



FORCEPOINT

NGFW SMC API

for Forcepoint Next Generation Firewall

Reference Guide

6.2

Revision A

Contents

- [Introduction](#) on page 2
- [Configure SMC API](#) on page 4
- [Working with RESTful principles](#) on page 6
- [Entry point structure](#) on page 9
- [Data element formats](#) on page 12
- [Working with resource elements](#) on page 14
- [Specific searches](#) on page 15
- [Examples](#) on page 16

Introduction

Forcepoint™ NGFW Security Management Center (SMC) is part of the Forcepoint™ Next Generation Firewall (Forcepoint NGFW) solution. These products were formerly known as Stonesoft Management Center (SMC) and Stonesoft Next Generation Firewall (Stonesoft NGFW). You can access the SMC in two ways: through the Management Client or through the SMC application programming interface (API).

This guide describes how to enable the SMC API and provides examples of its use. The target audience for this guide includes system administrators and developers.

The SMC API provides functions for adding, editing, and deleting elements in the Management Server database.

General use cases that are supported through the SMC API include the following:

- Adding, editing, and removing simple elements (such as Hosts, Networks, and Address Ranges)
- Adding, editing, and removing Access rules, NAT rules, and Inspection rules
- Uploading a policy to an engine
- Retrieving or changing the routing of an engine
- Configuring VPNs

A Python-based library that provides basic functions for interacting with the SMC API is available. See <https://github.com/gabstopper/smc-python>. The library is not supported by Forcepoint.

Although the SMC API follows the architectural style of RESTful web APIs, this guide introduces only the basic concepts. For more information, see:

- **Representational state transfer** — https://en.wikipedia.org/wiki/Representational_state_transfer and the linked content
- **Resources and actions** — Available in the complete generated API documentation in the documentation folder of the SMC installation files

Example use cases

These scenarios highlight the ways you can use the SMC API.

- Integrate the SMC with third-party policy management and risk management applications. The SMC API is already used by vendors such as Tufin and FireMon.
- Provide the necessary tools for managed security service providers (MSSPs) to include functions related to Management Servers and Log Servers on their own web portals.
- Automate frequent tasks through scripting without administrators manually configuring them in the Management Client.
- Develop an alternative user interface for managing Management Servers and Log Servers.

User session identification

The SMC API supports two methods for associating all requests with a single user between the logon and logoff actions.

- **Cookies** — The API Client sends back in each request all (non-expired) cookies that the server sent.
- **SSL Sessions** — Sessions are tracked by the server based on SSL connections.



Note: Cookies are the default. If you want to use SSL Sessions instead, you must enable it.

To save server resources, clients should log off at the end of the session.

Related tasks

[Enable SMC API](#) on page 5

Adjust the HTTP session inactivity timeout

You can change the HTTP session inactivity timeout.

By default, the HTTP session inactivity timeout is 30 minutes. After 30 minutes of inactivity, the SMC API prompts you to log on again to be able to execute any new HTTP requests.

Steps

- 1) On the computer where the Management Server is installed, browse to the <installation directory>/data directory.
- 2) Edit the SGConfiguration.txt file and add this parameter: WEB_SERVER_SESSION_TIMEOUT=<time in minutes>.
- 3) Save the SGConfiguration.txt file.

Backward compatibility

Backward compatibility is guaranteed to the previous major SMC release and also SMC version 5.10.

For example, SMC 6.2 provides access to the 6.1 and 6.2 versions of the SMC API, from two version-specific URIs, and also to the 5.10 version.

Limitations

Some limitations apply when using the SMC API.

Import synchronization — During the import or update package activation task, all requests to update a resource are delayed until the task is completed. This synchronization is necessary to avoid data integrity failure during the import.

Configure SMC API

The application programming interface (API) of SMC allows external applications to connect with the SMC.



Note: If there is a firewall between the SMC and external applications, make sure that there is an Access rule to allow communication.


The SMC API can be used to run actions remotely using an external application or script.


Create TLS credentials for SMC API Clients

If you want to use encrypted connections, the SMC API Client needs TLS credentials to connect with the Management Server.



Note: You can import the existing private key and certificate if they are available.

Steps  For more details about the product and how to configure features, click **Help** or press **F1**.

- 1) In the Management Client, select  **Configuration**, then browse to **Other Elements > Engine Properties > Certificates > Pending Certificate Requests**.
- 2) Right-click **Pending Certificate Requests**, then select **New Pending Certificate Request**.
- 3) Complete the certificate request details.
 - a) Enter a name and add "REQ" or "request" at the end.
 - b) In the **Common Name** field, enter the IP address or domain name of SMC.
 - c) Complete the remaining fields as needed.

- d) Click **OK**.
- 4) Right-click the completed request, then select **Self Sign**.
- 5) Enter the name from the previous step, removing "REQ" or "request" from the end.
- 6) Click **OK**.
- 7) When prompted, remove the pending Certificate Request.

Result

The TLS Credentials element is added to **Other Elements > Engine Properties > Certificates > TLS Credentials**.

Enable SMC API

To allow other applications to connect using the SMC API, enable SMC API on the Management Server.

Steps • For more details about the product and how to configure features, click **Help** or press **F1**.

- 1) In the Management Client, select **Home**.
- 2) Browse to **Others > Management Server**.
- 3) Right-click the Management Server, then select **Properties**.
- 4) Click the **SMC API** tab, then select **Enable**.
- 5) (Optional) In the **Host Name** field, enter the name that the SMC API service uses.



Note: API requests are served only if the API request is made to this host name. To allow API requests to any host name, leave this field blank.

- 6) Make sure that the listening port is set to the default of 8082 on the Management Server.
- 7) If the Management Server has several IP addresses and you want to restrict access to one, enter the IP address in the **Listen Only on Address** field.
- 8) If you want to use encrypted connections, click **Select**, then select the TLS Credentials element.
- 9) Click **OK**.


Create an API Client element

External applications use API clients to connect to SMC.

Before you begin

SMC API must be enabled for the Management Server.

Steps  For more details about the product and how to configure features, click **Help** or press **F1**.

- 1) Select  **Configuration**, then browse to **Administration**.
- 2) Browse to **Access Rights**.
- 3) Right-click **Access Rights**, then select **New > API Client**.
- 4) In the **Name** field, enter a unique name for the API Client.
- 5) Use the initial authentication key or click **Generate Authentication Key** to generate a new one. A random 24-character alphanumeric authentication key is automatically generated.



