGP Connection Profile.

Parameters

- **name** (*str*) – name of profile
- **md5_password** (*str*) – optional md5 password
- **connect_retry** (*int*) – The connect retry timer, in seconds
- **session_hold_timer** (*int*) – The session hold timer, in seconds
- **session_keep_alive** (*int*) – The session keep alive timer, in seconds

Raises ***CreateElementFailed*** – failed creating profile

Returns instance with meta

Return type *BGPConnectionProfile*

session_hold_timer
The session hold timer, in seconds

Returns in seconds

Return type int

session_keep_alive
The session keep alive, in seconds

Returns in seconds

Return type int

13.7.4.6 ASPathAccessList

class smc.routing.bgp_access_list.**ASPathAccessList**(*name*, ***meta*)
Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

An AS path is the autonomous systems that routing information passed through to get to a specified router. It indicates the origin of this route. The AS path is used to prevent routing loops in BGP.

ASPathAccessLists can be used as a MatchCondition in a RouteMap:

```
>>> aspath = ASPathAccessList.create(name='aspath', entries=[
...   {'expression': '123-456', 'action': 'permit'},
...   {'expression': '1234-567', 'action': 'deny'}])
>>> aspath
ASPathAccessList(name=aspath)
>>> aspath.add_entry(expression='897', action='permit')
>>> aspath.update()
'https://172.18.1.151:8082/6.4/elements/as_path_access_list/28'
...
>>> aspath.remove_entry(expression='123-456')
>>> aspath.update()
'https://172.18.1.151:8082/6.4/elements/as_path_access_list/28'
>>> for entry in aspath:
...   entry
...
ASPathListEntry(expression=u'1234-567', action=u'deny', comment=None)
ASPathListEntry(expression=u'897', action=u'permit', comment=None)
```

This is an iterable container yielding *ASPathListEntry*.

See also:

ASPathListEntry for valid *create* and add/remove parameters

class smc.routing.bgp_access_list.**ASPathListEntry**(*expression*, *action*, *comment*)
Bases: tuple

The ASPathAccessList is an iterable container and will return instances of *ASPathListEntry*.

Variables

- **expression** (*str*) – string expression identifying the AS path
- ***action*** (*str*) – 'permit' or 'deny'
- ***comment*** (*str*) – optional comment

13.7.4.7 CommunityAccessList

class smc.routing.bgp_access_list.**CommunityAccessList**(*name*, ***meta*)
Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

A CommunityAccessList is used to provide specific rules for BGP configurations providing and permit/deny capability based on the community defined. CommunityAccessLists can be used in a RouteMap match condition to refine the policy for a specific announced network.

When creating a new community ACL, *entries* is expecting a list of dict items using the valid field and values of this class. For example:

```
>>> from smc.routing.community_list import CommunityAccessList
>>> comm = CommunityAccessList.create(name='commacl',
        entries=[{'community': 123, 'action': 'permit'},{'community': 456, 'action
↪': 'deny'}],
        type='standard')
>>> comm
CommunityAccessList(name=commacl)
```

You can optionally also create an empty access list and use `add_entry()` to insert entries after:

```
>>> comm.add_entry(community=789, action='permit')
>>> comm.update()
```

Iterating the access list will return *CommunityListEntry*:

```
>>> for entries in comm:
...   entries
...
CommunityListEntry(community=u'789', action=u'permit', comment=None)
CommunityListEntry(community=u'456', action=u'deny', comment=None)
CommunityListEntry(community=u'123', action=u'permit', comment=None)
```

The *type* parameter for the *create* constructor can have values *standard* or *expanded*. If using *expanded*, the access list can then use a regex for matching the community string.

This is an iterable container yielding *CommunityListEntry*.

See also:

CommunityListEntry for valid *create* and add/remove parameters

Variables ***type*** (*str*) – 'standard' or 'expanded' (specify as kw when in *create* constructor when creating the top level access list.

class smc.routing.bgp_access_list.**CommunityListEntry**(*community*, *action*, *comment*)

Bases: `tuple`

The CommunityAccessList represents the entries for the community access lists.

Variables

- **community** (*str*) – community id
- ***action*** (*str*) – 'permit' or 'deny'
- ***comment*** (*str*) – optional comment

13.7.4.8 ExtendedCommunityAccessList

class smc.routing.bgp_access_list.**ExtendedCommunityAccessList**(*name*, ***meta*)

Bases: *smc.routing.access_list.AccessList*, *smc.base.model.Element*

Extended community access lists with the ability to specify route target or start of origin for an entry.

ExtendedCommunityAccessLists can be used in a RouteMap match condition to refine the policy for a specific announced network:

```
>>> comm = ExtendedCommunityAccessList.create(name='comm', entries=[
...   {'community': 123, 'action': 'permit', 'type': 'rt'},
...   {'community': 456, 'action': 'deny', 'type': 'soo'}],
...   type='standard')
>>> comm
ExtendedCommunityAccessList(name=comm)
>>> comm.add_entry(community=789, action='permit', type='rt')
>>> comm.update()
...
>>> comm.remove_entry(community=123)
>>> comm.update()
'https://172.18.1.151:8082/6.4/elements/extended_community_access_list/25'
>>> for entry in comm:
...   entry
...
ExtCommunityListEntry(community=u'456', action=u'deny', comment=None, type=u
↪'soo')
ExtCommunityListEntry(community=u'789', action=u'permit', comment=None, type=u
↪'rt')
```

This is an iterable container yielding *ExtCommunityListEntry*.

See also:

ExtCommunityListEntry for valid *create* and add/remove parameters

Variables ***type*** (*str*) – 'standard' or 'expanded' (specify as kw when in *create* constructor when creating the top level access list.

class smc.routing.bgp_access_list.**ExtCommunityListEntry**(*community*, *action*, *type*)

Bases: tuple

The ExtCommunityListEntry represents the entries for the extended community access lists.

Variables

- **community** (*str*) – community id
- ***action*** (*str*) – 'permit' or 'deny'
- ***type*** (*str*) – 'rt' (Route Target) or 'soo' (Site of Origin) (required)

13.7.5 OSPF Elements

Dynamic Routing can be enabled on devices configured in FW/VPN mode. Configuring dynamic routing consists of enabling the routing protocol on the engine and adding the routing elements on the interfaces at the engine routing level.

For adding OSPF configurations, several steps are required:

- Enable OSPF on the engine and specific the OSPF Profile
- Create or locate an existing OSPFArea to be used
- Modify the routing interface and add the OSPFArea

Enable OSPF on an existing engine using the default OSPF system profile:

```
engine.ospf.enable()
```

Create an OSPFArea using the default OSPF Interface Setting profile:

```
OSPFArea.create(name='customOSPFArea')
```

Add OSPF area to an interface routing configuration (add to nicid '0'):

```
interface = engine.routing.get(0)
interface.add_ospf_area(area)
```

Disable OSPF on an engine:

```
engine.ospf.disable()
```

Finding profiles or elements can also be done through collections:

```
>>> list(OSPFProfile.objects.all())
[OSPFProfile(name=Default OSPFv2 Profile)]

>>> list(OSPFArea.objects.all())
[OSPFArea(name=area0)]
```

The OSPF relationship can be represented as:

```
Engine --uses an--> OSPF Profile --has-a--> OSPF Domain Setting
Engine Routing --uses-an--> OSPF Area --has-a--> OSPF Interface Setting
```

Only Layer3Firewall and Layer3VirtualEngine types can support running OSPF.

See also:

smc.core.engines.Layer3Firewall and *smc.core.engines.Layer3VirtualEngine*

class smc.routing.ospf.**OSPF**(*data=None*)

OSPF configuration on the engine. Access through an engine reference:

```
engine.dynamic_routing.ospf.status
engine.dynamic_rotuing.ospf.enable(....)
```

When making changes to the OSPF configuration, any methods called that change the configuration also require that engine.update() is called once changes are complete. This way you can make multiple changes without refreshing the engine cache.

Variables **profile** (OSPFProfile) – OSPFProfile reference for this engine

disable()

Disable OSPF on this engine.

Returns None

enable(*ospf_profile=None*, *router_id=None*)

Enable OSPF on this engine. For master engines, enable OSPF on the virtual firewall.

Once enabled on the engine, add an OSPF area to an interface:

```
engine.dynamic_routing.ospf.enable()
interface = engine.routing.get(0)
interface.add_ospf_area(OSPFArea('myarea'))
```

Parameters

- **ospf_profile** (*str,OSPFProfile*) – OSPFProfile element or str href; if None, use default profile
- **router_id** (*str*) – single IP address router ID

Raises ***ElementNotFound*** – OSPF profile not found

Returns None

router_id
Get the router ID for this OSPF configuration. If None, then the ID will use the interface IP.

Returns str or None

status
Is OSPF enabled on this engine.

Return type bool

update_configuration (***kwargs*)
Update the OSPF configuration using kwargs that match the *enable* constructor.

Parameters **kwargs** (*dict*) – keyword arguments matching enable constructor.

Returns whether change was made

Return type bool

13.7.5.1 OSPFArea

class smc.routing.ospf.**OSPFArea**(*name*, ***meta*)
Bases: *smc.base.model.Element*

OSPF Area is an element that identifies general settings for an OSPF configuration applied to an engine routing node. The OSPFArea has a reference to an OSPFInterfaceSetting and is required when creating.

Create a basic OSPFArea with just area id:

```
OSPFArea.create(name='myarea', area_id=0)
```

Create an OSPFArea and use a custom OSPFInterfaceSetting element:

```
OSPFArea.create(
    name='customOSPFArea',
    interface_settings_ref=OSPFInterfaceSetting('myospf'),
    area_id=3)
```

Advanced example:

Adding ospf_virtual_links_endpoints:

```
OSPFArea.create(
    name='ospf',
    interface_settings_ref=intf,
    area_id=3,
    ospfv2_virtual_links_endpoints_container=
        [{'interface_settings_ref':
            'http://172.18.1.150:8082/6.1/elements/ospfv2_interface_settings/8',
          'router_id_endpoint_A': '192.168.1.1',
          'router_id_endpoint_B': '192.168.1.254'},
```

(continues on next page)

(continued from previous page)

```
        {'router_id_endpoint_A': '172.18.1.254',
         'router_id_endpoint_B': '172.18.1.200'}])
```

When using ABR substitute rules, there are 3 actions, 'aggregate', 'not_advertise' and 'substitute_with'. All references required are of type *smc.elements.network.Network*. These elements can either be created or retrieved using collections, or by getting the resource directly.

Example of creating an OSPF area and using ABR settings:

```
OSPFArea.create(
        name='area_with_abr',
        interface_settings_ref=intf,
        area_id=1,
        ospf_abr_substitute_container=[
            {'subnet_ref': 'http://172.18.1.150:8082/6.1/elements/network/143',
             'substitute_ref': 'http://172.18.1.150:8082/6.1/elements/network/1547
↪',
             'substitute_type': 'substitute_with'},
            {'subnet_ref': 'http://172.18.1.150:8082/6.1/elements/network/979',
             'substitute_type': 'aggregate'}])
```

Variables

- **interface_settings_ref** (OSPFInterfaceSetting) – reference to the OSPFInterfaceSetting
- **inbound_filters** (*list(IPPrefixList,IPAccessList)*) – Inbound filters attached to this OSPF Area.
- **outbound_filters** (*list(IPPrefixList,IPAccessList)*) – Outbound filter attached to this OSPF Area.

classmethod create(*name*, *interface_settings_ref=None*, *area_id=1*, *area_type='normal'*, *outbound_filters=None*, *inbound_filters=None*, *shortcut_capable_area=False*, *ospfv2_virtual_links_endpoints_container=None*, *ospf_abr_substitute_container=None*, *comment=None*, ***kwargs*)

Create a new OSPF Area

Parameters

- **name** (*str*) – name of OSPFArea configuration
- **interface_settings_ref** (*str,OSPFInterfaceSetting*) – an OSPFInterfaceSetting element or href. If None, uses the default system profile
- **name** – area id
- **area_type** (*str*) – |normal|stub|not_so_stubby|totally_stubby| totally_not_so_stubby
- **outbound_filters** (*list*) – reference to an IPAccessList and or IPPrefixList. You can only have one outbound prefix or access list
- **inbound_filters** (*list*) – reference to an IPAccessList and or IPPrefixList. You can only have one outbound prefix or access list
- **shortcut_capable_area** – True|False
- **ospfv2_virtual_links_endpoints_container** (*list*) – virtual link endpoints

- **ospf_abr_substitute_container** (*list*) – substitute types: laggregate|not_advertise|substitute_with
- **comment** (*str*) – optional comment

Raises ***CreateElementFailed*** – failed to create with reason

Return type *OSPFArea*

classmethod update_or_create(*with_status=False*, ***kwargs*)

Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to `get()`. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

13.7.5.2 OSPFKeyChain

class smc.routing.ospf.**OSPFKeyChain**(*name*, ***meta*)

Bases: *smc.base.model.Element*

OSPF Key Chain is used for authenticating OSPFv2 packets. If required, create a key chain and specify authentication in the OSPFInterfaceSetting referencing this element.

Is message-digest authentication is required on an OSPFInterfaceSetting, first create the key chain and use the reference to create the ospf interface profile:

```
key_chain = OSPFKeyChain('secure-keychain') #obtain resource
OSPFInterfaceSetting.create(
    name='authenicated-ospf',
```

(continues on next page)

(continued from previous page)

```
    authentication_type='message_digest',
    key_chain_ref=key_chain.href)
```

classmethod create(*name*, *key_chain_entry*)

Create a key chain with list of keys

Key_chain_entry format is:

```
[{'key': 'xxxx', 'key_id': 1-255, 'send_key': True|False}]
```

Parameters

- **name** (*str*) – Name of key chain
- **key_chain_entry** (*list*) – list of key chain entries

Raises ***CreateElementFailed*** – create failed with reason

Returns instance with meta

Return type *OSPFKeyChain*

13.7.5.3 OSPFProfile

class smc.routing.ospf.**OSPFProfile**(*name*, ***meta*)

Bases: *smc.base.model.Element*

An OSPF Profile contains administrative distance and redistribution settings. An OSPF Profile is set on the engine element when enabling OSPF.

These settings are always in effect:

- No autosummary

Example of creating an OSPFProfile with the default domain profile:

```
OSPFProfile.create(name='myospf')
```

Note: Enable OSPF on engine using engine.ospf.enable()

Variables

- ***external_distance*** (*int*) – external distance metric
- **inter_distance** (*int*) – inter distance metric
- **intra_distance** (*int*) – intra distance metric
- **default_metric** (*int*) – set a default metric for all unset areas
- **redistribution_entry** (*list*) – settings for static, connected, etc
- **domain_settings_ref** (OSPFDomainSetting) – OSPF Domain Settings profile used for this OSPF Profile

classmethod create(*name*, *domain_settings_ref=None*, *external_distance=110*, *inter_distance=110*, *intra_distance=110*, *redistribution_entry=None*, *default_metric=None*, *comment=None*)

Create an OSPF Profile.

If providing a list of redistribution entries, provide in the following dict format:

{'enabled': boolean, 'metric_type': 'external_1' or 'external_2', 'metric': 2, 'type': 'kernel'}

Valid types for redistribution entries are: kernel, static, connected, bgp, and default_originate.

You can also provide a 'filter' key with either an IPAccessList or RouteMap element to use for further access control on the redistributed route type. If metric_type is not provided, external_1 (E1) will be used.

An example of a redistribution_entry would be:

```
{u'enabled': True,
 u'metric': 123,
 u'metric_type': u'external_2',
 u'filter': RouteMap('myroutemap'),
 u'type': u'static'}
```

Parameters

- **name** (*str*) – name of profile
- **domain_settings_ref** (*str,OSPFDomainSetting*) – OSPFDomainSetting element or href
- **external_distance** (*int*) – route metric (E1-E2)
- **inter_distance** (*int*) – routes learned from different areas (O IA)
- **intra_distance** (*int*) – routes learned from same area (O)
- **redistribution_entry** (*list*) – how to redistribute the OSPF routes.

Raises ***CreateElementFailed*** – create failed with reason

Return type *OSPFProfile*

classmethod update_or_create(*filter_key=None*, *with_status=False*, ***kwargs*)

Update or create the element. If the element exists, update it using the kwargs provided if the provided kwargs after resolving differences from existing values. When comparing values, strings and ints are compared directly. If a list is provided and is a list of strings, it will be compared and updated if different. If the list contains unhashable elements, it is skipped. To handle complex comparisons, override this method on the subclass and process the comparison seperately. If an element does not have a *create* classmethod, then it is considered read-only and the request will be redirected to `get()`. Provide a `filter_key` dict key/value if you want to match the element by a specific attribute and value. If no filter_key is provided, the name field will be used to find the element.

```
>>> host = Host('kali')
>>> print(host.address)
12.12.12.12
>>> host = Host.update_or_create(name='kali', address='10.10.10.10')
>>> print(host, host.address)
Host(name=kali) 10.10.10.10
```

Parameters

- **filter_key** (*dict*) – filter key represents the data attribute and value to use to find the element. If none is provided, the name field will be used.
- **kwargs** – keyword arguments mapping to the elements `create` method.
- **with_status** (*bool*) – if set to True, a 3-tuple is returned with (Element, modified, created), where the second and third tuple items are booleans indicating the status

Raises

- ***CreateElementFailed*** – could not create element with reason
- ***ElementNotFound*** – if read-only element does not exist

Returns element instance by type

Return type *Element*

13.7.5.4 OSPFDomainSetting

class smc.routing.ospf.**OSPFDomainSetting**(*name*, ***meta*)

Bases: *smc.base.model.Element*

An OSPF Domain Setting provides settings for area border router (ABR) type, throttle timer settings, and the max metric router link-state advertisement (LSA) settings.

An OSPF Profile requires a reference to an OSPF Domain Setting.

Create a custom OSPF Domain Setting element:

```
OSPFDomainSetting.create(
    name='mydomain',
    abr_type='standard',
    auto_cost_bandwidth=200,
    deprecated_algorithm=True)
```

classmethod create(*name*, *abr_type='cisco'*, *auto_cost_bandwidth=100*, *deprecated_algorithm=False*, *initial_delay=200*, *initial_hold_time=1000*, *max_hold_time=10000*, *shutdown_max_metric_lsa=0*, *startup_max_metric_lsa=0*)

Create custom Domain Settings

Domain settings are referenced by an OSPFProfile

Parameters

- **name** (*str*) – name of custom domain settings
- **abr_type** (*str*) – cisco|shortcut|standard
- **auto_cost_bandwidth** (*int*) – Mbits/s
- **deprecated_algorithm** (*bool*) – RFC 1518 compatibility
- **initial_delay** (*int*) – in milliseconds
- **initial_hold_type** (*int*) – in milliseconds
- **max_hold_time** (*int*) – in milliseconds
- **shutdown_max_metric_lsa** (*int*) – in seconds
- **startup_max_metric_lsa** (*int*) – in seconds

Raises ***CreateElementFailed*** – create failed with reason

Returns instance with meta

Return type *OSPFDomainSetting*

13.7.5.5 OSPFInterfaceSetting

class smc.routing.ospf.**OSPFInterfaceSetting**(*name*, ***meta*)

Bases: *smc.base.model.Element*

OSPF Interface Setting indicate specific configurations that are applied to the interface and OSPF Area configuration, including authentication.

If you require non-default settings applied to your interface OSPF instance, you can create a custom interface profile:

```
OSPFInterfaceSetting.create(
    name='myprofile',
    dead_interval=30,
    hello_interval=5)
```

When using authentication on interface settings, there are two types, password authentication (plain text) or message digest.

When specifying an authentication_type='password', the password parameter must be provided.

When specifying authentication_type='message_digest', the key_chain_ref parameter must be specified.

classmethod create(*name*, *dead_interval=40*, *hello_interval=10*, *hello_interval_type='normal'*, *dead_multiplier=1*, *mtu_mismatch_detection=True*, *retransmit_interval=5*, *router_priority=1*, *transmit_delay=1*, *authentication_type=None*, *password=None*, *key_chain_ref=None*)

Create custom OSPF interface settings profile

Parameters

- **name** (*str*) – name of interface settings
- **dead_interval** (*int*) – in seconds
- **hello_interval** (*str*) – in seconds
- **hello_interval_type** (*str*) – |normal|fast_hello
- **dead_multipler** (*int*) – fast hello packet multipler
- **mtu_mismatch_detection** (*bool*) – True|False
- **retransmit_interval** (*int*) – in seconds
- **router_priority** (*int*) – set priority
- **transmit_delay** (*int*) – in seconds
- **authentication_type** (*str*) – |password|message_digest
- **password** (*str*) – max 8 chars (required when authentication_type='password')
- **key_chain_ref** (*str,Element*) – OSPFKeyChain (required when authentication_type='message_digest')

Raises ***CreateElementFailed*** – create failed with reason

Returns instance with meta

Return type *OSPFInterfaceSetting*

13.8 Policies

Policy module represents the classes required to obtaining and manipulating policies within the SMC.

Policy is the top level base class for all policy subclasses such as *smc.policy.layer3.FirewallPolicy*, *smc.policy.layer2.Layer2Policy*, *smc.policy.ips.IPSPolicy*, smc.policy.inspection.InspectionPolicy, *smc.policy.file_filtering.FileFilteringPolicy*

Policy represents actions that are common to all policy types, however for options that are not possible in a policy type, the method is overridden to return None. For example, 'upload' is not called on a template policy, but instead on the policy referencing that template. Therefore 'upload' is overidden.

Note: It is not required to call open() and save() on SMC API >= 6.1. It is also optional on earlier versions but if longer running operations are needed, calling open() will lock the policy from test_external modifications until save() is called.

---

class smc.policy.policy.**Policy**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Policy is the base class for all policy types managed by the SMC. This base class is not intended to be instantiated directly.

Subclasses should implement create(....) individually as each subclass will likely have different input requirements.

All generic methods that are policy level, such as 'open', 'save', 'force_unlock', 'export', and 'upload' are encapsulated into this base class.

Variables

- **template** (Element) – The template associated with this policy. Can be None
- ***inspection_policy*** (InspectionPolicy) – related inspection policy
- **file_filtering_policy** (FileFilteringPolicy) – related file policy

force_unlock()

Forcibly unlock a locked policy

Returns None

rule_counters(*engine*, *duration_type='one_week'*, *duration=0*, *start_time=0*)

New in version 0.5.6: Obtain rule counters for this policy. Requires SMC >= 6.2

Rule counters can be obtained for a given policy and duration for those counters can be provided in duration_type. A custom start range can also be provided.

Parameters

- **engine** (Engine) – the target engine to obtain rule counters from
- **duration_type** (*str*) – duration for obtaining rule counters. Valid options are: one_day, one_week, one_month, six_months, one_year, custom, since_last_upload
- **duration** (*int*) – if custom set for duration type, specify the duration in seconds (Default: 0)
- **start_time** (*int*) – start time in milliseconds (Default: 0)

Raises ActionCommandFailed

Returns list of rule counter objects

Return type RuleCounter

search_rule(*search*)
Search a rule for a rule tag or name value Result will be the meta data for rule (name, href, type)

Searching for a rule in specific policy:

```
f = FirewallPolicy(policy)
search = f.search_rule(searchable)
```

Parameters **search** (*str*) – search string

Returns rule elements matching criteria

Return type list(*Element*)

upload(*engine*, *timeout=5*, *wait_for_finish=False*, ***kw*)
Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.1 InterfacePolicy

New in version 0.5.6: Requires engine version >=6.3, SMC >=6.3

Interface Policies are applied at the engine level when layer 3 single FW's or cluster layer 3 FW's have layer 2 interfaces. The configuration is identical to creating Layer 2 Rules for layer 2 or IPS engines.

class smc.policy.interface.**InterfacePolicy**(*name*, ***meta*)
Bases: *smc.policy.interface.InterfaceRule*, *smc.policy.policy.Policy*

Layer 2 Interface Policy represents a set of rules applied to layer 2 interfaces installed on a single or cluster layer 3 FW. Set the interface policy on the engine properties. Interface policies do not have inspection policies and instead inherit from the engines primary policy.

Instance Resources:

Variables

- ***layer2_ipv4_access_rules*** – layer2_ipv4_access_rules
- ***layer2_ipv6_access_rules*** – layer2_ipv6_access_rules
- ***layer2_ethernet_rules*** – layer2_ethernet_rules

classmethod create(*name*, *template*)
Create a new Layer 2 Interface Policy.

Parameters

- **name** (*str*) – name of policy

- **template** (*str*) – name of the FW template to base policy on

Raises

- ***LoadPolicyFailed*** – cannot find policy by name
- ***CreatePolicyFailed*** – cannot create policy with reason

Returns Layer2InterfacePolicy

inspection_policy()
Descriptor to allow get/set operations on an element referenced in an Element.

class smc.policy.interface.**InterfaceRule**
Bases: object

Layer 2 Interface Rules are the same as Layer 2 FW/IPS rules.

layer2_ethernet_rules
Layer 2 Ethernet access rule

Return type rule_collection(*EthernetRule*)

layer2_ipv4_access_rules
Layer2 IPv4 access rule

Return type rule_collection(*IPv4Layer2Rule*)

layer2_ipv6_access_rules
Layer 2 IPv6 access rule

class smc.policy.interface.**InterfaceTemplatePolicy**(*name*, ***meta*)
Bases: *smc.policy.interface.InterfaceRule*, *smc.policy.policy.Policy*

Interface Template Policy. Required when creating a new Interface Policy. Useful for containing global rules or best practice configurations which will be inherited by the assigned policy.

```
print(list(InterfaceTemplatePolicy.objects.all())
```

inspection_policy()
Descriptor to allow get/set operations on an element referenced in an Element.

upload()
Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.2 FileFilteringPolicy

class smc.policy.file_filtering.**FileFilteringPolicy**(*name*, ***meta*)
Bases: *smc.policy.policy.Policy*

The File Filtering Policy references a specific file based policy for doing additional inspection based on file types. Use the policy parameters to specify how certain files are treated by either threat intelligence feeds,sandbox or by local AV scanning. You can also use this policy to disable threat prevention based on specific files.

export()
Export this element.

Usage:

```
engine = Engine('myfirewall')
extask = engine.export(filename='fooexport.zip')
while not extask.done():
    extask.wait(3)
print("Finished download task: %s" % extask.message())
print("File downloaded to: %s" % extask.filename)
```

Parameters **filename** (*str*) – filename to store exported element

Raises ***TaskRunFailed*** – invalid permissions, invalid directory, or this element is a system element and cannot be exported.

Returns DownloadTask

Note: It is not possible to export system elements

file_filtering_rules
File filtering rules for this policy.

Return type rule_collection(*FileFilteringRule*)

class smc.policy.file_filtering.**FileFilteringRule**(***meta*)
Bases: smc.policy.rule.RuleCommon, *smc.base.model.SubElement*

Represents a file filtering rule

13.8.3 FirewallPolicy

Layer 3 Firewall Policy

Module that represents resources related to creating and managing layer 3 firewall engine policies.

To get an existing policy:

```
>>> from smc.policy.layer3 import FirewallPolicy
>>> policy = FirewallPolicy('Standard Firewall Policy with Inspection')
>>> print(policy.template)
FirewallTemplatePolicy(name=Firewall Inspection Template)
```

Or through collections:

```
>>> list(FirewallPolicy.objects.all())
[FirewallPolicy(name=Standard Firewall Policy with Inspection),␣
↪FirewallPolicy(name=Layer 3 Virtual FW Policy)]
```

To create a new policy, use:

```
policy = FirewallPolicy.create(name='newpolicy', template='layer3_fw_template')
```

Example rule creation:

```
policy = FirewallPolicy('Amazon Cloud')
policy.open() #Only required for SMC API <= 6.0
policy.fw_ipv4_access_rules.create(name='mynewrule', sources='any',
                                   destinations='any', services='any',
                                   action='permit')
policy.save() #Only required for SMC API <= 6.0
```

Example rule deletion:

```
policy = FirewallPolicy('Amazon Cloud')
for rule in policy.fw_ipv4_access_rules.all():
    if rule.name == 'mynewrule':
        rule.delete()
```

class smc.policy.layer3.**FirewallPolicy**(*name*, ***meta*)

Bases: *smc.policy.layer3.FirewallRule*, *smc.policy.policy.Policy*

FirewallPolicy represents a set of rules installed on layer 3 devices. Layer 3 FW's support either ipv4 or ipv6 rules.

They also have NAT rules and reference to an Inspection and File Filtering Policy.

Variables **template** – which policy template is used

Instance Resources:

Variables

- ***fw_ipv4_access_rules*** – *fw_ipv4_access_rules*
- ***fw_ipv4_nat_rules*** – ipv4_nat_rules
- ***fw_ipv6_access_rules*** – ipv6_access_rules
- ***fw_ipv6_nat_rules*** – ipv6_nat_rules

classmethod create(*name*, *template='Firewall Inspection Template'*)

Create Firewall Policy. Template policy is required for the policy. The template parameter should be the name of the firewall template.

This policy will then inherit the Inspection and File Filtering policy from the specified template.

Parameters

- **name** (*str*) – name of policy
- **template** (*str*) – name of the FW template to base policy on

Raises

- ***LoadPolicyFailed*** – Cannot load the policy after creation
- ***CreatePolicyFailed*** – policy creation failed with message

Returns FirewallPolicy

To use after successful creation, reference the policy to obtain context:

```
FirewallPolicy('newpolicy')
```

class smc.policy.layer3.**FirewallRule**
Bases: object

Encapsulates all references to firewall rule related entry points. This is referenced by multiple classes such as FirewallPolicy and FirewallPolicyTemplate.

fw_ipv4_access_rules
IPv4 rule entry point

Return type rule_collection(*IPv4Rule*)

fw_ipv4_nat_rules
IPv4NAT Rule entry point

Return type rule_collection(*IPv4NATRule*)

fw_ipv6_access_rules
IPv6 Rule entry point

Return type rule_collection(*IPv6Rule*)

fw_ipv6_nat_rules
IPv6NAT Rule entry point

Return type rule_collection(*IPv6NATRule*)

class smc.policy.layer3.**FirewallSubPolicy**(*name*, ***meta*)
Bases: *smc.policy.policy.Policy*

A Firewall Sub Policy is a rule section within a firewall policy that provides a container to create rules that are referenced from a 'jump' rule. Typically rules in a sub policy are similar in some fashion such as applying to a specific service. Sub Policies can also be delegated from an administrative perspective.

Firewall Sub Policies only provide access to creating IPv4 rules. NAT is done on the parent firewall policy:

```
p = FirewallSubPolicy('MySubPolicy')
p.fw_ipv4_access_rules.create(
    name='newule',
    sources='any',
    destinations='any',
    services=[TCPService('SSH')],
    action='discard')
```

classmethod create(*name*)
Create a sub policy. Only name is required. Other settings are inherited from the parent firewall policy (template, inspection policy, etc).

Parameters **name** (*str*) – name of sub policy

Raises ***CreateElementFailed*** – failed to create policy

Return type *FirewallSubPolicy*

fw_ipv4_access_rules
IPv4 rule entry point

Return type rule_collection(*IPv4Rule*)

class smc.policy.layer3.**FirewallTemplatePolicy**(*name*, ***meta*)
Bases: *smc.policy.layer3.FirewallPolicy*

All Firewall Policies will reference a firewall policy template.

Most templates will be pre-configured best practice configurations and rarely need to be modified. However, you may want to view the details of rules configured in a template or possibly insert additional rules.

For example, view rules in firewall policy template after loading the firewall policy:

```
policy = FirewallPolicy('Amazon Cloud')
for rule in policy.template.fw_ipv4_access_rules.all():
    print rule
```

upload()
Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.4 InspectionPolicy

class smc.policy.policy.**InspectionPolicy**(*name*, ***meta*)
Bases: *smc.policy.policy.Policy*

The Inspection Policy references a specific inspection policy that is a property (reference) to either a FirewallPolicy, IPSPolicy or Layer2Policy. This policy defines specific characteristics for threat based prevention. In addition, exceptions can be made at this policy level to bypass scanning based on the rule properties.

export()
Export this element.

Usage:

```
engine = Engine('myfirewall')
extask = engine.export(filename='fooexport.zip')
while not extask.done():
    extask.wait(3)
print("Finished download task: %s" % extask.message())
print("File downloaded to: %s" % extask.filename)
```

Parameters **filename** (*str*) – filename to store exported element

Raises ***TaskRunFailed*** – invalid permissions, invalid directory, or this element is a system element and cannot be exported.

Returns DownloadTask

Note: It is not possible to export system elements

upload()
Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.5 IPSPolicy

IPS Engine policy

Module that represents resources related to creating and managing IPS engine policies.

To get an existing policy:

```
>>> policy = IPSPolicy('Default IPS Policy')
>>> print(policy.template)
IPSTemplatePolicy(name=High-Security IPS Template)
```

Or through collections:

```
>>> from smc.policy.ips import IPSPolicy
>>> list(IPSPolicy.objects.all())
[IPSPolicy(name=Default IPS Policy), IPSPolicy(name=High-Security Inspection IPS␣
↪Policy)]
```

To create a new policy, use:

```
policy = IPSPolicy.create(name='my_ips_policy',
                          template='High Security Inspection Template')
policy.ips_ipv4_access_rules.create(name='ipsrule1',
                                    sources='any',
                                    action='continue')

for rule in policy.ips_ipv4_access_rules.all():
    print(rule)
```

Example rule deletion:

```
policy = IPSPolicy('Amazon Cloud')
for rule in policy.ips_ipv4_access_rules.all():
    if rule.name == 'ipsrule1':
        rule.delete()
```

class smc.policy.ips.**IPSPolicy**(*name*, ***meta*)

Bases: *smc.policy.ips.IPSRule*, *smc.policy.policy.Policy*

IPS Policy represents a set of rules installed on an IPS / IDS engine. IPS mode supports both inline and SPAN interface types and ethernet based rules. Layer 2 and IPS engines do not current features that require routed interfaces.

Variables **template** – which policy template is used

Instance Resources:

Variables

- ***ips_ipv4_access_rules*** – *ips_ipv4_access_rules*
- ***ips_ipv6_access_rules*** – *ips_ipv6_access_rules*
- ***ips_ethernet_rules*** – *ips_ethernet_rules*

classmethod create(*name*, *template='High-Security IPS Template'*)

Create an IPS Policy

Parameters

- **name** (*str*) – Name of policy
- **template** (*str*) – name of template

Raises ***CreatePolicyFailed*** – policy failed to create

Returns IPSPolicy

class smc.policy.ips.**IPSRule**

Bases: object

Encapsulates all references to IPS rule related entry points. This is referenced by multiple classes such as IPSPolicy and IPSPolicyTemplate.

ips_ethernet_rules

IPS Ethernet access rule

Return type rule_collection(*EthernetRule*)

ips_ipv4_access_rules

IPS ipv4 access rules

Return type rule_collection(*IPv4Layer2Rule*)

ips_ipv6_access_rules

class smc.policy.ips.**IPSTemplatePolicy**(*name*, ***meta*)

Bases: *smc.policy.ips.IPSPolicy*

All IPS Policies will reference an IPS policy template.