Important: This key appears only once, so be sure to record it. The API Client uses the authentication key to log on to SMC API.

- 6) Click the **Permissions** tab.
- 7) Select the permissions for actions in the SMC API.
- 8) Click **OK**.

Working with RESTful principles

The SMC API is a RESTful API that includes these features.

- The API is strictly based on the HTTP protocol and is platform-independent.
- Each resource is identified by a unique URI, which is opaque to the API clients.
- URIs and actions that can be performed on resources are accessible through hyperlinks.
- The SMC API supports multiple representations for each resource. Currently, only JSON and XML, or plain text when it is understandable, are supported.
- ETags are used for cacheability and conditional updates.

Requests

There are several types of requests and they affect resources in different ways.

You can perform these actions:

- Create resources by using POST requests on the URI of the collection that lists all elements. The URI of the created resource is returned in the **Location** header field.
- Read resources by using GET requests. To save network bandwidth and avoid transferring the complete body in the response, HEAD requests and ETags are supported.
- Update resources by using PUT requests. All updates are conditional and rely on ETags.
- Delete resources by using DELETE requests.
- Trigger actions such as Policy Uploads by using POST requests.



Important: Only GET, HEAD, and OPTIONS requests are safe and do not have side effects. PUT and DELETE requests have no additional effect if they are called more than once with the same parameters, but We recommend avoiding redundant requests. POST requests might have additional effects if called more than once with the same input parameters; clients are responsible for avoiding multiple POST requests.

Status codes and error messages

When requests are made, they return HTTP response status codes.

For more information, the HTTP response status codes follow the principles outlined in https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

For an error message, the server also attempts to send relevant information in the response body.

ETags

When a GET request is made, it returns an ETag in the header that verifies the version of the returned data.

ETags are used for:

- **Caching purposes** — When performing GET requests, the client sends back the ETags of the last fetched data in an If-None-Match header. If the ETag has not changed, the server returns a 304 HTTP response status code (Not Modified).
- **Conditional updates** — Whenever a client attempts to update an element, an If-Match header must be sent with the ETag of the last retrieved data. If this data is changed in the meantime, the server returns a 412 HTTP response status code (Precondition Failed).

The concurrency of the HTTP requests is based on the same model as the concurrency of the heavyweight clients, so several HTTP sessions can be handled at the same time by the API server. For this reason, the ETag header is mandatory for all PUT requests. It ensures that you are updating the latest version of the element.

For more information, see https://en.wikipedia.org/wiki/HTTP_ETag.

304 status code handling

The SMC API supports the 304 Not Modified Error status code. This response indicates that the resource has not been modified since the version specified by the If-Modified-Since or If-None-Match request headers.

There is no need to retransmit the resource because the requested element has not been modified and the client still has a previously downloaded copy. For example, if you add your ETag element version to the ETag parameter in the header of your GET request, the SMC API returns a 304 status code instead of the element content. This response indicates that your downloaded copy of the element is the same as the requested element.

503 status code handling

The SMC API supports the 503 Service Unavailable status code. This response indicates that the server is currently unable to handle the request because it is running a system task that is accessing the same resource.

The implication is that this is a temporary condition and the request can be retried after some delay.

Opaque URIs, URI discovery, and hypermedia

All URIs must be considered opaque values; clients should not have to construct URIs by concatenating substrings.

All URIs must be recursively discovered from:

- The API entry-point URI, as defined in the SMC API configuration
- Top-level service URIs
- The top-level lists of elements
- Links to other resources that are mentioned in these elements
- The action links identified by the verbs (for example, upload) that are mentioned in these elements

Some URIs support or mandate the use of additional query parameters, for example, for filtering purposes.

Body content and query parameters

REST operations contain specific content or support additional parameters.

- Create and update operations require content in the body of the request.
- Read and delete operations on a single element do not require any additional content.
- Element listing operations support filtering arguments as query parameters.
- Some action URIs require additional parameters.

Entry point structure

Each entry point contains an operation verb and other information to direct the user to a URI.

Verbs

Verbs represent keywords for specific element operations.

Verbs are listed in the XML/JSON element description as a link entry.

Self verbs

The self verb is included in each element and is based on REST philosophy. The self verb allows you to retrieve the API URI for the current element.

Example: For a host, the self verb might look like the following.

```
"link":
[
  {
    "href": "http://localhost:8082/6.2/elements/host/86",
    "rel": "self",
    "type": "host"
  }
]
```

A link entry has the following structure:

- href: The API's URI to the associated verb
- rel: The keyword that is preserved beyond SMC versions. It represents the verb
- type: Optional information about the return type

Installing a policy from a Policy element

The upload verb is present on several Policy elements, such as fw_policy.

Example: The JSON description of a policy refers to the verb in this way:

```
"link":
[
  {
    "href": "http://localhost:8082/6.2/elements/fw_policy/56/upload",
    "rel": "upload"
  },
  ...
]
```

This verb can be found in each policy type.

Example: Here it is shown in the Firewall Policy. It presents a query parameter — a filter that can be uploaded on a specific engine (?filter=TheEngineName).

This verb starts the upload of the specific policy on the specified engine. It returns a 200 HTTP response status code and an upload status description similar to the following:

```
{
  "follower": "http://localhost:8082/6.2/elements/fw_policy/56/upload/
NWYyMDBiOTA4ZTY3NDM0ZTo0YzYzM2ZTg5MDoxM2ZlNzhhMDZlZTotN2VhZA==",
  "href": "http://localhost:8082/6.2/elements/fw_policy/56",
  "in_progress": true,
  "last_message": "",
  "success": true
}
```

The upload status has the following structure:

- follower: The API's URI to the current upload status
- href: The source of the upload (in this example, the policy)
- in_progress: A flag that shows whether the upload is still in progress
- last_message: The last upload status message
- success: A flag that shows whether the current upload has succeeded

Uploading the engine configuration from an engine element

The upload verb is present on each engine element, such as `single_fw`.

Example: the JSON description of an engine refers to the verb in the following way:

```
"link":
[
  ...
  {
    "href": "http://localhost:8082/6.2/elements/single_fw/1552/upload",
    "rel": "upload"
  },
  ...
],
```

This verb can be found in each engine type. In this example, it is shown in a Single Firewall. It presents a query parameter — a filter that can be uploaded on a specific policy (`?filter=ThePolicyName`).