Most templates will be pre-configured best practice configurations and rarely need to be modified. However, you may want to view the details of rules configured in a template or possibly insert additional rules.

For example, view rules in an ips policy template after loading the ips policy:

```
policy = IPSPolicy('InlineIPS')
for rule in policy.template.ips_ipv4_access_rules.all():
    print(rule)
```

upload()

Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.6 Layer2Policy

Layer 2 Firewall Policy

Module that represents resources related to creating and managing layer 2 firewall engine policies.

To get an existing policy:

```
>>> from smc.policy.layer2 import Layer2Policy
>>> policy = Layer2Policy('MyLayer2Policy')
>>> print(policy.template)
Layer2TemplatePolicy(name=Layer 2 Firewall Inspection Template)
```

Or through collections:

```
>>> from smc.policy.layer2 import Layer2Policy
>>> list(Layer2Policy.objects.all())
[Layer2Policy(name=MyLayer2Policy)]
```

To create a new policy, use:

```
policy = Layer2Policy.create(name='newpolicy', template='layer2_fw_template')
```

Example rule creation:

```
policy = Layer2Policy('smcpython-l2')

policy.layer2_ipv4_access_rules.create(
                              name='nonerule',
                              sources='any',
                              destinations='any',
                              services='any',
                              logical_interfaces=[location_href_to_logical_interface])
```

Create Ethernet rule for layer 2 firewall:

```
policy.layer2_ethernet_rules.create(name='nonerule',
                                    sources='any',
                                    destinations='any',
                                    services='any')
```

Note: Leaving parameter logical_interfaces out of create will default to 'ANY'.

Example rule deletion:

```
policy = Layer2Policy('Amazon Cloud')
for rule in policy.layer2_ipv4_access_rules.all():
    if rule.name == 'myrule':
        print rule.delete()
```

class smc.policy.layer2.**Layer2Policy**(*name*, ***meta*)
Bases: *smc.policy.layer2.Layer2Rule*, *smc.policy.policy.Policy*

Layer 2 Policy represents a set of rules installed on a layer 2 firewall engine. Layer 2 mode supports both inline and SPAN interface types and ethernet based rules. Layer 2 and IPS engines do not current features that require routed interfaces.

Variables **template** – which policy template is used

Instance Resources:

Variables

- ***layer2_ipv4_access_rules*** – *layer2_ipv4_access_rules*
- ***layer2_ipv6_access_rules*** – *layer2_ipv6_access_rules*
- ***layer2_ethernet_rules*** – *layer2_ethernet_rules*

classmethod create(*name*, *template='Layer 2 Firewall Inspection Template'*)
Create Layer 2 Firewall Policy. Template policy is required for the policy. The template parameter should be the name of the template.

The template should exist as a layer 2 template policy and should be referenced by name.

This policy will then inherit the Inspection and File Filtering policy from the specified template.

To use after successful creation, reference the policy to obtain context:

```
Layer2Policy('newpolicy')
```

Parameters

- **name** (*str*) – name of policy
- **template** (*str*) – name of the FW template to base policy on

Raises

- ***LoadPolicyFailed*** – cannot find policy by name
- ***CreatePolicyFailed*** – cannot create policy with reason

Returns Layer2Policy

class smc.policy.layer2.**Layer2Rule**
Bases: object

Encapsulates all references to layer 2 firewall rule related entry points. This is referenced by multiple classes such as Layer2Policy and Layer2TemplatePolicy.

layer2_ethernet_rules
Layer 2 Ethernet access rule

Return type rule_collection(*EthernetRule*)

layer2_ipv4_access_rules
Layer2 Firewall access rule

Return type rule_collection(*IPv4Layer2Rule*)

layer2_ipv6_access_rules
Layer 2 IPv6 access rule

class smc.policy.layer2.**Layer2TemplatePolicy**(*name*, ***meta*)
Bases: *smc.policy.layer2.Layer2Policy*

All Layer 2 Firewall Policies will reference a firewall policy template.

Most templates will be pre-configured best practice configurations and rarely need to be modified. However, you may want to view the details of rules configured in a template or possibly insert additional rules.

For example, view rules in the layer 2 policy template after loading the firewall policy:

```
policy = Layer2Policy('Amazon Cloud')
for rule in policy.template.layer2_ipv4_access_rules.all():
    print rule
```

upload()
Upload policy to specific device. Using wait for finish returns a poller thread for monitoring progress:

```
policy = FirewallPolicy('_NSX_Master_Default')
poller = policy.upload('myfirewall', wait_for_finish=True)
while not poller.done():
    poller.wait(3)
    print(poller.task.progress)
print("Task finished: %s" % poller.message())
```

Parameters **engine** (*str*) – name of device to upload policy to

Raises TaskRunFailed

Returns TaskOperationPoller

13.8.7 QoSPolicy

QoS Policy that would be applied to a rule set or physical / tunnel interface. QoS can also be applied at the VLAN level of an interface.

class smc.policy.qos.**QoSPolicy**(*name*, ***meta*)
Bases: *smc.base.model.Element*

13.9 Sub Policies

Sub Policies are referenced from within a normal policy as a parameter to a 'jump' action. They provide rule encapsulation for similar rules and can be delegated to an Admin User for more granular policy control.

13.9.1 FirewallSubPolicy

class smc.policy.layer3.**FirewallSubPolicy**(*name*, ***meta*)
Bases: *smc.policy.policy.Policy*

A Firewall Sub Policy is a rule section within a firewall policy that provides a container to create rules that are referenced from a 'jump' rule. Typically rules in a sub policy are similar in some fashion such as applying to a specific service. Sub Policies can also be delegated from an administrative perspective.

Firewall Sub Policies only provide access to creating IPv4 rules. NAT is done on the parent firewall policy:

```
p = FirewallSubPolicy('MySubPolicy')
p.fw_ipv4_access_rules.create(
    name='newule',
    sources='any',
    destinations='any',
    services=[TCPService('SSH')],
    action='discard')
```

classmethod create(*name*)
Create a sub policy. Only name is required. Other settings are inherited from the parent firewall policy (template, inspection policy, etc).

Parameters **name** (*str*) – name of sub policy

Raises ***CreateElementFailed*** – failed to create policy

Return type *FirewallSubPolicy*

fw_ipv4_access_rules
IPv4 rule entry point

Return type rule_collection(*IPv4Rule*)

13.10 Rules

Represents classes responsible for configuring rule types.

13.10.1 Rule

class smc.policy.rule.**Rule**
Bases: object

Top level rule construct with methods required to modify common behavior of any rule types. To retrieve a rule, access by reference:

```
policy = FirewallPolicy('mypolicy')
for rule in policy.fw_ipv4_nat_rules.all():
    print(rule.name, rule.comment, rule.is_disabled)
```

action
Action for this rule.

Return type *Action*

authentication_options
Read only authentication options field

Return type *AuthenticationOptions*

comment
Optional comment for this rule.

Parameters **value** (*str*) – string comment

Return type str

destinations
Destinations for this rule

Return type *Destination*

disable()
Disable this rule

enable()
Enable this rule

is_disabled
Whether the rule is enabled or disabled

Parameters **value** (*bool*) – True, False

Return type bool

is_rule_section
Is this rule considered a rule section

Return type bool

move_rule_after(*other_rule*)
Add this rule after another. This process will make a copy of the existing rule and add after the specified rule. If this raises an exception, processing is stopped. Otherwise the original rule is then deleted. You must re-retrieve the new element after running this operation as new references will be created.

Parameters **Rule** (*other_rule*) – rule where this rule will be positioned after

Raises ***CreateRuleFailed*** – failed to duplicate this rule, no move is made

move_rule_before(*other_rule*)
Move this rule after another. This process will make a copy of the existing rule and add after the specified rule. If this raises an exception, processing is stopped. Otherwise the original rule is then deleted. You must re-retrieve the new element after running this operation as new references will be created.

Parameters **Rule** (*other_rule*) – rule where this rule will be positioned before

Raises ***CreateRuleFailed*** – failed to duplicate this rule, no move is made

name
Name attribute of rule element

options
Rule based options for logging. Enabling and disabled specific log settings.

Return type *LogOptions*

parent_policy
Read-only name of the parent policy

Returns *smc.base.model.Element* of type policy

save()
After making changes to a rule element, you must call save to apply the changes. Rule changes are made to cache before sending to SMC.

Raises ***PolicyCommandFailed*** – failed to save with reason

Returns None

services
Services assigned to this rule

Return type *Service*

sources
Sources assigned to this rule

Return type *Source*

tag
Value of rule tag. Read only.

Returns rule tag

Return type str

13.10.1.1 IPv4Rule

class smc.policy.rule.**IPv4Rule**(***meta*)
Bases: smc.policy.rule.RuleCommon, *smc.policy.rule.Rule*, *smc.base.model.SubElement*

Represents an IPv4 Rule for a layer 3 engine.

Create a rule:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create(name='smcpython',
                                   sources='any',
                                   destinations='any',
                                   services='any')
```

Sources and Destinations can be one of any valid network element types defined in *smc.elements.network*.

Source entries by href:

```
sources=['http://1.1.1.1:8082/elements/network/myelement',
         'http://1.1.1.1:8082/elements/host/myhost'], etc
```

Source entries using network elements:

```
sources=[Host('myhost'), Network('thenetwork'), AddressRange('range')]
```

Services have a similar syntax and can take any type of *smc.elements.service* or the element href or both:

```
services=[TCPService('myservice'),
          'http://1.1.1.1/8082/elements/tcp_service/mytcpservice',
          'http://1.1.1.1/8082/elements/udp_server/myudpservice'], etc
```

You can obtain services and href for the elements by using the *smc.base.collection* collections:

```
>>> services = list(TCPService.objects.filter('80'))
>>> for service in services:
...   print(service, service.href)
...
(TCPService(name=tcp80443), u'http://172.18.1.150:8082/6.1/elements/tcp_service/
↪3535')
(TCPService(name=HTTP to Web SaaS), u'http://172.18.1.150:8082/6.1/elements/tcp_
↪service/589')
(TCPService(name=HTTP), u'http://172.18.1.150:8082/6.1/elements/tcp_service/440')
```

Services by application (get all facebook applications):

```
>>> applications = Search.objects.entry_point('application_situation').filter(
↪'facebook')
>>> print(list(applications))
[ApplicationSituation(name=Facebook-Plugins-Share-Button),␣
↪ApplicationSituation(name=Facebook-Plugins]
...
```

Sources / Destinations and Services can also take the string value 'any' to allow all. For example:

```
sources='any'
```

create(*name*, *sources=None*, *destinations=None*, *services=None*, *action='allow'*, *log_options=None*, *authentication_options=None*, *connection_tracking=None*, *is_disabled=False*, *vpn_policy=None*, *mobile_vpn=False*, *add_pos=None*, *after=None*, *before=None*, *sub_policy=None*, *comment=None*, ***kw*)

Create a layer 3 firewall rule

Parameters

- **name** (*str*) – name of rule
- **sources** (*list[str, Element]*) – source/s for rule
- **destinations** (*list[str, Element]*) – destination/s for rule
- **services** (*list[str, Element]*) – service/s for rule
- **action** (*Action or str*) – allow,continue,discard,refuse,enforce_vpn, apply_vpn,forward_vpn, blacklist (default: allow)
- **log_options** (*LogOptions*) – LogOptions object
- **connection_tracking** (*ConnectionTracking*) – custom connection tracking settings
- **authentication_options** (*AuthenticationOptions*) – options for auth if any
- **vpn_policy** (*PolicyVPN,str*) – policy element or str href; required for enforce_vpn, use_vpn and apply_vpn actions
- **mobile_vpn** (*bool*) – if using a vpn action, you can set mobile_vpn to True and omit the vpn_policy setting if you want this VPN to apply to any mobile VPN based on the policy VPN associated with the engine
- **sub_policy** (*str,Element*) – sub policy required when rule has an action of 'jump'. Can be the FirewallSubPolicy element or href.
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with *after* and *before* params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with *add_pos* and *before* params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with *add_pos* and *after* params.
- **comment** (*str*) – optional comment for this rule

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns the created ipv4 rule

Return type *IPv4Rule*

create_rule_section(*name*, *add_pos=None*, *after=None*, *before=None*)

Create a rule section in a Firewall Policy. To specify a specific numbering position for the rule section, use the *add_pos* field. If no position or before/after is specified, the rule section will be placed at the top which will encapsulate all rules below. Create a rule section for the relavant policy:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create_rule_section(name='attop')
# For NAT rules
policy.fw_ipv4_nat_rules.create_rule_section(name='mysection', add_pos=5)
```

Parameters

- **name** (*str*) – create a rule section by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns the created ipv4 rule

Return type *IPv4Rule*

13.10.1.2 IPv4Layer2Rule

class smc.policy.rule.**IPv4Layer2Rule**(***meta*)

Bases: smc.policy.rule.RuleCommon, *smc.policy.rule.Rule*, *smc.base.model.SubElement*

Create IPv4 rules for Layer 2 Firewalls

Example of creating an allow all rule:

```
policy = Layer2Policy('mylayer2')
policy.layer2_ipv4_access_rules.create(name='myrule',
                                       sources='any',
                                       destinations='any',
                                       services='any')
```

create(*name*, *sources=None*, *destinations=None*, *services=None*, *action='allow'*, *is_disabled=False*, *logical_interfaces=None*, *add_pos=None*, *after=None*, *before=None*, *comment=None*)

Create an IPv4 Layer 2 FW rule

Parameters

- **name** (*str*) – name of rule
- **sources** (*list[str, Element]*) – source/s for rule
- **destinations** (*list[str, Element]*) – destination/s for rule
- **services** (*list[str, Element]*) – service/s for rule
- **Action action** (*str,*) – |allow|continue|discard|refuse|blacklist
- **is_disabled** (*bool*) – whether to disable rule or not
- **logical_interfaces** (*list*) – logical interfaces by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.
- **comment** (*str*) – optional comment for this rule

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns newly created rule

Return type *IPv4Layer2Rule*

create_rule_section(*name*, *add_pos=None*, *after=None*, *before=None*)

Create a rule section in a Firewall Policy. To specify a specific numbering position for the rule section, use the *add_pos* field. If no position or before/after is specified, the rule section will be placed at the top which will encapsulate all rules below. Create a rule section for the relavant policy:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create_rule_section(name='attop')
# For NAT rules
policy.fw_ipv4_nat_rules.create_rule_section(name='mysection', add_pos=5)
```

Parameters

- **name** (*str*) – create a rule section by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.

- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns the created ipv4 rule

Return type *IPv4Rule*

13.10.1.3 EthernetRule

class smc.policy.rule.**EthernetRule**(***meta*)

Bases: smc.policy.rule.RuleCommon, *smc.policy.rule.Rule*, *smc.base.model.SubElement*

Ethernet Rule represents a policy on a layer 2 or IPS engine.

If logical_interfaces parameter is left blank, 'any' logical interface is used.

Create an ethernet rule for a layer 2 policy:

```
policy = Layer2Policy('layer2policy')
policy.layer2_ethernet_rules.create(name='l2rule',
                                    logical_interfaces=['dmz'],
                                    sources='any',
                                    action='discard')
```

create(*name*, *sources=None*, *destinations=None*, *services=None*, *action='allow'*, *is_disabled=False*, *logical_interfaces=None*, *add_pos=None*, *after=None*, *before=None*, *comment=None*)

Create an Ethernet rule

Parameters

- **name** (*str*) – name of rule
- **sources** (*list[str, Element]*) – source/s for rule
- **destinations** (*list[str, Element]*) – destination/s for rule
- **services** (*list[str, Element]*) – service/s for rule
- **action** (*str*) – |allow|continue|discard|refuse|blacklist
- **is_disabled** (*bool*) – whether to disable rule or not
- **logical_interfaces** (*list*) – logical interfaces by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.

Raises

- **MissingReuqiredInput** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns newly created rule

Return type *EthernetRule*

create_rule_section(*name*, *add_pos=None*, *after=None*, *before=None*)

Create a rule section in a Firewall Policy. To specify a specific numbering position for the rule section, use the *add_pos* field. If no position or before/after is specified, the rule section will be placed at the top which will encapsulate all rules below. Create a rule section for the relavant policy:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create_rule_section(name='attop')
# For NAT rules
policy.fw_ipv4_nat_rules.create_rule_section(name='mysection', add_pos=5)
```

Parameters

- **name** (*str*) – create a rule section by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns the created ipv4 rule

Return type *IPv4Rule*

13.10.1.4 IPv6Rule

class smc.policy.rule.**IPv6Rule**(***meta*)

Bases: *smc.policy.rule.IPv4Rule*

IPv6 access rule defines sources and destinations that must be in IPv6 format.

Note: It is possible to submit a source or destination in IPv4 format, however this will fail validation when attempting to push policy.