This verb starts the upload on the specific engine and the specified policy. It returns a 200 HTTP response status code and a similar upload status description as for the policy upload.

Refreshing the engine configuration from an engine element

You can use the `refresh` verb to refresh the policy from engine elements.

Example: The JSON description of an engine refers to the verb in the following way:

```
"link":
[
  ...
  {
    "href": "http://localhost:8082/6.2/elements/single_fw/1552/refresh",
    "rel": "refresh"
  },
  ...
],
```

This verb can be found in each engine type. In this example, it is shown in a Single Firewall.

This verb starts the policy refresh of the specific engine if a policy has already been installed on the engine. It returns a 200 HTTP response status code and a similar upload status description as for the policy upload.

API discovery

To ease migration during major version upgrades, it is preferable to discover the API starting from an entry-point rather than to define all URIs.

Verbs do not change in major version upgrades but URIs might change.

The execution of a GET request on this URI returns a list of available functions.

GET `https://[server]:[port]/[version]/api`

Example: GET `https://localhost:8082/6.2/api`

Links

GET `6.2/api` allows the discovery of all available entry-points of the API.

The examples in this section show a sample of the HTTP response body in XML and JSON.

The structure of each entry-point is:

- href: The API's URI to the associated entry-point
- rel: A keyword that is the same in all version-specific entry-points

The OPTIONS method returns a list of the methods that you can use with each rel. Execute the OPTIONS method using the following syntax: OPTIONS `http://localhost:8082/6.2/<rel>`

Table 1: Examples of the OPTIONS method

| rel | OPTIONS method | Supported methods |
|----------|---|--|
| logout | OPTIONS <code>http://localhost:8082/6.2/logout</code> | OPTIONS, PUT To use the logout rel, use a PUT method. Example: PUT <code>http://localhost:8082/6.2/logout</code> |
| elements | OPTIONS <code>http://localhost:8082/6.2/elements</code> | HEAD, GET, OPTIONS To use the elements rel, use a GET method. Example: GET <code>http://localhost:8082/6.2/elements</code> |

For example, for the entry-point host, the API's URI is GET `6.2/elements/host`. Execution of GET `6.2/elements/host` returns all defined Host elements in the HTTP response body.

To log in, execute POST `6.2/login`.



Note: GET `6.2/api` does not give query parameter information. Query parameters are defined in the API documentation.

The execution of GET 6.2/api with Accept: application/xml returns the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<api>
  <entry_point href="http://localhost:8082/6.2/logout" rel="logout"/>
  <entry_point href="http://localhost:8082/6.2/elements" rel="elements"/>
  <entry_point href="http://localhost:8082/6.2/elements/sub_ipv6_fw_policy"
    rel="sub_ipv6_fw_policy"/>
  ...
</api>
```

The execution of GET 6.2/api with Accept: application/json returns the following:

```
{
  "entry_point": [
    {
      "href": "http://localhost:8082/6.2/logout",
      "rel": "logout"
    },
    {
      "href": "http://localhost:8082/6.2/elements",
      "rel": "elements"
    },
    {
      "href": "http://localhost:8082/6.2/elements/sub_ipv6_fw_policy",
      "rel": "sub_ipv6_fw_policy"
    },
    ...
  ]
}
```

Data element formats

You can retrieve elements from the API in both JSON and XML formats.

The format depends on the **Accept** HTTP header parameter. Accept: application/json returns elements in JSON. Accept: application/xml returns elements in XML.

Elements include at least the name and comment information. If administrative Domains are used, elements also include a link to the domains to which the elements belong. In addition, elements include two flags that show whether they are system and/or read-only. Custom elements have system and read-only attributes with a false value.

Elements must show the key attribute to be updated. This key attribute allows the API to identify the element. Elements show their specific attributes and elements as a content description.

By default, all attributes and elements from the data element input that are not supported are ignored. For this reason, we recommend to first retrieve an existing element in XML or JSON and then create an element or update the existing element.

For example, the “link” element is always ignored in data element input, the “key” attribute is always ignored in element creation, and the “system” and “read_only” attributes are always ignored.

JSON

The default data format for the API is JSON.

JSON is based on key/value arrays.

Example: The system "Your-Freedom Servers" host is represented in JSON as follows:

```
{
  "address": "66.90.73.46",
  "comment": "Your-Freedom Servers to help blocking access from Your-Freedom clients",
  "key": 86,
  "link": [
    {
      "href": "http://localhost:8082/6.2/elements/host/86",
      "rel": "self",
      "type": "host"
    }
  ],
  "name": "Your-Freedom Servers",
  "read_only": true,
  "secondary": [
    "193.164.133.72",
    ...
  ],
  "system": true,
  "third_party_monitoring": {
    "netflow": false,
    "snmp_trap": false
  }
}
```

The system and read-only flags are correctly set to `true` to indicate that the element in question is a system/read-only element. The name and comment attributes are correctly shown. In addition, there is more specific information — the address, the secondary address, and the "third_party_monitoring" status. Finally, the self verb is shown on the link row.

The primary IP address of this system host is 66.90.73.46. The host also has several secondary IP addresses, and third-party monitoring is disabled.

For more information about the JSON format, see <https://en.wikipedia.org/wiki/Json>.

XML

The API supports the XML format. As a standard, the XML format is more verbose than the JSON format.

The XML format appears more verbose particularly in collections, which present an XML tag grouping of all XML child elements.

Example: the system "Your-Freedom Servers" host has the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<host comment="Your-Freedom Servers to help blocking access from Your-Freedom clients"
address="66.90.73.46" key="86" name="Your-Freedom Servers" read_only="true" system="true">
  <links>
    <link href="http://localhost:8082/6.2/elements/host/86" rel="self" type="host"/>
  </links>
  <secondary_addresses>
    <secondary>193.164.133.72</secondary>
    ...
  </secondary_addresses>
  <third_party_monitoring netflow="false" snmp_trap="false"/>
</host>
```

There are common attributes with the JSON format — name, comment, key, system, and read-only. The attributes specific to XML are address (XML attribute), secondary_addresses, and third_party_monitoring (XML children elements).

For more information about the XML format, see <https://en.wikipedia.org/wiki/XML>.

Working with resource elements

You can manage resource elements with several operations.

Searching for resources

It is possible to filter each element entry point based on a specified part of a name, comment, or IP address.