13.10.2 NATRule

class smc.policy.rule_nat.**NATRule**
Bases: *smc.policy.rule.Rule*

dynamic_src_nat
Dynamic Source NAT configuration for this NAT rule.

Return type *DynamicSourceNAT*

static_dst_nat
Static Destination NAT configuration for this NAT rule

Return type StaticDestNAT

static_src_nat
Static Source NAT configuraiton for this NAT rule.

Return type *StaticSourceNAT*

used_on
Used on specific whether this NAT rule has a specific engine that this rule applies to. Default is ANY (unspecified).

Parameters **value** (*str*, Element) – *smc.elements.network* element to apply to this NAT rule, or str href

Returns Element value: name of element this NAT rule is applied on

13.10.2.1 IPv4NATRule

class smc.policy.rule_nat.**IPv4NATRule**(***meta*)
Bases: smc.policy.rule.RuleCommon, *smc.policy.rule_nat.NATRule*, *smc.base.model.SubElement*

Create NAT Rules for relevant policy types. Rule requirements are similar to a normal rule with exception of the NAT field and no action field.

Like policy rules, specifying source/destination and services can be done either using the element href or element defined in element classes defined under package smc.elements. For example, using networks from *smc.elements.network* or services from *smc.elements.service*.

Example of creating a dynamic source NAT for host 'kali':

```
policy = FirewallPolicy('smcpython')
policy.fw_ipv4_nat_rules.create(name='mynat',
                                sources=[Host('kali')],
                                destinations='any',
                                services='any',
                                dynamic_src_nat='1.1.1.1',
                                dynamic_src_nat_ports=(1024,65535))
```

Example of creating a static source NAT for host 'kali':

```
policy.fw_ipv4_nat_rules.create(name='mynat',
                                sources=[Host('kali')],
                                destinations='any',
                                services='any',
                                static_src_nat='1.1.1.1')
```

Example of creating a destination NAT rule for destination host '3.3.3.3' with destination translation address of '1.1.1.1':

```
policy.fw_ipv4_nat_rules.create(name='mynat',
                                sources='any',
                                destinations=[Host('3.3.3.3')],
                                services='any',
                                static_dst_nat='1.1.1.1')
```

Destination NAT with destination port translation:

```
policy.fw_ipv4_nat_rules.create(name='aws_client',
                                sources='any',
                                destinations=[Alias('$$ Interface ID 0.ip')],
                                services='any',
                                static_dst_nat='1.1.1.1',
                                static_dst_nat_ports=(2222,22),
                                used_on=engine.href)
```

Create an any/any no NAT rule from host 'kali':

```
policy.fw_ipv4_nat_rules.create(name='nonat',
                                sources=[Host('kali')],
                                destinations='any',
                                services='any')
```

create(*name*, *sources=None*, *destinations=None*, *services=None*, *dynamic_src_nat=None*, *dynamic_src_nat_ports=(1024, 65535)*, *static_src_nat=None*, *static_dst_nat=None*, *static_dst_nat_ports=None*, *is_disabled=False*, *used_on=None*, *add_pos=None*, *after=None*, *before=None*, *comment=None*)

Create a NAT rule.

When providing sources/destinations or services, you can provide the element href, network element or services from `smc.elements`. You can also mix href strings with Element types in these fields.

Parameters

- **name** (*str*) – name of NAT rule
- **sources** (*list(str,Element)*) – list of sources by href or Element
- **destinations** (*list(str,Element)*) – list of destinations by href or Element
- **services** (*list(str,Element)*) – list of services by href or Element
- **dynamic_src_nat** (*str,Element*) – str ip or Element for dest NAT
- **dynamic_src_nat_ports** (*tuple*) – starting and ending ports for PAT. Default: (1024, 65535)
- **static_src_nat** (*str*) – ip or element href of used for source NAT
- **static_dst_nat** (*str*) – destination NAT IP address or element href
- **static_dst_nat_ports** (*tuple*) – ports or port range used for original and destination ports (only needed if a different destination port is used and does not match the rules service port)
- **is_disabled** (*bool*) – whether to disable rule or not
- **used_on** (*str,Element*) – Can be a str href of an Element or Element of type AddressRange('ANY'), AddressRange('NONE') or an engine element.

- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.
- **comment** (*str*) – optional comment for the NAT rule

Raises

- ***InvalidRuleValue*** – if rule requirements are not met
- ***CreateRuleFailed*** – rule creation failure

Returns newly created NAT rule

Return type *IPv4NATRule*

create_rule_section(*name*, *add_pos=None*, *after=None*, *before=None*)

Create a rule section in a Firewall Policy. To specify a specific numbering position for the rule section, use the *add_pos* field. If no position or before/after is specified, the rule section will be placed at the top which will encapsulate all rules below. Create a rule section for the relavant policy:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create_rule_section(name='attop')
# For NAT rules
policy.fw_ipv4_nat_rules.create_rule_section(name='mysection', add_pos=5)
```

Parameters

- **name** (*str*) – create a rule section by name
- **add_pos** (*int*) – position to insert the rule, starting with position 1. If the position value is greater than the number of rules, the rule is inserted at the bottom. If add_pos is not provided, rule is inserted in position 1. Mutually exclusive with `after` and `before` params.
- **after** (*str*) – Rule tag to add this rule after. Mutually exclusive with `add_pos` and `before` params.
- **before** (*str*) – Rule tag to add this rule before. Mutually exclusive with `add_pos` and `after` params.

Raises

- ***MissingRequiredInput*** – when options are specified the need additional setting, i.e. use_vpn action requires a vpn policy be specified.
- ***CreateRuleFailed*** – rule creation failure

Returns the created ipv4 rule

Return type *IPv4Rule*

13.10.2.2 IPv6NATRule

class smc.policy.rule_nat.**IPv6NATRule**(***meta*)
Bases: *smc.policy.rule_nat.IPv4NATRule*

Represents an IPv6 NAT rule. Source and/or destination (depending on NAT type) should be an IPv6 address. It will be possible to submit an IPv4 address however the policy validation engine will fail when being deployed to an engine and the rule will be ignored.

13.10.3 RuleElements

class smc.policy.rule_elements.**RuleElement**
Rule Element encapsulates actions for source, destination and service fields.

add(*data*)
Add a single entry to field.

Entries can be added to a rule using the href of the element or by loading the element directly. Element should be of type *smc.elements.network*. After modifying rule, call *save()*.

Example of adding entry by element:

```
policy = FirewallPolicy('policy')
for rule in policy.fw_ipv4_nat_rules.all():
    if rule.name == 'therule':
        rule.sources.add(Host('myhost'))
        rule.save()
```

Note: If submitting type Element and the element cannot be found, it will be skipped.

Parameters **data** (*Element or str*) – entry to add

add_many(*data*)
Add multiple entries to field. Entries should be list format. Entries can be of types relavant to the field type. For example, for source and destination fields, elements may be of type *smc.elements.network* or be the elements direct href, or a combination of both.

Add several entries to existing rule:

```
policy = FirewallPolicy('policy')
for rule in policy.fw_ipv4_nat_rules.all():
    if rule.name == 'therule':
        rule.sources.add_many([Host('myhost'),
                               'http://1.1.1.1/hosts/12345'])
        rule.save()
```

Parameters **data** (*list*) – list of sources

Note: If submitting type Element and the element cannot be found, it will be skipped.

all()
Return all destinations for this rule. Elements returned are of the object type for the given element for further introspection.

Search the fields in rule:

```
for sources in rule.sources.all():
    print('My source: %s' % sources)
```

Returns elements by resolved object type

Return type list(*Element*)

all_as_href()
Return all elements without resolving to *smc.elements.network* or *smc.elements.service*. Just raw representation as href.

Returns elements in href form

Return type list

is_any
Is the field set to any

Return type bool

is_none
Is the field set to none

Return type bool

set_any()
Set field to any

set_none()
Set field to none

update_field(*elements*)
Update the field with a list of provided values but only if the values are different. Return a boolean indicating whether a change was made indicating whether *save* should be called. If the field is currently set to any or none, then no comparison is made and field is updated.

Parameters **elements** (*list*) – list of elements in href or Element format to compare to existing field

Return type bool

13.10.3.1 Source

class smc.policy.rule_elements.**Source**(*rule=None*)
Bases: *smc.policy.rule_elements.RuleElement*, smc.base.structs.NestedDict

Source fields for a rule

13.10.3.2 Destination

class smc.policy.rule_elements.**Destination**(*rule=None*)
Bases: *smc.policy.rule_elements.RuleElement*, smc.base.structs.NestedDict

Destination fields for a rule.

13.10.3.3 Service

class smc.policy.rule_elements.**Service**(*rule=None*)
Bases: *smc.policy.rule_elements.RuleElement*, smc.base.structs.NestedDict

Service fields for a rule

13.10.3.4 Action

class smc.policy.rule_elements.**Action**(*rule=None*)
Bases: smc.base.structs.NestedDict

This represents the action associated with the rule.

action
Action set for this rule

Parameters **value** (*str*) – allow|discard|continue|refuse|jump|apply_vpn |enforce_vpn|forward_vpn|blacklist

Return type str

connection_tracking_options
Enables connection tracking. The firewall allows or discards packets according to the selected Connection Tracking mode. Reply packets are allowed as part of the allowed connection without an explicit Access rule. Protocols that use a dynamic port assignment must be allowed using a Service with the appropriate Protocol Agent for that protocol (in Access rules and NAT rules).

Return type *ConnectionTracking*

decrypting
New in version 0.6.0: Requires SMC version >= 6.3.3

Whether the decryption is enabled on this rule.

Parameters **value** (*bool*) – True, False, None (inherit from continue rule)

Return type bool

deep_inspection
Selects traffic that matches this rule for checking against the Inspection Policy referenced by this policy. Traffic is inspected as the Protocol that is attached to the Service element in this rule.

Parameters **value** (*bool*) – True, False, None (inherit from continue rule)

Return type bool

dos_protection
Enable or disable DOS protection mode

Parameters **value** (*bool*) – True, False, None (inherit from continue rule)

Return type bool

file_filtering
(IPv4 Only) Inspects matching traffic against the File Filtering policy. Selecting this option should also activates the Deep Inspection option. You can further adjust virus scanning in the Inspection Policy.

Parameters **value** (*bool*) – True, False, None (inherit from continue rule)

Return type bool

mobile_vpn
Mobile VPN only applies to engines that support VPN and that have the action of 'enforce_vpn', 'apply_vpn' or 'forward_vpn' set. This will enable mobile VPN traffic on this VPN rule.

Parameters **value** (*boolean*) – set mobile vpn on or off

Return type boolean

scan_detection
Enable or disable Scan Detection for traffic that matches the rule. This overrides the option set in the Engine properties.

Enable scan detection on this rule:

```
for rule in policy.fw_ipv4_access_rules.all():
    rule.action.scan_detection = 'on'
```

Parameters **value** (*str*) – onlofflundefined

Returns scan detection setting (on,off,undefined)

Return type str

sub_policy
Sub policy is used when `action=jump`.

Return type *FirewallSubPolicy*

user_response
Read-only user response setting

vpn
Return vpn reference. Only used if 'enforce_vpn', 'apply_vpn', or 'forward_vpn' is the action type.

Parameters **value** (`PolicyVPN`) – set the policy VPN for VPN action

Return type *PolicyVPN*

13.10.3.5 ConnectionTracking

class smc.policy.rule_elements.**ConnectionTracking**(*rule=None*)
Bases: `smc.base.structs.NestedDict`

Connection tracking settings can be configured on a per rule basis to control settings such as enforced MSS and how to handle connection states.

Configuring a rule to enable MSS and set connection state tracking to normal:

```
for rule in policy.fw_ipv4_access_rules.all():
    rule.action.connection_tracking_options.mss_enforced = True
    rule.action.connection_tracking_options.state = 'normal'
    rule.action.connection_tracking_options.mss_enforced_min_max = (1400, 1450)
    rule.action.connection_tracking_options.sync_connections = True
    rule.save()
```

mss_enforced
Is MSS enforced

Parameters **value** (*bool*) – True, False

Returns bool

mss_enforced_min_max
Allows entering the Minimum and Maximum value for the MSS in bytes. Headers are not included in the MSS value; MSS concerns only the payload portion of the packet.

Parameters **int value** (*tuple*) – tuple containing (min, max) in bytes

Returns (min, max) values

Return type tuple

state
Connection tracking mode. See Stonesoft documentation for more info.

Parameters **value** (*str*) – no,loose,normal,strict

Returns str

sync_connections
Are sync connections enabled for this engine. If None, then this is set to inherit from a continue rule.

:return True, False, None (inherit from continue rule)

timeout
The timeout (in seconds) after which inactive connections are closed. This timeout only concerns idle connections. Connections are not cut because of timeouts while the hosts are still communicating.

Parameters **value** (*int*) – time in seconds

Returns int

13.10.3.6 LogOptions

class smc.policy.rule_elements.**LogOptions**(*rule=None*)
Bases: smc.base.structs.NestedDict

Log Options represent the settings related to per rule logging.

Example of obtaining a rule reference and turning logging on for a particular rule:

```
policy = FirewallPolicy('smcpython')
for rule in policy.fw_ipv4_access_rules.all():
    if rule.name == 'foo':
        rule.options.log_accounting_info_mode = True
        rule.options.log_level = 'stored'
        rule.options.application_logging = 'enforced'
        rule.options.user_logging = 'enforced'
        rule.save()
```

application_logging
Stores information about Application use. You can log spplication use even if you do not use Applications for access control.

Parameters **value** (*str*) – offldefaultlenforced

Returns str

log_accounting_info_mode
Both connection opening and closing are logged and information on the volume of traffic is collected. This option is not available for rules that issue alerts. If you want to create reports that are based on traffic volume, you must select this option for all rules that allow traffic that you want to include in the reports.

Parameters **value** (*bool*) – log accounting information (bits/bytes transferred)

Returns bool

log_closing_mode
Specifying False means no log entries are created when connections are closed. True will mean both connection opening and closing are logged, but no information is collected on the volume of traffic.

Parameters **value** (*bool*) – enable/disable accounting data

Returns bool

log_level
Configure per rule logging. It is recommended to configure an Any/Any/Any/Continue rule in position 1 if global logging is required. This can be used to override any global logging setting.

Parameters **value** (*str*) – none|stored|transient|essential|alert|undefined

Returns str

log_payload_additional
Stores packet payload extracted from the traffic. The collected payload provides information for some of the additional log fields depending on the type of traffic.

Parameters **value** (*bool*) – True, False

Returns bool

log_payload_excerpt
Stores an excerpt of the packet that matched. The maximum recorded excerpt size is 4 KB. This allows quick viewing of the payload in the logs view.

Parameters **value** (*bool*) – collect excerpt or not

Returns bool

log_payload_record
Records the traffic up to the limit that is set in the Record Length field.

Parameters **value** (*bool*) – True, False

Returns bool

log_severity
Read only log severity level

Returns str

user_logging
Stores information about Users when they are used as the Source or Destination of an Access rule. You must select this option if you want Users to be referenced by name in log entries, statistics, reports, and user monitoring. Otherwise, only the IP address associated with the User at the time the log was created is stored.

Parameters **value** (*str*) – off|default|enforced

Returns str

13.10.3.7 AuthenticationOptions

class smc.policy.rule_elements.**AuthenticationOptions**(*rule=None*)
Bases: smc.base.structs.NestedDict

Authentication options are set on a per rule basis and dictate whether a user requires identification to match.

methods
Read only authentication methods enabled

Returns list value: auth methods enabled

require_auth
Ready only authentication required

Returns boolean

timeout
Timeout between authentications

Returns int

users
List of users required to authenticate

Returns list

13.10.3.8 MatchExpression

class smc.policy.rule_elements.**MatchExpression**(*name*, ***meta*)
Bases: *smc.base.model.Element*

A match expression is used in the source / destination / service fields to group together elements into an 'AND'ed configuration. For example, a normal rule might have a source field that could include network=172.18.1.0/24 and zone=Internal objects. A match expression enables you to AND these elements together to enforce the match requires both. Logically it would be represented as (network 172.18.1.0/24 AND zone Internal).