For example, all elements can be listed with `GET 6.2/elements`, so it is possible to search all elements using the `192.168.*` IP address pattern with the following query:

```
GET 6.2/elements?filter=192.168.*
```

It is possible to filter specifically by type, for example, to get a list of all hosts with 'host' in their names or in their comments:

```
GET 6.2/elements/host?filter=host
```

Related concepts

[Specific searches](#) on page 15

Retrieving a resource

You can use a GET request on the specific API Client element's URI to retrieve the content of the element.

Example: After having retrieved the API's URI for hosts, `GET 6.2/elements/host` returns the following:

```
{
  "result":
  [
    {
      "href": "http://localhost:8082/6.2/elements/host/86",
      "name": "Your-Freedom Servers",
      "type": "host"
    },
    {
      "href": "http://localhost:8082/6.2/elements/host/39",
      "name": "DHCP Broadcast Destination",
      "type": "host"
    },
    ...
  ]
}
```

The HTTP request lists all defined hosts with their API's URIs. If a specific host is needed, search for the host by name to get a similar result but only including the particular host.

For example, `GET 6.2/elements/host/39` returns a 200 HTTP response status code and the specified XML/JSON description.



Tip: The `Accept` HTTP request header determines the output format (XML or JSON).

Creating a resource

To create an element, you need the associated element entry-point to execute a POST on it.

The API documentation describes all attributes that are needed for constructing elements in JSON or XML.

Example: For a host, the `POST 6.2/elements/host` REST call returns a 201 HTTP response status code and the created element API's URI in the HTTP header:

```
{
  "name": "myHost",
  "address": "192.168.0.1"
}
```



Tip: The **Content-Type** HTTP request header determines the input format (XML or JSON).

Updating a resource

When updating an element, the REST operation is a PUT.

First, you must execute a GET operation on the element to retrieve the ETag from the HTTP header.

After modifying the JSON/XML element content, you can execute a PUT operation with the ETag. The ETag is required to make sure that the version element is the most current version.

It is important to modify the results of the GET execution to make sure that all attributes are present for the update (for example, the key).

No merge is done for collections during an update. The API replaces the existing resource with the new one.

If the execution succeeds, it returns a 200 HTTP response status code and, in the HTTP header, the updated element API's URI.



Tip: The **Content-Type** HTTP request header determines the input format (XML or JSON).

Deleting a resource

When you want to delete an element, the REST operation is a DELETE.

Once you know the element API's URI, you can execute a DELETE operation on it. If the execution succeeds, it returns a 204 HTTP response status code.

Specific searches

The API makes it possible to execute specific searches, such as unused elements or duplicate IP addresses.

Searching for unused elements

This operation executes a search for all unused elements.

`GET 6.2/elements/search_unused`

It is also possible to filter this search by a specific name, comment, or IP address with the query parameter filter.

Searching for duplicate IP addresses

This operation executes a search for all duplicate IP addresses.

`GET 6.2/elements/search_duplicate`

It is also possible to filter this search by a specific name, comment, or IP address with the query parameter filter.

System information

This entry-point operation returns the current SMC version and the last activated Dynamic Update package.

`GET 6.2/system`

Examples

As you begin working with the SMC API, see these examples as a reference.

The following configuration is used for these examples:

- The SMC API is configured on port 8082, without host name restrictions, and reached from the same system as the Management Server.
- The SMC API entry-point URI is `http://localhost:8082/api`.
- An API Client element with the appropriate permissions and an authentication key of `sqfTm8UCd6havtycRP7P0001` has been defined in the Management Client.

Unless otherwise specified, all examples use JSON representations. The example elements (such as Helsinki FW and HQ Policy) derive names and properties from the elements that exist in the SMC installed in demo mode.

There are several python example scripts in the samples directory. Explanations of these samples are provided in the following sections.

Client access and logon

These tasks give the client access to the SMC through the API.

Version-specific entry point

You can create a list of all supported API versions and their entry points.

First, the client must retrieve the version-specific entry-point URI. A GET request on the API entry-point URI (<http://localhost:8082/api>) returns an array, named `version`, which lists all supported API versions and their entry-point URIs.

```
GET http://localhost:8082/api
Status Code: 200 OK
{
  "version": [
    {
      "href": "http://localhost:8082/6.1/api",
      "rel": "6.1"
    },
    {
      "href": "http://localhost:8082/6.2/api",
      "rel": "6.2"
    }
  ]
}
```

Global services and element URIs

The client must retrieve the login URI, as most services and element URIs require the client to be properly authenticated.

The version-specific URI declares the URIs for all elements and root services in a list named `entry_point`. The login URI is named `login`:

```
GET http://localhost:8082/6.2/api
Status Code: 200 OK
{
  "entry_point": [
    {
      "href": "http://localhost:8082/6.2/elements",
      "rel": "elements"
    },
    ...
    {
      "href": "http://localhost:8082/6.2/elements/host",
      "rel": "host"
    },
    ...
    {
      "href": "http://localhost:8082/6.2/login",
      "rel": "login"
    }
  ]
}
```

Logging on

Logon is performed with a POST request on the login service URI.

Before using protected services, clients must log on using their authentication key, which is generated when the API Client element is configured in the Management Client.

The API Client authentication key must be specified in the payload:

- For JSON content type {"authenticationkey":"XXXXX"}
- For XML content type <login_info authenticationkey='XXXXX' />

```
POST http://localhost:8082/6.2/login
Content-type: application/json
Payload : {"authenticationkey":"sqfTm8UCd6havtycRP7P0001"}
Status code: 200 OK
```

Working with hosts

Use the SMC API to find and configure hosts and Host elements.

Listing the hosts collection

After logon, you can get a list of all defined hosts with this request:

```
GET http://localhost:8082/6.2/elements/host
```

This request returns a 200 HTTP response status code and this result:

```
{
  "result":
  [
    {
      "href": "http://localhost:8082/6.2/elements/host/40",
      "name": "DHCP Broadcast Originator",
      "type": "host"
    },
    {
      "href": "http://localhost:8082/6.2/elements/host/43",
      "name": "IPv6 Unspecified Address",
      "type": "host"
    },
    ...
  ]
}
```

Filtering elements

Use filters to narrow your element search.