```
>>> from smc.elements.network import Host, Zone
>>> from smc.policy.rule_elements import MatchExpression
>>> from smc.policy.layer3 import FirewallPolicy
>>> match = MatchExpression.create(name='mymatch', network_element=Host('kali'),␣
↪zone=Zone('Mail'))
>>> policy = FirewallPolicy('smcpython')
>>> policy.fw_ipv4_access_rules.create(name='myrule', sources=[match],␣
↪destinations='any', services='any')
'http://172.18.1.150:8082/6.2/elements/fw_policy/261/fw_ipv4_access_rule/2099740'
>>> rule = policy.search_rule('myrule')
...
>>> for source in rule[0].sources.all():
...   print(source, source.values())
...
MatchExpression(name=MatchExpression _1491760686976_2) [Zone(name=Mail),␣
↪Host(name=kali)]
```

Note: MatchExpression is currently only supported on source and destination fields.

classmethod create(*name*, *user=None*, *network_element=None*, *domain_name=None*, *zone=None*, *executable=None*)
Create a match expression

Parameters

- **name** (*str*) – name of match expression
- **user** (*str*) – name of user or user group

- **network_element** (Element) – valid network element type, i.e. host, network, etc
- **domain_name** (DomainName) – domain name network element
- **zone** (Zone) – zone to use
- **executable** (*str*) – name of executable or group

Raises *ElementNotFound* – specified object does not exist

Returns instance with meta

Return type *MatchExpression*

13.10.4 NATElements

class smc.policy.rule_nat.**NATElement**(*rule=None*)

Common structure for source and destination NAT configurations.

automatic_proxy

Is proxy arp enabled. Leaving this in the on state is recommended.

Parameters **value** (*bool*) – enable/disable proxy arp

Return type bool

has_nat

Is NAT already enabled (assuming modification) or newly created.

Returns boolean

set_none()

Clear the NAT field for this NAT rule. You must call *update* or *save* on the rule to commit this change.

Returns None

translated_value

The translated value for this NAT type. If this rule does not have a NAT value defined, this will return None.

Returns NATValue or None

Return type NATValue

update_field(*element_or_ip_address=None*, *start_port=None*, *end_port=None*, ***kw*)

Update the source NAT translation on this rule. You must call *save* or *update* on the rule to make this modification. To update the source target for this NAT rule, update the source field directly using rule.sources.update_field(...). This will automatically update the NAT value. This method should be used when you want to change the translated value or the port mappings for dynamic source NAT.

Starting and ending ports are only used for dynamic source NAT and define the available ports for doing PAT on the outbound connection.

Parameters

- **element_or_ip_address** (*str*,Element) – Element or IP address that is the NAT target
- **start_port** (*int*) – starting port value, only used for dynamic source NAT
- **end_port** (*int*) – ending port value, only used for dynamic source NAT
- **automatic_proxy** (*bool*) – whether to enable proxy ARP (default: True)

Returns boolean indicating whether the rule was modified

Return type bool

13.10.4.1 DynamicSourceNAT

class smc.policy.rule_nat.**DynamicSourceNAT**(*rule=None*)
Bases: *smc.policy.rule_nat.NATElement*

Dynamic source NAT is typically used for outbound traffic and typically uses a range of ports to perform PAT operations.

end_port
Ending port specified for outbound dynamic source NAT (PAT)

Return type int

start_port
Start port for dynamic source NAT (PAT)

Return type int

translated_value
The translated value for this NAT type. If this rule does not have a NAT value defined, this will return None.

Returns NATValue or None

Return type NATValue

13.10.4.2 StaticSourceNAT

class smc.policy.rule_nat.**StaticSourceNAT**(*rule=None*)
Bases: *smc.policy.rule_nat.NATElement*

Source NAT defines the available options for configuration. This is typically used for outbound traffic where you need to hide the original source address.

Example of changing existing source NAT rule to use a different source NAT address:

```
for rule in policy.fw_ipv4_nat_rules.all():
    if rule.name == 'sourcenat':
        rule.static_src_nat.translated_value = '10.10.50.50'
        rule.save()
```

13.10.4.3 DynamicSourceNAT

class smc.policy.rule_nat.**DynamicSourceNAT**(*rule=None*)
Bases: *smc.policy.rule_nat.NATElement*

Dynamic source NAT is typically used for outbound traffic and typically uses a range of ports to perform PAT operations.

end_port
Ending port specified for outbound dynamic source NAT (PAT)

Return type int

start_port
Start port for dynamic source NAT (PAT)

Return type int

translated_value
The translated value for this NAT type. If this rule does not have a NAT value defined, this will return None.

Returns NATValue or None

Return type NATValue

13.11 VPN

Represents classes responsible for configuring VPN settings such as PolicyVPN, RouteVPN and all associated configurations.

Note: See API reference documentation on the Engine for instructions on how to enable the engine for VPN.

13.11.1 PolicyVPN

class smc.vpn.policy.**PolicyVPN**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Create a new VPN Policy.

```
>>> PolicyVPN.create(name='myvpn')
PolicyVPN(name=myvpn)
>>> v = PolicyVPN('myvpn')
>>> print(v.vpn_profile)
VPNProfile(name=VPN-A Suite)
```

When making VPN Policy modifications, you must first call *open()*, make your modifications and then call *save()* followed by *close()*.

Variables **vpn_profile** (*VPNProfile*) – VPN Profile used by this Policy VPN

add_central_gateway(*gateway*)
Add SMC managed internal gateway to the Central Gateways of this VPN

Parameters **gateway** (Engine,ExternalGateway) – An external gateway, engine or href for the central gateway

Raises ***PolicyCommandFailed*** – could not add gateway

Returns None

static add_internal_gateway_to_vpn(*internal_gateway_href*, *vpn_policy*, *vpn_role='central'*)
Add an internal gateway (managed engine node) to a VPN policy based on the internal gateway href.

Parameters

- **internal_gateway_href** (*str*) – href for engine internal gw
- **vpn_policy** (*str*) – name of vpn policy
- **vpn_role** (*str*) – centrallsatellite

Returns True for success

Return type bool

add_mobile_gateway(*gateway*)
Add a mobile VPN gateway to this policy VPN. Example of adding or removing a mobile VPN gateway:

```
policy_vpn = PolicyVPN('myvpn')
policy_vpn.open()
policy_vpn.add_mobile_vpn_gateway(ExternalGateway('extgw3'))

for mobile_gateway in policy_vpn.mobile_gateway_node:
    if mobile_gateway.gateway == ExternalGateway('extgw3'):
        mobile_gateway.delete()
policy_vpn.save()
policy_vpn.close()
```

Parameters **gateway** (Engine,ExternalGateway) – An external gateway, engine or href for the mobile gateway

Raises ***PolicyCommandFailed*** – could not add gateway

add_satellite_gateway(*gateway*)
Add gateway node as a satellite gateway for this VPN. You must first have the gateway object created. This is typically used when you either want a hub-and-spoke topology or the test_external gateway is a non-SMC managed device.

Parameters **gateway** (Engine,ExternalGateway) – An external gateway, engine or href for the central gateway

Raises ***PolicyCommandFailed*** – could not add gateway

Returns None

central_gateway_node
Central Gateway Node acts as the hub of a hub-spoke VPN.

Return type *SubElementCollection*(*GatewayNode*)

close()
Close the policy. This is only a valid method for SMC version <= 6.1

Raises ***PolicyCommandFailed*** – close failed with reason

Returns None

classmethod create(*name*, *nat=False*, *mobile_vpn_toplogy_mode=None*, *vpn_profile=None*)
Create a new policy based VPN

Parameters

- **name** – name of vpn policy
- **nat** (*bool*) – whether to apply NAT to the VPN (default False)
- **mobile_vpn_toplogy_mode** – whether to allow remote vpn
- **vpn_profile** (*VPNProfile*) – reference to VPN profile, or uses default

Return type *PolicyVPN*

enable_disable_nat()
Enable or disable NAT on this policy. If NAT is disabled, it will be enabled and vice versa.

Returns None

mobile_gateway_node
Mobile Gateway's are represented by client endpoints connecting to the policy based VPN.

Return type *SubElementCollection*(*GatewayNode*)

mobile_vpn_topology
Is the policy VPN configured for mobile VPN gateways. Valid modes: 'Selected Gateways below', 'Only central Gateways from overall topology', 'All Gateways from overall topology', 'None'

nat
Is NAT enabled on this vpn policy

Returns NAT enabled

Return type bool

open()
Open the policy for editing. This is only a valid method for SMC version <= 6.1

Raises ***PolicyCommandFailed*** – couldn't open policy with reason

Returns None

satellite_gateway_node
Node level settings for configured satellite gateways

Return type *SubElementCollection*(*GatewayNode*)

save()
Save the policy after editing. This is only a valid method for SMC version <= 6.1

Raises ***PolicyCommandFailed*** – save failed with reason

Returns None

tunnels
Return all tunnels for this VPN. A tunnel is defined as two end points within the VPN topology. Endpoints are automatically configureed based on whether they are a central gateway or satellite gateway. This provides access to enabling/disabling and setting the preshared key for the linked endpoints. List all tunnel mappings for this policy vpn:

```
for tunnel in policy.tunnels:
    tunnela = tunnel.tunnel_side_a
    tunnelb = tunnel.tunnel_side_b
    print(tunnela.gateway)
    print(tunnelb.gateway)
```

Return type *SubElementCollection*(*GatewayTunnel*)

validate()
Return a validation string from the SMC after running validate on this VPN policy.

Returns status as string

Return type str

13.11.2 RouteVPN

New in version 0.5.6: Route based VPNs with multi-domain support, requires SMC >=6.3

Module for configuring Route Based VPN. Creating a route based VPN consists of creating a local and remote tunnel endpoint. Once you have the required endpoints, use TunnelEndpoint classmethods to create the VPN by type (i.e. GRE, IPSEC).

List all existing route based VPNs:

```
print(list(RouteVPN.objects.all()))
```

Example of fully provisioning an IPSEC wrapped RBVPN using a third party remote GW:

```
engine = Layer3Firewall.create(name='myfw', mgmt_ip='1.1.1.1', mgmt_network='1.1.1.0/
↪24')

# Add a second layer 3 interface for VPN
engine.physical_interface.add_layer3_interface(
    interface_id=1, address='10.10.10.10', network_value='10.10.10.0/24', zone_ref=
↪'vpn')

engine.tunnel_interface.add_layer3_interface(
    interface_id=1000,
    address='2.2.2.2',
    network_value='2.2.2.0/24')

# Enable VPN on the 'Internal Endpoint' interface
vpn_endpoint = engine.vpn_endpoint.get_contains('10.10.10.10')
vpn_endpoint.update(enabled=True)

# A Tunnel Endpoint pairs the interface of the NGFW with it's local VPN gateway.
# You must create a tunnel endpoint for both sides of the Route VPN.

# Create the local Tunnel Endpoint using the engine internal gateway
# and previously created tunnel interface
tunnel_if = engine.tunnel_interface.get(1000)
local_gateway = TunnelEndpoint.create_ipsec_endpoint(engine.vpn.internal_gateway,␣
↪tunnel_if)

# Define the remote side details

# Create the remote side network elements
Network.create(name='remotenet', ipv4_network='172.18.10.0/24')

# An ExternalGateway defines the remote side as a 3rd party gateway
# Add the address of the remote gateway and the network element created
# that defines the remote network/s.
gw = ExternalGateway.create(name='remotegw')
gw.external_endpoint.create(name='endpoint1', address='10.10.10.10')
gw.vpn_site.create(name='remotesite', site_element=[Network('remotenet')])

# Create the remote Tunnel Endpoint using the external gateway
remote_gateway = TunnelEndpoint.create_ipsec_endpoint(gw)

RouteVPN.create_ipsec_tunnel(
    name='myvpn',
    preshared_key='abcdefgh123456789',
    local_endpoint=local_gateway,
    remote_endpoint=remote_gateway)
```

Create a GRE Tunnel Mode RBVPN with a remote gateway (non-SMC managed):

```
engine = Engine('fw')

# Enable VPN endpoint on interface 0
# Note: An interface can have multiple IP addresses in which case you
# may want to get the VPN endpoint match by address
vpn_endpoint = None
for endpoint in engine.vpn_endpoint:
    if endpoint.physical_interface.interface_id == '0':
        endpoint.update(enabled=True)
        vpn_endpoint = endpoint
        break

# Create a new Tunnel Interface for the engine
engine.tunnel_interface.add_layer3_interface(
    interface_id=3000, address='30.30.30.30', network_value='30.30.30.0/24')

tunnel_interface =  engine.tunnel_interface.get(3000)
local_endpoint = TunnelEndpoint.create_gre_tunnel_endpoint(
    endpoint=vpn_endpoint, tunnel_interface=tunnel_interface)

# Create GRE tunnel endpoint for remote gateway
remote_endpoint = TunnelEndpoint.create_gre_tunnel_endpoint(
    remote_address='10.1.1.2')

# Create the top level IPSEC tunnel to encapsulate RBVPN
policy_vpn = PolicyVPN.create(name='myIPSEC')

RouteVPN.create_gre_tunnel_mode(
    name='mytunnelvpn',
    local_endpoint=local_endpoint,
    remote_endpoint=remote_endpoint,
    policy_vpn=policy_vpn)
```

Create a no-encryption GRE route based VPN between two managed NGFWs:

```
engine1 = Layer3Firewall.create(name='engine1', mgmt_ip='1.1.1.1', mgmt_network='1.1.
↪1.0/24')
engine1.tunnel_interface.add_layer3_interface(
    interface_id=1000,
    address='2.2.2.2',
    network_value='2.2.2.0/24')

# Obtain the 'internal endpoint' from the NGFW and enable VPN
for vpn in engine1.vpn_endpoint:
    internal_endpoint = vpn
    vpn.update(enabled=True)

tunnel_if = engine1.tunnel_interface.get(1000)
local_gateway = TunnelEndpoint.create_gre_tunnel_endpoint(
    internal_endpoint, tunnel_if)

engine2 = Layer3Firewall.create(name='engine2', mgmt_ip='1.1.1.1', mgmt_network='1.1.
↪1.0/24')
engine2.tunnel_interface.add_layer3_interface(
    interface_id=1000,
    address='2.2.2.2',
    network_value='2.2.2.0/24')
```

(continues on next page)

(continued from previous page)

```
# Obtain the 'internal endpoint' from the NGFW and enable VPN
for vpn in engine2.vpn_endpoint:
    internal_endpoint = vpn
    vpn.update(enabled=True)

tunnel_if = engine2.tunnel_interface.get(1000)
remote_gateway = TunnelEndpoint.create_gre_tunnel_endpoint(
    internal_endpoint, tunnel_if)