After logon, you can search for a specific host called Your-Freedom Servers with the following request:

```
GET http://localhost:8082/6.2/elements/host?filter=Your-Freedom Servers
```

This request returns a 200 HTTP response status code and this result:

```
{
  "result":
  [
    {
      "href": "http://localhost:8082/6.2/elements/host/86",
      "name": "Your-Freedom Servers",
      "type": "host"
    }
  ]
}
```

Creating a host

Create a host with the JSON format.

After login, create a Host element in the JSON format with the following request:

```
POST http://localhost:8082/6.2/elements/host
Request body:
{
  "name": "mySrc1",
  "comment": "My SMC API's my Src Host 1",
  "address": "192.168.0.13",
  "secondary": ["10.0.0.156"]
}
```

The request returns a 201 HTTP response status code and the following in the Location HTTP header:

`http://localhost:8082/6.2/elements/host/1704`

See `createHostThenDeleteIt.py` JSON or XML samples.

The system prevents you from creating an element without a unique name.

If you try to create an element with an existing name, you receive a 404 HTTP error status code and the following error message:

```
{
  "details":
  [
    "Element name fra-hide is already used."
  ],
  "message": "Impossible to store the element fra-hide."
}
```

Modifying an existing host

After login, you must first search for the host and then you can modify an existing host.

Search for the host using the filtering feature:

`GET http://localhost:8082/6.2/elements/host?filter=mySrc1`

After the element is found, use the following request:

`GET http://localhost:8082/6.2/elements/host/1704`

It returns the JSON host description and its ETag in the HTTP header in the following way:

```
Etag: MTCwNDMxMTEzNzQwNDMwNzMlNDQ=
{
  "address": "192.168.0.13",
  "comment": "My SMC API's my Src Host 2",
  "key": 1704,
  "link": [
    {
      "href": "http://localhost:8082/6.2/elements/host/1704",
      "rel": "self",
      "type": "host"
    }
  ],
  "name": "mySrc2",
  "read_only": false,
  "secondary": [
    "10.0.0.156"
  ],
  "system": false,
  "third_party_monitoring": {
    "netflow": false,
    "snmp_trap": false
  }
}
```

From the JSON content, you can update the host as needed (add attributes, or add, remove, or update hosts).

```
PUT http://localhost:8082/6.2/elements/host/1704
```

Using Etag: MTCwNDMxMTEzNzQwNDMwNzMlNDQ= as the HTTP request header, and the updated JSON content as the HTTP request payload, returns a 200 HTTP response status code and the following in the HTTP response header:

```
Location: http://localhost:8082/6.2/elements/host/1704
```

See `updateHostThenDeleteIt.py` JSON or XML samples.

Deleting a host

After logon, you must first search for the host and then you can find and delete an existing host.

Search for the host using the filtering feature:

```
GET http://localhost:8082/6.2/elements/host?filter=mySrc1
```

```
http://localhost:8082/6.2/elements/host/1704
```

After the host is found, using the following request returns a 204 HTTP response status code:

```
DELETE http://localhost:8082/6.2/elements/host/1704
```

See `createHostThenDeleteIt.py` and `updateHostThenDeleteIt.py` JSON or XML samples.

Working with IP address lists

Use the SMC API to import IP address lists.

Listing existing IP address lists

After logon, you can get a list of all defined IP Address List elements with this request:

```
GET http://localhost:8082/6.2/elements/ip_list
```

This request returns a 200 HTTP response code and this result:

```
{
  "result":
  [
    {
      "href": "http://localhost:8082/6.2/elements/ip_list/667",
      "name": "new_ip_list",
      "type": "ip_list"
    },
    {
      "href": "http://localhost:8082/6.2/elements/ip_list/312",
      "name": "Skype Servers IP Address List",
      "type": "ip_list"
    },
    ...
  ]
}
```

Creating an IP address list

IP address lists are created in two phases: first create the IP Address List element, then upload the content to the IP Address List element.

After logon, create an IP Address List element with the following request:

```
POST http://localhost:8082/6.2/elements/ip_list
Request body:
{
  "name": "myIpList1",
  "comment": "My SMC API's my IP Address List 1"
}
```

The request returns a 201 HTTP response status code and the following in the Location HTTP header:

`http://localhost:8082/6.2/elements/ip_list/1704`

Upload the content for the created IP Address List element with the following request:

```
POST http://localhost:8082/6.2/elements/ip_list/1704/ip_address_list
Request body:
{
  "ip":
  [
    "1.2.3.4",
    "10.0.0.0/8",
    "192.168.0.0-192.168.255.255"
  ]
}
```

On success, the request returns HTTP response status code 202.

You can modify the content of the IP Address List element by uploading new content for an existing IP Address List element. The existing content is overwritten by the content in the request.

To read the content of the IP Address List element, call GET to the same URI as above:

`GET http://localhost:8082/6.2/elements/ip_list/1704/ip_address_list`

On success, the response code is 200 and the content is provided in an identical format to the content upload above:

```
{
  "ip":
  [
    "1.2.3.4",
    "10.0.0.0/8",
    "192.168.0.0-192.168.255.255"
  ]
}
```

See `createIpAddressListModifyAndDeleteIt.py` JSON and XML samples.

Working with Policy elements

The SMC API can be used to modify, upload, and monitor the status of Policy elements.

Modifying a rule in a policy

You can modify a rule within an existing policy.

After login, you must first search for the policy using the filtering feature:

```
GET http://localhost:8082/6.2/elements/fw_policy?filter=HQ Policy
```

After the policy is found, you can retrieve a specific type of rule with the following request:

```
http://localhost:8082/6.2/elements/fw_policy/56
```

```
GET http://localhost:8082/6.2/elements/fw_policy/56
```

These special links to the Firewall Policy retrieve all applicable rules in the current policy:

- `fw_ipv4_access_rules` — Retrieves all Firewall IPv4 Access rules
- `fw_ipv6_access_rules` — Retrieves all Firewall IPv6 Access rules
- `fw_ipv4_nat_rules` — Retrieves all Firewall IPv4 NAT rules
- `fw_ipv6_nat_rules` — Retrieves all Firewall IPv6 NAT rules

For example, in Firewall IPv4 Access rules, the first rule is `@514.0`:

```
{
  "href": "http://localhost:8082/6.2/elements/fw_policy/56/fw_ipv4_access_rule/514",
  "name": "Rule @514.0",
  "type": "fw_ipv4_access_rule"
}
```

```
GET http://localhost:8082/6.2/elements/fw_policy/56/fw_ipv4_access_rule/514
```

The content of the `@514` Firewall IPv4 Access rule is retrieved:

```
{
  "comment": "Set logging default, set long timeout for SSH connections",
  "is_disabled": false,
  "key": 2543,
  "link": [
    {
      "href": "http://localhost:8082/6.2/elements/fw_policy/56/fw_ipv4_access_rule/514",
      "rel": "self",
      "type": "fw_ipv4_access_rule"
    }
  ],
  "parent_policy": "http://localhost:8082/6.2/elements/fw_policy/56",
  "rank": 4,
  "read_only": false,
  "system": false,
  "tag": "514.0"
}
```

The result has ETag: `MjU0Mzk4MTEzMDYyMzMzMzYxMTg=` as the HTTP response header.

This rule seems to be a comment rule (no source/destination/service attributes are defined), so you could update the comment, for example:

```
PUT http://localhost:8082/6.2/elements/fw_policy/56/fw_ipv4_access_rule/514
```

The new JSON content with the updated comment and ETag: MjU0Mzk4MTEzMMDYyMzMzMzYxMTg= as the HTTP request header returns a 200 HTTP response status code and the following in the HTTP response header:

`http://localhost:8082/6.2/elements/fw_policy/56/fw_ipv4_access_rule/514`

See `addRuleAndUpload.py` JSON or XML samples.

Uploading a policy and monitoring its status

There are two ways of uploading or refreshing a policy — from the engine and from the policy.

To upload a policy from the engine, you must first search for the engine after logging in using the filtering feature:

`GET http://localhost:8082/6.2/elements?filter=Helsinki FW`

Engine

After the engine has been retrieved, the following JSON content is displayed:

```
{
  "link":
  [
    {
      "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/refresh",
      "rel": "refresh"
    },
    {
      "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/upload",
      "rel": "upload"
    },
    ...
  ]
}
```

Policy

The verb 'upload' is listed, so you can execute the following request:

`POST http://localhost:8082/6.2/elements/fw_cluster/1563/upload?filter=HQ Policy`

By filtering the REST call with the HQ Policy, you enable the upload of the HQ Policy on the Helsinki Firewall Cluster.

This results in the 201 HTTP response status code and the following:

```
{
  "follower": "http://localhost:8082/6.2/elements/fw_cluster/1563/upload/
NWYyMDBiOTA4ZTY3NDM0ZTotNzgyM2JmMmI6MTNmZWxMGI3ZGY6LTdmZDA=",
  "href": "http://localhost:8082/6.2/elements/fw_cluster/1563",
  "in_progress": true,
  "last_message": "",
  "success": true
}
```

To follow up on the upload, you can periodically request for its status in the following way:

`GET http://localhost:8082/6.2/elements/fw_cluster/1563/upload/
NWYyMDBiOTA4ZTY3NDM0ZTotNzgyM2JmMmI6MTNmZWxMGI3ZGY6LTdmZDA=`

For as long as the attribute `in_progress` is not set to false, the upload continues with a new `last_message` attribute.

It is also possible to refresh a policy on the engine. As you can see from the engine links, the verb 'refresh' is also available on the engine:

`POST http://localhost:8082/6.2/elements/fw_cluster/1563/refresh`

This process ends in the same way as an upload. The engine must have a policy already installed to proceed to the upload.

See `addRuleAndUpload.py` JSON or XML samples.

Working with VPNs

You can use the SMC API to configure gateways, certificates, VPN topology, and settings for VPNs. The following data elements are used in VPN configuration.

Table 2: Data elements for VPN configuration

| Data element | Data type | Parent element | Actions |
|-----------------------------|-----------------------------|--------------------------------------|---|
| vpn | VPN | elements | none |
| vpn_profile | VPN Profile | elements | none |
| gateway_profile | Gateway Profile | elements | none |
| gateway_settings | Gateway Settings | elements | none |
| gateway_certificate | Gateway Certificate | internal_gateway | certificate_export, renew |
| gateway_certificate_request | Gateway Certificate Request | internal_gateway | certificate_import, certificate_export, self_sign |
| internal_gateway | Internal Gateway | single_fw, fw_cluster, master_engine | generate_certificate |
| external_gateway | External Gateway | elements | none |
| vpn_certificate_authority | VPN Certificate Authority | elements | certificate_import, certificate_export |

Data elements for VPN configuration support the following standard operations:

- List (GET)
- Read (GET)
- Create (POST)



Note: The gateway_certificate and gateway_certificate_request data elements do not support the Create (POST) operation. You must use the generate_certificate action for the internal_gateway data element to create gateway_certificate and gateway_certificate_request data elements.

- Modify (PUT)



Note: The gateway_certificate and gateway_certificate_request data elements do not support the Modify (PUT) operation. You must use the generate_certificate action for the internal_gateway data element to modify gateway_certificate and gateway_certificate_request data elements.

- Delete (DELETE)

Viewing information about VPNs

You can use GET requests to list VPNs and view information about VPNs.

After login, use this request to list all defined VPNs:

```
GET http://localhost:8082/6.2/elements/vpn
```




Note: In all examples, the VPN's ID number is 5.

After logon, use this request to view information about a VPN:

GET `http://localhost:8082/6.2/elements/vpn/5`

This request returns a 200 HTTP status response code and this result:

```
{
  "key":5
  "link":[
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/gateway_tree_nodes/central",
      "rel":"central_gateway_node"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite",
      "rel":"satellite_gateway_node"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/gateway_tree_nodes/mobile",
      "rel":"mobile_gateway_node"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/open",
      "rel":"open"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/save",
      "rel":"save"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/close",
      "rel":"close"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/validate",
      "rel":"validate"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/tunnels",
      "rel":"gateway_tunnel"
    },
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5",
      "rel":"self",
      "type":"vpn"
    }
  ],
  "mobile_vpn_topology_mode":"None",
  "name":"Corporate VPN",
  "nat":false,
  "read_only":false,
  "system":false,
  "vpn_profile":"http://127.0.0.1:8082/6.2/elements/vpn_profile/1"
}
```

Opening a VPN topology

You must open a VPN topology before you can modify it.



Note: Only one VPN topology can be opened at a time for each HTTP session.

Open a VPN topology with this request:

```
POST http://localhost:8082/6.2/elements/vpn/5/open
```

This request returns a 200 HTTP status response code. You are now able to query inside the VPN topology.