RouteVPN.create_gre_tunnel_no_encryption(
    name='openvpn',
    local_endpoint=local_gateway,
    remote_endpoint=remote_gateway)
```

class smc.vpn.route.**RouteVPN**(*name*, ***meta*)

Bases: *smc.base.model.Element*

Route based VPN in NGFW.

Variables

- **vpn_profile** (*VPNProfile*) – VPNProfile reference for this RouteVPN
- **monitoring_group** (TunnelMonitoringGroup) – tunnel monitoring group reference

classmethod create_gre_transport_mode(*name*, *local_endpoint*, *remote_endpoint*, *preshared_key*, *monitoring_group=None*, *vpn_profile=None*, *mtu=0*, *ttl=0*, *pmtu_discovery=True*, *enabled=True*, *comment=None*)

Create a transport based route VPN. This VPN type uses IPSEC for protecting the payload, therefore a VPN Profile is specified.

Parameters

- **name** (*str*) – name of VPN
- **local_endpoint** (TunnelEndpoint) – the local side endpoint for this VPN.
- **remote_endpoint** (TunnelEndpoint) – the remote side endpoint for this VPN.
- **preshared_key** (*str*) – preshared key for RBVPN
- **monitoring_group** (TunnelMonitoringGroup) – the group to place this VPN in for monitoring. (default: 'Uncategorized')
- **vpn_profile** (*VPNProfile*) – VPN profile for this VPN. (default: VPN-A Suite)
- **mtu** (*int*) – Set MTU for this VPN tunnel (default: 0)
- **pmtu_discovery** (*boolean*) – enable pmtu discovery (default: True)
- **ttl** (*int*) – ttl for connections on the VPN (default: 0)
- **comment** (*str*) – optional comment

Raises ***CreateVPNFailed*** – failed to create the VPN with reason

Return type *RouteVPN*

classmethod create_gre_tunnel_mode(*name*, *local_endpoint*, *remote_endpoint*, *policy_vpn*, *mtu=0*, *pmtu_discovery=True*, *ttl=0*, *enabled=True*, *comment=None*)

Create a GRE based tunnel mode route VPN. Tunnel mode GRE wraps the GRE tunnel in an IPSEC tunnel to provide encrypted end-to-end security. Therefore a policy based VPN is required to 'wrap' the GRE into IPSEC.

Parameters

- **name** (*str*) – name of VPN
- **local_endpoint** (TunnelEndpoint) – the local side endpoint for this VPN.
- **remote_endpoint** (TunnelEndpoint) – the remote side endpoint for this VPN.
- **policy_vpn** (PolicyVPN) – reference to a policy VPN
- **monitoring_group** (TunnelMonitoringGroup) – the group to place this VPN in for monitoring. (default: 'Uncategorized')
- **mtu** (*int*) – Set MTU for this VPN tunnel (default: 0)
- **pmtu_discovery** (*boolean*) – enable pmtu discovery (default: True)
- **ttl** (*int*) – ttl for connections on the VPN (default: 0)
- **comment** (*str*) – optional comment

Raises ***CreateVPNFailed*** – failed to create the VPN with reason

Return type *RouteVPN*

classmethod create_gre_tunnel_no_encryption(*name*, *local_endpoint*, *remote_endpoint*, *mtu=0*, *pmtu_discovery=True*, *ttl=0*, *enabled=True*, *comment=None*)

Create a GRE Tunnel with no encryption. See *create_gre_tunnel_mode* for constructor descriptions.

classmethod create_ipsec_tunnel(*name*, *local_endpoint*, *remote_endpoint*, *preshared_key=None*, *monitoring_group=None*, *vpn_profile=None*, *mtu=0*, *pmtu_discovery=True*, *ttl=0*, *enabled=True*, *comment=None*)

The VPN tunnel type negotiates IPsec tunnels in the same way as policy-based VPNs, but traffic is selected to be sent into the tunnel based on routing.

Parameters

- **name** (*str*) – name of VPN
- **local_endpoint** (TunnelEndpoint) – the local side endpoint for this VPN.
- **remote_endpoint** (TunnelEndpoint) – the remote side endpoint for this VPN.
- **preshared_key** (*str*) – required if remote endpoint is an ExternalGateway
- **monitoring_group** (TunnelMonitoringGroup) – the group to place this VPN in for monitoring. Default: 'Uncategorized'.
- **vpn_profile** (*VPNProfile*) – VPN profile for this VPN. (default: VPN-A Suite)
- **mtu** (*int*) – Set MTU for this VPN tunnel (default: 0)
- **pmtu_discovery** (*boolean*) – enable pmtu discovery (default: True)
- **ttl** (*int*) – ttl for connections on the VPN (default: 0)
- **enabled** (*bool*) – enable the RBVPN or leave it disabled

- **comment** (*str*) – optional comment

Raises ***CreateVPNFailed*** – failed to create the VPN with reason

Return type *RouteVPN*

disable()
Disable this route based VPN

Returns None

enable()
Enable this route based VPN

Returns None

local_endpoint
The local endpoint for this RBVPN

Return type *TunnelEndpoint*

remote_endpoint
The remote endpoint for this RBVPN

Return type *TunnelEndpoint*

set_preshared_key(*new_key*)
Set the preshared key for this VPN. A pre-shared key is only present when the tunnel type is 'VPN' or the encryption mode is 'transport'.

Returns None

tunnel_mode
The tunnel mode for this RBVPN

Return type str

class smc.vpn.route.**TunnelEndpoint**(*gateway_ref=None*, *tunnel_interface_ref=None*, *endpoint_ref=None*, *ip_address=None*)
Bases: object

A Tunnel Endpoint represents one side of a route based VPN. Based on the RBVPN type required, you must create the local and remote endpoints and pass them into the RouteVPN create classmethods.

Variables

- **gateway** (InternalGateway,ExternalGateway) – reference to the element that is used by this tunnel endpoint
- ***tunnel_interface*** (TunnelInterface) – Tunnel interface used by this tunnel endpoint

classmethod create_gre_transport_endpoint(*endpoint*, *tunnel_interface=None*)
Create the GRE transport mode endpoint. If the GRE transport mode endpoint is an SMC managed device, both an endpoint and a tunnel interface is required. If the GRE endpoint is an externally managed device, only an endpoint is required.

Parameters

- **endpoint** (InternalEndpoint,ExternalEndpoint) – the endpoint element for this tunnel endpoint.
- **tunnel_interface** (TunnelInterface) – the tunnel interface for this tunnel endpoint. Required for SMC managed devices.

Return type *TunnelEndpoint*

classmethod create_gre_tunnel_endpoint(*endpoint=None*, *tunnel_interface=None*, *remote_address=None*)

Create the GRE tunnel mode or no encryption mode endpoint. If the GRE tunnel mode endpoint is an SMC managed device, both an endpoint and a tunnel interface is required. If the endpoint is externally managed, only an IP address is required.

Parameters

- **endpoint** (InternalEndpoint,ExternalEndpoint) – the endpoint element for this tunnel endpoint.
- **tunnel_interface** (TunnelInterface) – the tunnel interface for this tunnel endpoint. Required for SMC managed devices.
- **remote_address** (*str*) – IP address, only required if the tunnel endpoint is a remote gateway.

Return type *TunnelEndpoint*

classmethod create_ipsec_endpoint(*gateway*, *tunnel_interface=None*)

Create the VPN tunnel endpoint. If the VPN tunnel endpoint is an SMC managed device, both a gateway and a tunnel interface is required. If the VPN endpoint is an externally managed device, only a gateway is required.

Parameters

- **gateway** (InternalGateway,ExternalGateway) – the gateway for this tunnel endpoint
- **tunnel_interface** (TunnelInterface) – Tunnel interface for this RBVPN. This can be None if the gateway is a non-SMC managed gateway.

Return type *TunnelEndpoint*

endpoint

Endpoint is used to specify which interface is enabled for VPN. This is the InternalEndpoint property of the InternalGateway.

Returns internal endpoint where VPN is enabled

Return type *InternalEndpoint*,*ExternalGateway*

remote_address

Show the remote IP address configured for a GRE RBVPN using Tunnel or No Encryption Mode configurations.

tunnel_interface

Show the tunnel interface for this TunnelEndpoint.

Returns interface for this endpoint

Return type *TunnelInterface*

class smc.vpn.route.**TunnelMonitoringGroup**(*name*, ***meta*)

Bases: *smc.base.model.Element*

A tunnel monitoring group is used to group route based VPNs for monitoring on the Home->VPN dashboard.

13.11.3 Gateways

13.11.3.1 ExternalGateway

VPN Elements are used in conjunction with Policy or Route Based VPN configurations. VPN elements consist of external gateway and VPN site settings that identify 3rd party gateways to be used as a VPN termination endpoint.

There are several ways to create an external gateway configuration. A step by step process which first creates a network element to be used in a VPN site, then creates the ExternalGatway, an ExternalEndpoint for the gateway, and inserts the VPN site into the configuration:

```
Network.create(name='mynetwork', ipv4_network='172.18.1.0/24')
gw = ExternalGateway.create(name='mygw')
gw.external_endpoint.create(name='myendpoint', address='10.10.10.10')
gw.vpn_site.create(name='mysite', site_element=[Network('mynetwork')])
```

You can also use the convenience method *update_or_create* on the ExternalGateway to fully provision in a single step. Note that the ExternalEndpoint and VPNSite also have an *update_or_create* method to limit the update to those respective configurations:

```
>>> from smc.elements.network import Network
>>> from smc.vpn.elements import ExternalGateway
>>> network = Network.get_or_create(name='network-172.18.1.0/24', ipv4_network='172.
↪18.1.0/24')
>>>
>>> g = ExternalGateway.update_or_create(name='newgw',
    external_endpoint=[
        {'name': 'endpoint1', 'address': '1.1.1.1', 'enabled': True},
        {'name': 'endpoint2', 'address': '2.2.2.2', 'enabled': True}],
    vpn_site=[{'name': 'sitea', 'site_element':[network]}])
>>> g
ExternalGateway(name=newgw)
>>> for endpoint in g.external_endpoint:
...   endpoint
...
ExternalEndpoint(name=endpoint1 (1.1.1.1))
ExternalEndpoint(name=endpoint2 (2.2.2.2))
>>> for site in g.vpn_site:
...   site, site.site_element
...
(VPNSite(name=sitea), [Network(name=network-172.18.1.0/24)])
```

Note: When calling *update_or_create* from the ExternalGateway, providing the parameters for external_endpoints and vpn_site is optional.

class smc.vpn.elements.**ExternalGateway**(*name*, ***meta*)

Bases: *smc.base.model.Element*

External Gateway defines an VPN Gateway for a non-SMC managed device. This will specify details such as the endpoint IP, and VPN site protected networks. Example of manually provisioning each step:

```
Network.create(name='mynetwork', ipv4_network='172.18.1.0/24')
gw = ExternalGateway.create(name='mygw')
gw.external_endpoint.create(name='myendpoint', address='10.10.10.10')
gw.vpn_site.create(name='mysite', site_element=[Network('mynetwork')])
```

Variables ***gateway_profile*** (GatewayProfile) – A gateway profile will define the capabilities (i.e. crypto) allowed for this VPN.

classmethod create(*name*, *trust_all_cas=True*)
Create new External Gateway

Parameters

- **name** (*str*) – name of test_external gateway
- **trust_all_cas** (*bool*) – whether to trust all internal CA's (default: True)

Returns instance with meta

Return type *ExternalGateway*

external_endpoint
An External Endpoint is the IP based definition for the destination VPN peers. There may be multiple per External Gateway. Add a new endpoint to an existing test_external gateway:

```
>>> list(ExternalGateway.objects.all())
[ExternalGateway(name=cisco-remote-side), ExternalGateway(name=remoteside)]
>>> gateway.external_endpoint.create('someendpoint', '12.12.12.12')
'http://1.1.1.1:8082/6.1/elements/external_gateway/22961/external_endpoint/
↪27467'
```

Return type *CreateCollection*(*ExternalEndpoint*)

trust_all_cas
Gateway setting identifying whether all CA's specified in the profile are supported or only specific ones.

Return type bool

classmethod update_or_create(*name*, *external_endpoint=None*, *vpn_site=None*, *trust_all_cas=True*, *with_status=False*)
Update or create an ExternalGateway. The `external_endpoint` and `vpn_site` parameters are expected to be a list of dicts with key/value pairs to satisfy the respective elements create constructor. VPN Sites will represent the final state of the VPN site list. ExternalEndpoint that are pre-existing will not be deleted if not provided in the `external_endpoint` parameter, however existing elements will be updated as specified.

Parameters

- **name** (*str*) – name of external gateway
- **external_endpoint** (*list(dict)*) – list of dict items with key/value to satisfy ExternalEndpoint.create constructor
- **vpn_site** (*list(dict)*) – list of dict items with key/value to satisfy VPNSite.create constructor
- **with_status** (*bool*) – If set to True, returns a 3-tuple of (ExternalGateway, modified, created), where modified and created is the boolean status for operations performed.

Raises **ValueError** – missing required argument/s for constructor argument

Return type *ExternalGateway*

vpn_site
A VPN site defines a collection of IP's or networks that identify address space that is defined on the other end of the VPN tunnel.

Return type *CreateCollection*(*VPNSite*)

13.11.3.2 ExternalEndpoint

class smc.vpn.elements.**ExternalEndpoint**(***meta*)
Bases: *smc.base.model.SubElement*

External Endpoint is used by the External Gateway and defines the IP and other VPN related settings to identify the VPN peer. This is created to define the details of the non-SMC managed gateway. This class is a property of *smc.vpn.elements.ExternalGateway* and should not be called directly. Add an endpoint to existing External Gateway:

```
gw = ExternalGateway('aws')
gw.external_endpoint.create(name='aws01', address='2.2.2.2')
```

create(*name*, *address=None*, *enabled=True*, *balancing_mode='active'*, *ipsec_vpn=True*, *nat_t=False*, *force_nat_t=False*, *dynamic=False*, *ike_phase1_id_type=None*, *ike_phase1_id_value=None*)
Create an test_external endpoint. Define common settings for that specify the address, enabled, nat_t, name, etc. You can also omit the IP address if the endpoint is dynamic. In that case, you must also specify the ike_phase1 settings.

Parameters

- **name** (*str*) – name of test_external endpoint
- **address** (*str*) – address of remote host
- **enabled** (*bool*) – True|False (default: True)
- **balancing_mode** (*str*) – active
- **ipsec_vpn** (*bool*) – True|False (default: True)
- **nat_t** (*bool*) – True|False (default: False)
- **force_nat_t** (*bool*) – True|False (default: False)
- **dynamic** (*bool*) – is a dynamic VPN (default: False)
- **ike_phase1_id_type** (*int*) – If using a dynamic endpoint, you must set this value. Valid options: 0=DNS name, 1=Email, 2=DN, 3=IP Address
- **ike_phase1_id_value** (*str*) – value of ike_phase1_id. Required if ike_phase1_id_type and dynamic set.

Raises *CreateElementFailed* – create element with reason

Returns newly created element

Return type *ExternalEndpoint*

enable_disable()
Enable or disable this endpoint. If enabled, it will be disabled and vice versa.

Returns None

enable_disable_force_nat_t()
Enable or disable NAT-T on this endpoint. If enabled, it will be disabled and vice versa.

Returns None

enabled
Whether this endpoint is enabled.

Return type bool

force_nat_t
Whether force_nat_t is enabled on this endpoint.

Return type bool

classmethod update_or_create(*external_gateway*, *name*, *with_status=False*, ***kw*)
Update or create external endpoints for the specified external gateway. An ExternalEndpoint is considered unique based on the IP address for the endpoint (you cannot add two external endpoints with the same IP). If the external endpoint is dynamic, then the name is the unique identifier.

Parameters

- **external_gateway** (ExternalGateway) – external gateway reference
- **name** (*str*) – name of the ExternalEndpoint. This is only used as a direct match if the endpoint is dynamic. Otherwise the address field in the keyword arguments will be used as you cannot add multiple external endpoints with the same IP address.
- **with_status** (*bool*) – If set to True, returns a 3-tuple of (ExternalEndpoint, modified, created), where modified and created is the boolean status for operations performed.
- **kw** (*dict*) – keyword arguments to satisfy ExternalEndpoint.create constructor

Raises

- ***CreateElementFailed*** – Failed to create external endpoint with reason
- ***ElementNotFound*** – If specified ExternalGateway is not valid

Returns if with_status=True, return tuple(ExternalEndpoint, created). Otherwise return only ExternalEndpoint.

13.11.4 VPNSite

class smc.vpn.elements.**VPNSite**(***meta*)
Bases: *smc.base.model.SubElement*

VPN Site information for an internal or test_external gateway Sites are used to encapsulate hosts or networks as 'protected' for VPN policy configuration.

Create a new vpn site for an engine:

```
engine = Engine('myengine')
network = Network('network-192.168.5.0/25') #get resource
engine.vpn.sites.create('newsite', [network.href])
```

Sites can also be added to ExternalGateway's as well:

```
extgw = ExternalGateway('mygw')
extgw.vpn_site.create('newsite', [Network('foo')])
```

This class is a property of *smc.core.engine.InternalGateway* or *smc.vpn.elements.ExternalGateway* and should not be accessed directly.

Variables **gateway** (InternalGateway,ExternalGateway) – gateway referenced

add_site_element(*element*)
Add a site element or list of elements to this VPN.

Parameters **element** (*list(str,Network)*) – list of Elements or href's of vpn site elements

Raises *UpdateElementFailed* – fails due to reason

Returns None

create(*name*, *site_element*)
Create a VPN site for an internal or external gateway

Parameters

- **name** (*str*) – name of site
- **site_element** (*list[str,Element]*) – list of protected networks/hosts

Raises *CreateElementFailed* – create element failed with reason

Returns href of new element

Return type str

site_element
Site elements for this VPN Site.

Returns Elements used in this VPN site

Return type list(*Element*)

classmethod update_or_create(*external_gateway*, *name*, *site_element=None*, *with_status=False*)
Update or create a VPN Site elements or modify an existing VPN site based on value of provided site_element list. The resultant VPN site end result will be what is provided in the site_element argument (can also be an empty list to clear existing).

Parameters

- **external_gateway** (ExternalGateway) – The external gateway for this VPN site
- **name** (*str*) – name of the VPN site
- **site_element** (*list(str,Element)*) – list of resolved Elements to add to the VPN site
- **with_status** (*bool*) – If set to True, returns a 3-tuple of (VPNSite, modified, created), where modified and created is the boolean status for operations performed.

Raises *ElementNotFound* – ExternalGateway or unresolved site_element

13.11.5 Other Elements

Other elements associated with VPN configurations

13.11.5.1 GatewaySettings

class smc.vpn.elements.**GatewaySettings**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Gateway settings define various VPN related settings that are applied at the firewall level such as negotiation timers and mobike settings. A gateway setting is a property of an engine:

```
engine = Engine('myfw')
engine.vpn.gateway_settings
```

classmethod create(*name*, *negotiation_expiration=200000*, *negotiation_retry_timer=500*, *negotiation_retry_max_number=32*, *negotiation_retry_timer_max=7000*, *certificate_cache_crl_validity=90000*, *mobike_after_sa_update=False*, *mobike_before_sa_update=False*, *mobike_no_rrc=True*)

Create a new gateway setting profile.

Parameters

- **name** (*str*) – name of profile
- **negotiation_expiration** (*int*) – expire after (ms)
- **negotiation_retry_timer** (*int*) – retry time length (ms)
- **negotiation_retry_max_num** (*int*) – max number of retries allowed
- **negotiation_retry_timer_max** (*int*) – maximum length for retry (ms)
- **certificate_cache_crl_validity** (*int*) – cert cache validity (seconds)
- **mobike_after_sa_update** (*boolean*) – Whether the After SA flag is set for Mobike Policy
- **mobike_before_sa_update** (*boolean*) – Whether the Before SA flag is set for Mobike Policy
- **mobike_no_rrc** (*boolean*) – Whether the No RRC flag is set for Mobike Policy

Raises ***CreateElementFailed*** – failed creating profile

Returns instance with meta

Return type *GatewaySettings*

13.11.5.2 GatewayNode

class smc.vpn.policy.**GatewayNode**(***meta*)

Bases: *smc.base.model.SubElement*

Top level VPN gateway node operations. A gateway node is characterized by a Central Gateway, Satellite Gateway or Mobile Gateway node. This template class will return these as a collection. Gateway Node references need to be obtained from a VPN Policy reference:

```
>>> vpn = PolicyVPN('sg_vm_vpn')
>>> vpn.open()
>>> for gw in vpn.central_gateway_node.all():
...   list(gw.enabled_sites)
...
[GatewayTreeNode(name=Automatic Site for sg_vm_vpn)]
>>> vpn.close()
```

disabled_sites

Return a collection of VPN Site elements that are disabled for this VPN gateway.

Return type *SubElementCollection*(*VPNSite*)

enabled_sites

Return a collection of VPN Site elements that are enabled for this VPN gateway.

Return type *SubElementCollection*(*VPNSite*)

name

Get the name from the gateway_profile reference

13.11.5.3 GatewayProfile

class smc.vpn.elements.**GatewayProfile**(*name*, ***meta*)
Bases: *smc.base.model.Element*

Gateway Profiles describe the capabilities of a Gateway, i.e. supported cipher, hash, etc. Gateway Profiles of Internal Gateways are read-only and computed from the firewall version and FIPS mode. Gateway Profiles of External Gateways are user-defined.

13.11.5.4 GatewayTreeNode

class smc.vpn.policy.**GatewayTreeNode**(***meta*)
Bases: *smc.base.model.SubElement*

Gateway Tree node is a list of VPN Site elements returned when retrieving a VPN policies enabled or disabled site list. These provide an enable_disable link to the VPN site.