Viewing information about gateway-to-gateway tunnels

You can use GET requests to list the tunnels between gateways in a VPN, and to view information about a specific gateway-to-gateway tunnel.

After opening the VPN topology, use this request to list the gateway-to-gateway tunnels in the VPN:

```
GET http://localhost:8082/6.2/elements/vpn/5/tunnels
```

This request returns a 200 HTTP status response code and this result:

```
{
  "result":
  [
    {
      "href":"http://127.0.0.1:8082/6.2/elements/vpn/5/tunnels/ADcAOw==",
      "name":"Gateway Tunnel 55-59",
      "type":"gateway_tunnel"
    },
  ]
}
```

Use this request to view information about a specific gateway-to-gateway tunnel:

```
GET http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==
```

This request returns a 200 HTTP status response code and this result:

```
{
  "enabled":true,
  "gateway_node_1":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/55",
  "gateway_node_2":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/central/59",
  "key":0,
  "link":[
    {
      "href":"http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==",
      "rel":"self",
      "type":"gateway_tunnel"
    },
    {
      "href":"http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==/endpoints",
      "rel":"gateway_endpoint_tunnel",
      "type":"gateway_endpoint_tunnel"
    }
  ],
  "preshared_key":"iAb28nUyhweaQ6nLLrBG7NyaxJc48zFU5nn9HphYGVpZcrupy
WiEf47z6JENq2EXZXgceStoRArsMxHQqoDY9wCnaVGns3Vv4G9rcm6X9EPQRqaQBq
pprfKAHEToaCMR97rE7dq9BBJFD"
}
```

Viewing information about endpoint-to-endpoint tunnels

You can use GET requests to list the tunnels between endpoints in a VPN, and to view information about a specific endpoint-to-endpoint tunnel.

After opening the VPN topology, use this request to list the endpoint-to-endpoint tunnels for a specific gateway:

```
GET http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==/endpoints
```

The request returns a 200 HTTP status response code and this result:

```
{
  "result":
  [
    {
      "href": "http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==/endpoints/AGoAcg==",
      "name": "Gateway EndPoint Tunnel 106-114",
      "type": "gateway_endpoint_tunnel"
    },
  ]
}
```

Use this request to view information about a specific endpoint-to-endpoint tunnel:

GET <http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==/endpoints/AGoAcg==>

The request returns a 200 HTTP status response code and this result:

```
{
  "enabled": true,
  "endpoint_1": "http://localhost:8082/6.2/elements/fw_cluster/1554/internal_gateway/55/internal_endpoint/106",
  "endpoint_2": "http://localhost:8082/6.2/elements/fw_cluster/1563/internal_gateway/59/internal_endpoint/114",
  "key": 0,
  "link": [
    {
      "href": "http://localhost:8082/6.2/elements/vpn/5/tunnels/ADcAOw==/endpoints/AGoAcg==",
      "rel": "self",
      "type": "gateway_endpoint_tunnel"
    }
  ]
}
```

Viewing information about a gateway

You can use GET requests to view information about gateways in a VPN topology.

There are two requests for viewing information about gateways in the VPN topology:

- GET http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/central — Shows information about gateways on the central gateways list.
- GET http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite — Shows information about gateways on the satellite gateways list.

After opening the VPN topology, use this request to view information about a specific gateway node:

GET http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67

The request returns a 200 HTTP status response code and this result:

```
{
  "child_node":["http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/65"],
  "gateway":"http://localhost:8082/6.2/elements/fw_cluster/1588/internal_gateway/67",
  "key":67,
  "link":[
    {
      "href":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67",
      "rel":"self",
      "type":"satellite_gateway_node"
    },
    {
      "href":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67/sites/
enabled",
      "rel":"vpn_site",
      "type":"vpn_site"
    },
    {
      "href":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67/sites/
disabled",
      "rel":"vpn_site",
      "type":"vpn_site"
    }
  ],
  "node_usage":"satellite",
  "parent_node":"http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66",
  "vpn_key":5
}
```



Note: The gateway_tree_nodes data element represents nodes in the VPN topology tree. The internal_gateway attribute gives access to the gateway data element.

Adding a gateway node to the VPN topology

You can use POST requests to add a gateway node to the VPN topology.

After opening the VPN topology, use this request to add an internal_gateway data element to the gateway_tree_nodes in the VPN topology:

```
POST http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite
{"gateway":"http://localhost:8082/6.2/elements/fw_cluster/1557/internal_gateway/56",
"node_usage":"central"}
```



Note: The gateway_tree_nodes data element represents nodes in the VPN topology tree. The internal_gateway attribute gives access to the gateway data element.

Deleting a gateway node from the VPN topology

You can use DELETE requests to delete a gateway node from the VPN topology.



Note: Deleting a gateway node from the VPN topology does not delete the internal_gateway or external_gateway data element.

After opening the VPN topology, use this request to delete a gateway node from the VPN topology:

```
DELETE http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67
```

Moving a gateway node in the VPN topology

You can move a gateway node to the central or satellite gateways list, or move a gateway node under a parent node in the VPN topology.

After opening the VPN topology, use this request to move a gateway node from the central gateways list to the satellite gateways list in the VPN topology:

```
PUT http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/central/67
"node_usage":"satellite"
```

After opening the VPN topology, use this request to move a gateway node from the satellite gateways list to the central gateways list in the VPN topology:

```
PUT http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67
"node_usage":"central"
```

After opening the VPN topology, use this request to move a gateway node under a parent node in the VPN topology:

```
PUT http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/67
"parent_node":http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66
```

Listing the sites in a VPN

You can use GET requests to list the enabled and disabled sites in a VPN.

After opening the VPN topology, use these requests to list the sites associated with a central gateway in a VPN:

- GET `http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/central/88/sites/enabled` — Lists the enabled sites associated with the central gateway.
- GET `http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/central/88/sites/disabled` — Lists the disabled sites associated with the central gateway..

After opening the VPN topology, use these requests to list the sites associated with a satellite gateway in a VPN:

- GET `http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/enabled` — Lists the enabled sites associated with the satellite gateway.
- GET `http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/disabled` — Lists the disabled sites associated with the satellite gateway.

After opening the VPN topology, use this request to list the enabled sites associated with a satellite gateway in the VPN:

```
GET http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/enabled
```



Note: In this example, the satellite gateway's ID number is 66.