```
for gw in policy.central_gateway_node.all():
    for site in list(gw.enabled_sites):
        site.enable_disable()
```

enable_disable()
Enable or disable this VPN Site from within the VPN policy context.

Raises *PolicyCommandFailed* – enabling or disabling failed

Returns None

vpn_site
The VPN Site element associated with this gateway

:return VPNSite element :rtype: VPNSite

13.11.5.5 GatewayTunnel

class smc.vpn.policy.**GatewayTunnel**(***meta*)
Bases: *smc.base.model.SubElement*

A gateway tunnel represents the point to point connection between two IPSEC endpoints in a PolicyVPN configuration. The tunnel arrangement is based on whether the nodes are placed as a central gateway or a satellite gateway. This provides access to see the point to point connections, whether the link is enabled, and setting the presharred key.

Note: Setting the preshared key is only required if using an ExternalGateway element as one side of the VPN. Preshared keys are generated automatically but read only, therefore if two gateways are internally managed by SMC, the key is generated and shared between the gateways automatically. However for external gateways, you must set a new key to provide the same value to the remote gateway.

---

enable_disable()
Enable or disable the tunnel link between endpoints.

Raises *UpdateElementFailed* – failed with reason

Returns None

enabled
Whether the VPN link between endpoints is enabled

Return type bool

preshared_key(*key*)
Set a new preshared key for the IPSEC endpoints.

Parameters **key** (*str*) – shared secret key to use

Raises ***UpdateElementFailed*** – fail with reason

Returns None

tunnel_side_a
Return the gateway node for tunnel side A. This will be an instance of GatewayNode.

Return type *GatewayNode*

tunnel_side_b
Return the gateway node for tunnel side B. This will be an instance of GatewayNode.

Return type *GatewayNode*

13.12 Collections Reference

Collections module provides interfaces to obtain resources from this API and provides searching mechanisms to autoload resources into the correct class type.

An ElementCollection is bound to *smc.base.model.Element* as the *objects* class property and provides the ability to use an element as the base for iterating elements of that type:

```
for hosts in Host.objects.all():
    ...
```

SubElementCollections are used when references to element data require a fetch from the SMC, but these element references do not have a direct SMC entry point.

See *Collections* for examples on search capabilities.

13.12.1 ElementCollection

class smc.base.collection.**ElementCollection**(***params*)
ElementCollection is generated dynamically from the CollectionManager and provides methods to obtain data from the SMC. Filters can be chained together to generate more complex queries. Each time a filter is added, a clone is returned to preserve the parent query parameters.

Chaining filters do not affect the parent iterator:

```
>>> iterator = Host.objects.iterator()    <-- Obtain iterator from␣
↪CollectionManager
>>> query1 = iterator.filter('10.10.10.1')
>>> query1._params, query1._iexact
({'filter': '10.10.10.1', 'exact_match': False, 'filter_context': 'router'}, None)
>>> query2 = query1.limit(2)
>>> query2._params, query2._iexact
({'filter': '10.10.10.1', 'exact_match': False, 'filter_context': 'router', 'limit
↪': 2}, None)
>>> query3 = query2.filter(address='10.10.10.1')
>>> query3._params, query3._iexact
({'filter': '10.10.10.1', 'exact_match': False, 'filter_context': 'router', 'limit
↪': 2}, {'address': '10.10.10.1'})
```

(continues on next page)

(continued from previous page)

Searcb operations can access a collection directly through chained syntax:

```
>>> for router in Router.objects.filter('192.168'):
...   print(router)
...
Router(name=router-192.168.19.241)
Router(name=router-192.168.21.241)
Router(name=router-192.168.5.241)
Router(name=router-192.168.15.241)
```

Adding additional filtering via kwargs:

```
>>> print(list(Router.objects.filter(address='10.10.10.1')))
[Router(name=Router-10.10.10.1)]
```

Checking if items from the query exist before accessing:

```
>>> query1 = iterator.filter('10.10.10.1')
>>> if query1.exists():
...   list(query1.all())
...
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
```

Helper methods `first`, `last` and `exists` are provided to simplify retrieving a result from the collection:

```
>>> query1 = iterator.filter('10.10.10.1')
>>> list(query1)
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
>>> query1.first()
Router(name=Router-110.10.10.10)
>>> query1.last()
Router(name=Router-10.10.10.1)
>>> query1.count()
3
>>> query2 = query1.filter(address='10.10.10.1')  # change filter to kwarg
>>> list(query2)
[Router(name=Router-10.10.10.1)]
```

Note: `exists` does not perform filtering when using `filter_key`. Results on filter(kwargs) are only done by retrieving the list of results or iterating.

---

all()

Retrieve all elements based on element type. When using the `all` option, any filters are automatically removed.

Returns *ElementCollection*

batch(*num*)

Iterator returning results in batches. When making more general queries that might have larger results, specify a batch result that should be returned with each iteration.

Parameters **num** (*int*) – number of results per iteration

Returns iterator holding list of results

count()
Return number of results

Return type int

exists()
Returns True if the query contains any results, and False if not. This is handy for checking existence without having to iterate.

```
>>> host = Host.objects.filter('1.1.1.1')
>>> if host.exists():
...   print(host.first())
...
Host(name=hax0r)
```

Return type bool

filter(**filter*, ***kw*)
Filter results for specific element type.

keyword arguments can be used to specify a match against the elements attribute directly. It's important to note that if the search filter contains a / or -, the SMC will only search the name and comment fields. Otherwise other key fields of an element are searched. In addition, SMC searches are a 'contains' search meaning you may return more results than wanted. Use a key word argument to specify the elements attribute and value expected.

```
>>> list(Router.objects.filter('10.10.10.1'))
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
>>> list(Router.objects.filter(address='10.10.10.1'))
[Router(name=Router-10.10.10.1)]
```

Parameters

- **filter** (*str*) – any parameter to attempt to match on. For example, if this is a service, you could match on service name 'http' or ports of interest, '80'.
- **exact_match** (*bool*) – Can be passed as a keyword arg. Specifies whether the match needs to be exact or not (default: False)
- **case_sensitive** (*bool*) – Can be passed as a keyword arg. Specifies whether the match is case sensitive or not. (default: True)
- **kw** – keyword args can specify an attribute=value to use as an exact match against the elements attribute.

Returns *ElementCollection*

first()
Returns the first object matched or None if there is no matching object.

```
>>> iterator = Host.objects.iterator()
>>> c = iterator.filter('kali')
>>> if c.exists():
>>>     print(c.count())
>>>     print(c.first())
```

(continues on next page)

(continued from previous page)

```
7
Host(name=kali67)
```

If results are not needed and you only 1 result, this can be called from the CollectionManager:

```
>>> Host.objects.first()
Host(name=SMC)
```

Returns element or None

last()

Returns the last object matched or None if there is no matching object.

```
>>> iterator = Host.objects.iterator()
>>> c = iterator.filter('kali')
>>> if c.exists():
>>>     print(c.last())
Host(name=kali-foo)
```

Returns element or None

limit(*count*)

Limit provides the ability to limit the number of results returned from the collection.

Parameters **count** (*int*) – number of records to page

Returns *ElementCollection*

class smc.base.collection.**CollectionManager**(*resource*)

CollectionManager takes a class type as input and dynamically creates an ElementCollection for that class. All classes of type Element have an *objects* property which returns a manager. You can consume the manager as a re-usable iterator or just called it and it's methods directly.

To get an iterator object that can be re-used, obtain an iterator() from the manager:

```
it = Host.objects.iterator()
it.filter(....)
...
```

Or more simply call the managers proxied methods to return the ElementCollection for the class type it was called for:

```
>>> from smc.elements.network import Host
>>> for host in Host.objects.all():
...   host
...
Host(name=IGMP v3)
Host(name=ALL-PIM-ROUTERS)
Host(name=Microsoft Lync Online Servers)
...
```

Returns *CollectionManager*

all()

Retrieve all elements based on element type. When using the `all` option, any filters are automatically removed.

Returns *ElementCollection*

batch (*num*)
Iterator returning results in batches. When making more general queries that might have larger results, specify a batch result that should be returned with each iteration.

Parameters **num** (*int*) – number of results per iteration

Returns iterator holding list of results

filter (**filter*, ***kw*)
Filter results for specific element type.

keyword arguments can be used to specify a match against the elements attribute directly. It's important to note that if the search filter contains a / or -, the SMC will only search the name and comment fields. Otherwise other key fields of an element are searched. In addition, SMC searches are a 'contains' search meaning you may return more results than wanted. Use a key word argument to specify the elements attribute and value expected.

```
>>> list(Router.objects.filter('10.10.10.1'))
[Router(name=Router-110.10.10.10), Router(name=Router-10.10.10.10),␣
↪Router(name=Router-10.10.10.1)]
>>> list(Router.objects.filter(address='10.10.10.1'))
[Router(name=Router-10.10.10.1)]
```

Parameters

- **filter** (*str*) – any parameter to attempt to match on. For example, if this is a service, you could match on service name 'http' or ports of interest, '80'.
- **exact_match** (*bool*) – Can be passed as a keyword arg. Specifies whether the match needs to be exact or not (default: False)
- **case_sensitive** (*bool*) – Can be passed as a keyword arg. Specifies whether the match is case sensitive or not. (default: True)
- **kw** – keyword args can specify an attribute=value to use as an exact match against the elements attribute.

Returns *ElementCollection*

first ()
Returns the first object matched or None if there is no matching object.

```
>>> iterator = Host.objects.iterator()
>>> c = iterator.filter('kali')
>>> if c.exists():
>>>     print(c.count())
>>>     print(c.first())
7
Host(name=kali67)
```

If results are not needed and you only 1 result, this can be called from the CollectionManager:

```
>>> Host.objects.first()
Host(name=SMC)
```

Returns element or None

iterator(***kwargs*)
Return an iterator from the collection manager. The iterator can be re-used to chain together filters, each chaining event will be it's own unique element collection.

Returns *ElementCollection*

limit(*count*)
Limit provides the ability to limit the number of results returned from the collection.

Parameters **count** (*int*) – number of records to page

Returns *ElementCollection*

13.12.2 SubElementCollection

class smc.base.collection.**SubElementCollection**(*href*, *cls*)
Collection class providing an iterable interface to sub elements referenced from a top level Element resource. Return types for this collection will be based on the class where the collection was obtained. Elements returned will be serialized into their Element types and only contain the top level meta for each element. The element cache will only be inflated (resulting in an additional query) if an operation is performed that requires the *data* (cache) attribute.

Helper methods are provided to simplify fetching from the collection without having to iterate and code the matching yourself. Fetching from the collection has the limitation that only the returned *name* field is used to find a match (to prevent inflating every element before it is needed). If you want to match an available attribute in the resulting class that requires the elements full json, use a loop to attempt your match.

Example of using SubElementCollection results to obtain matches from the collection:

```
>>> from smc.administration.system import System
>>> system = System()
>>> upgrades = system.engine_upgrade()
>>> upgrades
EngineUpgradeCollection(items: 29)
>>> list(upgrades)
[EngineUpgrade(name=Security Engine upgrade 6.1.2 build 17037 for x86-64),␣
↪EngineUpgrade(name=Security Engine upgrade 6.2.3 build 18067 for x86-64),  ....]
>>> upgrades.get(5)
EngineUpgrade(name=Security Engine upgrade 5.8.8 build 12093 for i386)
>>> upgrades.get_contains('6.2')
EngineUpgrade(name=Security Engine upgrade 6.2.3 build 18067 for x86-64)
>>> upgrades.get_contains('6.1')
EngineUpgrade(name=Security Engine upgrade 6.1.2 build 17037 for x86-64)
>>> upgrades.get_all_contains('6.2')
[EngineUpgrade(name=Security Engine upgrade 6.2.3 build 18067 for x86-64),␣
↪EngineUpgrade(name=Security Engine upgrade 6.2.2 build 18062 for x86-64), ...]
>>>
```

Raises ***FetchElementFailed*** – If the resource could not be retrieved

all()
Generator returning collection for sub element types. Return full contents as list or iterate through each.

Returns element type based on collection

Return type list(*SubElement*)

count()
Return the number of results in this collection

Returns int

get (*index*)
Get the element by index. If index is out of bounds for the internal list, None is returned. Indexes cannot be negative.

Parameters **index** (*int*) – retrieve element by positive index in list

Return type *SubElement* or None

get_all_contains (*value*, *case_sensitive=True*)
A partial match on the name field. Does an *in* comparsion to elements by the meta *name* field. Returns all elements that match the specified value.

See also:

get_contains() for returning only a single item.

Parameters

- **value** (*str*) – searchable string for contains match
- **case_sensitive** (*bool*) – whether the match should consider case (default: True)

Returns element or empty list

Return type list(*SubElement*)

get_contains (*value*, *case_sensitive=True*)
A partial match on the name field. Does an *in* comparsion to elements by the meta *name* field. Sub elements created by SMC will generally have a descriptive name that helps to identify their purpose. Returns only the first entry matched even if there are multiple.

See also:

get_all_contains() to return all matches

Parameters

- **value** (*str*) – searchable string for contains match
- **case_sensitive** (*bool*) – whether the match should consider case (default: True)

Return type *SubElement* or None

get_exact (*value*)
Get an element using an exact match based on the elements meta *name* field. The SMC is case sensitive so the name will need to honor the case for a valid value match.

See also:

get_contains() and *get_all_contains()* for partial matching

Parameters **value** (*str*) – name to match

Return type *SubElement* or None

13.12.2.1 CreateCollection

class smc.base.collection.**CreateCollection** (*href*, *cls*)
Bases: *smc.base.collection.SubElementCollection*

A CreateCollection extends SubElementCollection by dynamically proxying the elements *create* method into the collection. This provides a simplified way to create sub elements and also iterate through existing.

For example, obtaining VPN Sites from an engine returns a CreateCollection so existing sites can be iterated while still being able to create new sites:

```
>>> engine = Engine('dingo')
>>> print(engine.vpn.sites)
<smc.base.collection.VPNSite object at 0x1098a9ed0>
>>> print(help(engine.vpn.sites))
Help on VPNSite in module smc.base.collection object:

class VPNSite(CreateCollection)
 |  Method resolution order:
 |      VPNSite
 |      CreateCollection
 |      SubElementCollection
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  create(self, name, site_element) from smc.vpn.elements.VPNSite
 |      Create a VPN site for an internal or external gateway
 |
 |      :param str name: name of site
 |      :param list site_element: list of protected networks/hosts
 |      :type site_element: list[str,Element]
 |      :raises CreateElementFailed: create element failed with reason
 |      :return: href of new element
 |      :rtype: str
 |
 ....
```

List existing sites:

```
list(engine.vpn.sites.all())
```

Creating new VPN sites:

```
engine.vpn.sites.create('mynewsite')
```

create(**args*, ***kwargs*)

The create function from the sub element is proxied by this collections class to provide the iterable functionality from the parent container, but also protected access to the create method of the instance.

13.12.2.2 RuleCollection

smc.base.collection.**rule_collection**(*href*, *cls*)

Rule collections insert a `create` and `create_rule_section` method into the collection. This collection type is returned when accessing rules through a reference, as:

```
policy = FirewallPolicy('mypolicy')
policy.fw_ipv4_access_rules.create(....)
policy.fw_ipv4_access_rules.create_rule_section(...)
```

See the class types documentation, or use help():

```
print(help(policy.fw_ipv4_access_rules))
```

Return type *SubElementCollection*

13.12.3 Search

class smc.base.collection.**Search**(**params*)

Bases: *smc.base.collection.ElementCollection*

Changed in version 0.5.6: Added entry_point and context_filter chaining to make search syntax the same as direct element object searches.

Search extends ElementCollection and provides a way to search for any object by type, as long as there is a valid entry point. Syntax for general searches are the same as initializing a search by a specific element:

```
Search.object_types()    # Get all available search entry points
...
Search.objects.entry_point('ips_alert') # Search for IPS Alerts
...
Search.objects.entry_point('network').filter('1.1.1') # Network with filter
...
Search.objects.context_filter('engine_clusters') # by context filter
...
Search.objects.filter('2.2.2.2') # All element types with filter
...
Search.objects.entry_point('router,host')) # Search using multiple element types
...
Search.objects.entry_point('router,host').filter('2.2.2.2') # with filter
```

Search also provides convenience shortcuts to find duplicate and unused elements:

```
Search.objects.unused()
...
Search objects.duplicates()
```

If searching a broad range of elements, it is advisable to return results in batches:

```
for batch in Search.objects.batch(100): # All elements search
    ...
```

Note: If no entry point is specified, the search is done at the 'elements' entry point which contains all SMC elements. It is recommended to use `filter` and possibly `batch` to control the result set.

---

context_filter(*context*)

Provide a context filter to search.

Parameters **context** (*str*) – Context filter by name

duplicates()

Return duplicate user-created elements.

Return type list(*Element*)

entry_point(*entry_point*)

Provide an entry point for element types to search.

Parameters **entry_point** (*str*) – valid entry point. Use *~object_types()* to find all available entry points.

static object_types()
Show all available 'entry points' available for searching. An entry point defines a uri that provides unfiltered access to all elements of the entry point type.

Returns list of entry points by name

Return type list(str)

unused()
Return unused user-created elements.

Return type list(*Element*)

13.12.4 BaseIterable

Common structures

class smc.base.structs.**BaseIterable**(*items*)
A collections container that provides a pre-filled container. This container type is used when an element retrieval returns all of an elements data in a single query and will contain multiple values for the same serialized type. Elements can be retrieved from the container through iteration, slicing, or by using *get* and providing either the index or an attribute / value pair.

If subclassing, it may be useful to override *get* to provide a restricted interface to common attributes to fetch.

Examples:

```
>>> for status in engine.nodes[0].interface_status:
...   status
...
InterfaceStatus(aggregate_is_active=False, ....
```

By index:

```
>>> engine.nodes[0].interface_status[1]
```

Slicing:

```
>>> engine.nodes[0].interface_status[1:5:2]
>>> engine.nodes[0].interface_status[::-1]
```

Using get by index or attribute:

```
>>> engine.nodes[0].interface_status.get(1)
>>> engine.nodes[0].interface_status.get(interface_id=2)
```

Parameters **item** (*iterable*) – items for which to perform iteration. Can be another class with an __iter__ method also to chain iterators.

all()
Return the iterable as a list

count()
Return the number of entries

Return type int

get (**args*, ***kwargs*)
Get an element from the iterable by an arg or kwarg. Args can be a single positional argument that is an index value to retrieve. If the specified index is out of range, None is returned. Otherwise use kwargs to provide a key/value. The key is expected to be a valid attribute of the iterated class. For example, to get an element that has a attribute name of 'foo', pass name='foo'.

Raises **ValueError** – An argument was missing

Returns the specified item, type is based on what is returned by this iterable, may be None

13.12.5 SerializedIterable

class smc.base.structs.**SerializedIterable**(*items*, *model*)
Bases: *smc.base.structs.BaseIterable*

A pre-serialized list of elements. This is used when it's easier to provide a pre-serialized class as long as all elements are of the same type.

Parameters

- **item** (*iterable*) – items for which to perform iteration. Can be another class with an __iter__ method also to chain iterators.
- **model** – optional class to serialize each iteration.

13.13 Advanced Usage

13.13.1 SMCRequest

Middle tier helper module to wrap CRUD operations and catch exceptions

SMCRequest is the general data structure that is sent to the send_request method in smc.api.web.SMCConnection to submit the data to the SMC.

class smc.api.common.**SMCRequest**(*href=None*, *json=None*, *params=None*, *filename=None*, *etag=None*, *user_session=None*, ***kwargs*)
SMCRequest represents the data structure that will be submitted to the web layer for submission to the SMC API.

Parameters

- **href** (*str*) – href for request, required by all methods
- **json** (*dict*) – json to submit, required by create, update
- **params** (*dict*) – query string parameters
- **filename** (*str*) – name of file for download, optional for create
- **etag** (*str*) – etag of element, required for update

etag = None
ETag for PUT or DELETE request modifications

filename = None
Filename if a file download is requested

headers = None
Default headers

href = None
href for this request

json = None
JSON data to send in request

params = None
dictionary of query parameters

13.13.2 SMCResult

Operations being performed that involve REST calls to SMC will return an SMCResult object. This object will hold attributes that are useful to determine if the operation was successful and if not, the reason. An SMCResult is handled automatically and uses exceptions to provide statuses between modules and user interaction. The simplest way to get access to an SMCResult directly is to make an SMCRequest using `smc.base.model.prepared_request()` and observe the attributes in the return message. All response data is serialized into the SMCResult.json attribute when it is received by the SMC.

Web actions to SMC

SSL certificates are not verified to the CA authority, need to implement for urllib3: https://urllib3.readthedocs.io/en/latest/user-guide.html#ssl

class smc.api.web.**SMCResult**(*respobj=None*, *msg=None*, *user_session=None*)
SMCResult will store the return data for operations performed against the SMC API. If the function returns an SMCResult, the following attributes are set. Note: SMC API will return a list if searches are done and a dict if the attempt is made to get the element directly from href.

Instance attributes:

Variables

- ***etag*** (*str*) – etag from HTTP GET, representing unique value from server
- ***href*** (*str*) – href of location header if it exists
- **content** (*str*) – content if return was application/octet
- **msg** (*str*) – error message, if set
- **code** (*int*) – http code
- ***json*** (*dict*) – element full json

Example of using SMCRequest to fetch an element by href, returning an SMCResult:

```
>>> vars(SMCRequest(href='http://1.1.1.1:8082/6.2/elements/host/978').read())
{'code': 200, 'content': None, 'json': {u'comment': u'this is a searchable comment', u
↪'read_only': False, u'ipv6_address': u'2001:db8:85a3::8a2e:370:7334', u'name': u
↪'kali', u'third_party_monitoring': {u'netflow': False, u'snmp_trap': False}, u
↪'system': False, u'link': [{u'href': u'http://1.1.1.1:8082/6.2/elements/host/978', u
↪'type': u'host', u'rel': u'self'}, {u'href': u'http://1.1.1.1:8082/6.2/elements/
↪host/978/export', u'rel': u'export'}, {u'href': u'http://1.1.1.1:8082/6.2/elements/
↪host/978/search_category_tags_from_element', u'rel': u'search_category_tags_from_
↪element'}], u'key': 978, u'address': u'1.1.11.1', u'secondary': [u'7.7.7.7']}, 'href
↪': None, 'etag': '"OTc4MzExMzkxNDk2MzI1MTMyMDI4"', 'msg': None}
```

13.14 Waiters

Waiters are convenience classes that use blocking or non-blocking threads to monitor for a particular state of an engine node.

A waiter can have a callback added that will be executed after either the state has matched, a number of iterations exceeded or an exception is caught while monitoring. The callback should be a callable that takes a single argument.

They provide the ability to perform logical actions such as "wait for the engine to have status 'Configured', then fire a policy upload task".

Example of waiting for an engine to be ready, then send policy:

```
class ContainerPolicyCallback(object):
    def __init__(self, container):
        self.engine = engine

    def __call__(self, status):
        if status == 'Configured':
            self.engine.upload(policy='MyPolicy')

engine = Engine('myengine')
callback = ContainerPolicyCallback(engine)

waiter = ConfigurationStatusWaiter(engine.nodes[0], 'Configured')
waiter.add_done_callback(callback)
```

Waiters can also be blocking while waiting for status. Example of using a waiter to block input while waiting for the engine to reach a specific status:

```
waiter = ConfigurationStatusWaiter(node, 'Initial', max_wait=5)
while not waiter.done():
    print("Status after 5 sec wait: %s" % waiter.result(5))
```

smc.core.waiters.**CFG_STATUS = frozenset(['Declared', 'Initial', 'Configured', 'Installed'])**
Configuration status constant values

class smc.core.waiters.**ConfigurationStatusWaiter**(*resource*, *status*, ***kw*)
Bases: *smc.core.waiters.NodeWaiter*

Configuration status waiter provides a current engine status with respects to having a configuration.

Parameters

- **resource** (Node) – Engine node to check for status
- **status** (*str*) – used defined status to wait for.

Raises ***NodeCommandFailed*** – Failure to obtain a status back from the engine. This can be thrown when getting initial status. If thrown after the thread has started, it is caught and returned in the `result` after ending the thread.

class smc.core.waiters.**NodeStateWaiter**(*resource*, *status*, ***kw*)
Bases: *smc.core.waiters.NodeWaiter*

Node State specifies where the engine is within it's lifecycle, such as initial state, ready state, error, timeout, etc.

Parameters

- **resource** (Node) – Engine node to check for status
- **status** (*str*) – used defined status to wait for.

Raises ***NodeCommandFailed*** – Failure to obtain a status back from the engine. This can be thrown when getting initial status. If thrown after the thread has started, it is caught and returned in the `result` after ending the thread.

class smc.core.waiters.**NodeStatusWaiter**(*resource*, *status*, ***kw*)
Bases: *smc.core.waiters.NodeWaiter*

Node Status specifies the current state of the engine such as offline, online, locked offline, no policy installed, etc.

Parameters

- **resource** (Node) – Engine node to check for status
- **status** (*str*) – used defined status to wait for.

Raises ***NodeCommandFailed*** – Failure to obtain a status back from the engine. This can be thrown when getting initial status. If thrown after the thread has started, it is caught and returned in the `result` after ending the thread.

class smc.core.waiters.**NodeWaiter**(*resource*, *status*, *timeout=5*, *max_wait=36*, ***kw*)
Bases: threading.Thread

Node Waiter provides a common threaded interface to monitoring a nodes status and wait for a specific response.

add_done_callback(*callback*)
Add a callback to run after the task completes. The callable must take 1 argument which will be the completed Task.

:param callable callback

done()
Is the task still running or considered complete

Return type bool

result(*timeout=None*)
Get current status result after waiting timeout Result does a join on the thread to get a status update. It is possible the first couple of statuses are None if an update has not yet been joined.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Stop thread if it's still running

wait(*timeout=None*)
Blocking method to wait for thread

smc.core.waiters.**STATE = frozenset(['DUMMY', 'SERVER_ERROR', 'DELETED', 'INITIAL', 'TIMEOU**
Node state constant values

smc.core.waiters.**STATUS = frozenset(['Going Locked Offline', 'Locked Online', 'Policy Out C**
Node status constant values

13.15 Exceptions

Exceptions thrown throughout smc-python. Be sure to check functions or class methods that have raises documentation. All exception classes subclass SMCException

Exceptions Module

exception smc.api.exceptions.**ActionCommandFailed**
Bases: *smc.api.exceptions.SMCException*

Action type commands use this exception

exception smc.api.exceptions.**CertificateError**
Bases: *smc.api.exceptions.SMCException*

Related to certificate based operations like requests, signing, or creation. For example, engines that are not initialized can not respond to certificate creation requests and SMC API will return an error.

exception smc.api.exceptions.**CertificateExportError**
Bases: *smc.api.exceptions.CertificateError*

Failure to export a certificate

exception smc.api.exceptions.**CertificateImportError**
Bases: *smc.api.exceptions.CertificateError*

Failure to import a certificate or private key

exception smc.api.exceptions.**ConfigLoadError**
Bases: *smc.api.exceptions.SMCException*

Thrown when there was a problem reading credential information from file. Typically caused by missing settings.

exception smc.api.exceptions.**CreateElementFailed**
Bases: *smc.api.exceptions.SMCException*

Generic exception when there was a failure calling a create method

exception smc.api.exceptions.**CreateEngineFailed**
Bases: *smc.api.exceptions.SMCException*

Thrown when a POST operation returns with a failed response. API based response will be returned as the exception message

exception smc.api.exceptions.**CreatePolicyFailed**
Bases: *smc.api.exceptions.SMCException*

Thrown when failures occur when creating specific poliies like Firewall Policy, IPS, VPN, etc.

exception smc.api.exceptions.**CreateRuleFailed**
Bases: *smc.api.exceptions.SMCException*

Indicates a failed response when creating a rule of any type.

exception smc.api.exceptions.**CreateVPNFailed**
Bases: *smc.api.exceptions.SMCException*

Creating a policy or route based VPN failed.

exception smc.api.exceptions.**DeleteElementFailed**
Bases: *smc.api.exceptions.SMCException*

Used when deletion fails, typically due to dependencies for the target element

exception smc.api.exceptions.**ElementNotFound**
Bases: *smc.api.exceptions.SMCException*

Generic exception when an attempt is made to load an element that is not found.

exception smc.api.exceptions.**EngineCommandFailed**
Bases: *smc.api.exceptions.SMCException*

Engines will have some commands that are specifically executed such as adding blacklist entries, flushing blacklist or adding routes. This exception will be thrown if the SMC API responds with any sort of error and wrap the response

exception smc.api.exceptions.**FetchElementFailed**
Bases: *smc.api.exceptions.SMCException*

Failure when fetching results

exception smc.api.exceptions.**InterfaceNotFound**
Bases: *smc.api.exceptions.SMCException*

Returned when attempting to fetch an interface directly

exception smc.api.exceptions.**InvalidRuleValue**
Bases: *smc.api.exceptions.SMCException*

Used within rule creation methods to prevent invalid submissions

exception smc.api.exceptions.**InvalidSearchFilter**
Bases: *smc.api.exceptions.SMCException*

Thrown by collections when using invalid search sequences.

exception smc.api.exceptions.**LicenseError**
Bases: *smc.api.exceptions.SMCException*

Thrown when operations to perform Node specific license related operations such as bind license, fetch license or cancel license fail. For node licensing specific actions, see: :py:class: *smc.core.node.Node*

exception smc.api.exceptions.**LoadElementFailed**
Bases: *smc.api.exceptions.SMCException*

Failure when attempting to obtain the settings for a specific element. This is more generic for a broad class of elements.

exception smc.api.exceptions.**LoadEngineFailed**
Bases: *smc.api.exceptions.SMCException*

Thrown when attempting to load an engine that does not exist

exception smc.api.exceptions.**LoadPolicyFailed**
Bases: *smc.api.exceptions.SMCException*

Failure when trying to load a specific policy type

exception smc.api.exceptions.**MissingDependency**
Bases: *smc.api.exceptions.SMCException*

A dependency is missing for the given operation.

exception smc.api.exceptions.**MissingRequiredInput**
Bases: *smc.api.exceptions.SMCException*

Some functinos will flat out fail if certain fields are not provided. This is to ensure that some functions have some protection in case the user doesn't read the doc's.

exception smc.api.exceptions.**ModificationAborted**
Bases: *smc.api.exceptions.SMCException*

A previous requirement was not met which prevented an attempted change from being executed.

exception smc.api.exceptions.**ModificationFailed**
Bases: *smc.api.exceptions.SMCException*

Used when making generic modifications to elements.

exception smc.api.exceptions.**NodeCommandFailed**
Bases: *smc.api.exceptions.SMCException*

Each engine node will have multiple commands that can be executed such as go online, go offline, go standby, locking, etc. When these commands fail, this exception will be thrown and wrap the SMC API response. For all node specific command actions, see: :py:class: *smc.core.node.Node*

exception smc.api.exceptions.**PolicyCommandFailed**
Bases: *smc.api.exceptions.SMCException*

Generic policy related command failures such as opening or closing a VPN policy.

exception smc.api.exceptions.**ResourceNotFound**
Bases: *smc.api.exceptions.SMCException*

Used to indicate a resource link is not found on the queried node. For example, the *smc.core.engine.Engine* class will expose available resources but some engines may not have those links.

exception smc.api.exceptions.**SMCConnectionError**
Bases: *smc.api.exceptions.SMCException*

Thrown when there are connection related issues with the SMC. This could be that the underlying http requests library could not connect due to wrong IP address, wrong port, or time out

exception smc.api.exceptions.**SMCException**
Bases: exceptions.Exception

Base class for exceptions

exception smc.api.exceptions.**SMCOperationFailure**(*response=None*)
Bases: *smc.api.exceptions.SMCException*

Exception class for storing results from calls to the SMC This is thrown for HTTP methods that do not return the expected HTTP status code. See each http_* method in *smc.api.web* for expected success status

Parameters

- **response** – response object returned from HTTP method
- **msg** – optional msg to insert

Instance attributes:

Variables

- **response** – http request response object
- **code** – http status code
- ***status*** – status from SMC API
- **message** – message attribute from SMC API
- **details** – details list from SMC API (may not always exist)
- **smcresult** – *smc.api.web.SMCResult* object for consistent returns

exception smc.api.exceptions.**SessionManagerNotFound**(*message=''*)
Bases: exceptions.Exception

exception smc.api.exceptions.**SessionNotFound**
Bases: *smc.api.exceptions.SMCException*

Retrieving a session by name did not succeed because the session did not already exist

exception smc.api.exceptions.**TaskRunFailed**
Bases: *smc.api.exceptions.SMCException*

When running tasks such as policy upload, refresh policy, etc, if the result from SMC is a failure, possibly due to an incorrect input (i.e. missing policy), then this exception will be thrown

exception smc.api.exceptions.**UnsupportedEngineFeature**
Bases: *smc.api.exceptions.SMCException*

If an operation is performed on an engine that does not support the functionality, this is thrown. For example, only Master Engine has virtual resources. IPS and Layer 2 Firewall do not have internal gateways (used for VPN).

exception smc.api.exceptions.**UnsupportedEntryPoint**
Bases: *smc.api.exceptions.SMCException*

An entry point was specified that was not found in this API version. This is likely due to using an older version of the SMC API that does not support that feature. The exception is thrown specifying the entry point specified.

exception smc.api.exceptions.**UnsupportedInterfaceType**
Bases: *smc.api.exceptions.SMCException*

Some interface types are not supported on certain engines. For example, Virtual Engines only have Virtual Physical Interfaces. Layer 3 Firewalls do not support Capture or Inline Interfaces. This exception will be thrown when an attempt is made to enumerate interfaces for an engine type missing a reference to an unsupported interface type

exception smc.api.exceptions.**UpdateElementFailed**
Bases: *smc.api.exceptions.SMCException*

Failure when updating element. When failure is due to ETag being invalid, target was modified before change was submitted. A resubmit would be required.

exception smc.api.exceptions.**UserElementNotFound**
Bases: *smc.api.exceptions.SMCException*

Raised when attempting to find a user element that cannot be found in a mapped database (internal or external LDAP)

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z