This request returns a 200 HTTP status response code and this result:

```
{
  result: [2]
  0: {
    href: "http://127.0.0.1:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/enabled/68"
    name: "vpn_site 68"
    obsolete: false
    type: "vpn_site"
  }-
  1: {
    href: "http://127.0.0.1:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/enabled/61"
    name: "vpn_site 61"
    obsolete: false
    type: "vpn_site"
  }-
  -
}
```

Enabling or disabling sites in a VPN

You can use DELETE requests to change the status of a site in a VPN.



Note: The DELETE request toggles the status of the site. Sites in the enabled sites list are moved to the disabled sites list. Sites in the disabled sites list are moved to the enabled sites list.

After opening the VPN topology, use this request to disable a site in a VPN:

```
DELETE http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/enabled/68
```

The site is removed from the enabled sites list and added to the disabled sites list.

After opening the VPN topology, use this request to enable a site in a VPN:

```
DELETE http://localhost:8082/6.2/elements/vpn/5/gateway_tree_nodes/satellite/66/sites/disabled/68
```

The site is removed from the disabled sites list and added to the enabled sites list.

Validating a VPN topology

You can use the SMC API to retrieve a list of VPN topology validation issues.



Note: In this example, the validation has already been done. The example request queries the result of the validation. It does not trigger the validation action.

There are three kinds of validation issues:

- VPN Global Issues — Issues that affect the whole VPN.
- GwGw Issues — Issues that affect tunnels between gateways.
- EpEp Issues — Issues that affect tunnels between endpoints.

Retrieve the list of VPN topology validation issues with this request:

```
GET http://localhost:8082/6.2/elements/vpn/5/validate
```

This request returns a 200 HTTP status response code and this result:

```
{
  "value": "VPN Topology validation detects some warnings/errors for VPN 5, please check it:
  GwGw Issues:
    - 66<->67
      -- WARNING: The Gateway Riyadh VPN Gateway is a hub in the Overall Topology, but has no Site in
      Hub mode in this VPN.
    - 65<->67
      -- WARNING: The Gateway Tunis VPN Gateway is a hub in the Overall Topology, but has no Site in Hub
      mode in this VPN."
}
```

Saving a VPN topology

When you finish changing a VPN topology, save the VPN topology.



Note: Saving a VPN topology is resource-intensive. Avoid excessive save operations.

To save the VPN topology, use this request:

POST <http://localhost:8082/6.2/elements/vpn/5/save>

This request returns a 200 HTTP status response code.

Closing a VPN topology

When you finish working with a VPN topology, close the VPN topology.



CAUTION: Closing the VPN topology without saving the VPN topology discards the changes.

To close the VPN topology, use this request:

POST <http://localhost:8082/6.2/elements/vpn/5/close>

This request returns a 200 HTTP status response code.

Filtering searches by group type

It is possible to filter searches by group type.

These search context groups are currently available:

- **Network_elements** — Search for all Network elements. Network elements are used in the Source/Destination cells in the **Policy Editing** view.
- **Services** — Search for all services. Services are used in the Service cell in the **Policy Editing** view.
- **Services_and_applications** — Search for all Services and Applications. Services and Applications are used in the Service cell in the **Policy Editing** view.
- **Tags** — Search for all tags. Tags are used in the **Policy Editing** view for Inspection rules.
- **Situations** — Search for all Situations. Situations are used in the **Policy Editing** view for Inspection rules.

For example, the REST call could have the following content:

```
https://[server]:[port]/[version]/elements?
filter=NameOfElement&filter_context=ElementTypeOrSearchContextGroup
```

In this example, `ElementTypeOrSearchContextGroup` can be either the type of the element, like `host / address_range / ...`, or `network_elements / services / services_and_applications / tags / situations`. Lists of element types are also supported. For example, “host, router, network” can be used to filter the types to host, router, or network elements.

Retrieving routing/antispoofing information

You can retrieve static or dynamic routing information from an engine.

To retrieve the complete (static/dynamic) routing information from an engine, you can execute the following request:

```
GET /[version]/elements/[cluster_type]/[cluster_key]/routing/[routing_key]
```

To retrieve antispoofing information, you can execute the following request:

```
GET /[version]/elements/[cluster_type]/[cluster_key]/antispoofing/[antispoofing_key]
```

For example, for the Helsinki Firewall Cluster, you would have the following:

```
"link":
[
...
{
  "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/routing/887",
  "rel": "routing",
  "type": "routing"
},
{
  "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/antispoofing/990",
  "rel": "antispoofing",
  "type": "antispoofing"
},
...
],
```

To access the routing information, you must use the routing link:

```
GET http://localhost:8082/6.2/elements/fw_cluster/1563/routing/887
```

The routing link returns a 200 HTTP response status code and the following:

```
{
  "href": "http://localhost:8082/6.2/elements/fw_cluster/1563",
  "ip": "10.8.0.21",
  "key": 887,
  "level": "engine_cluster",
  "link":
  [
    {
      "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/routing/887",
      "rel": "self",
      "type": "routing"
    }
  ],
  "name": "Helsinki FW",
  "read_only": false,
  "routing_node":
  [
    {
      "exclude_from_ip_counting": false,
      "href": "http://localhost:8082/6.2/elements/fw_cluster/1563/physical_interface/276",
      "key": 888,
      "level": "interface",
      "name": "Interface 0",
      "nic_id": "0",
      "read_only": false,
      "routing_node":
      [

```


To access the antispoofing information, you must use the antispoofing link:

GET http://localhost:8082/6.2/elements/fw_cluster/1563/antispoofing/990

The antispoofing link returns a 200 HTTP response status code and the following:

```
...
"auto_generated": "true",
"href": "http://localhost:8082/6.2/elements/fw_cluster/1563/tunnel_interface/343",
"key": 1333,
"level": "interface",
"name": "Tunnel Interface 1002",
"nic_id": "1002",
"read_only": false,
"system": false,
"validity": "enable"
},
{
"auto_generated": "true",
"href": "http://localhost:8082/6.2/elements/fw_cluster/1563",
"ip": "10.8.0.21",
"key": 990,
"level": "engine_cluster",
"link":
[
{
"href": "http://localhost:8082/6.2/elements/fw_cluster/1563/antispoofing/990",
"rel": "self",
"type": "antispoofing"
}
],
"name": "Helsinki FW",
"read_only": false,
"system": false,
"valid"
}
}
```

